CHAIN ROUTING:
A NOVEL ROUTING FRAMEWORK FOR INCREASING RESILIENCE AND STABILITY IN THE INTERNET

by

PEDRO DAVID ARJONA VILLICAÑA

A thesis submitted to
The University of Birmingham
for the degree of
DOCTOR OF PHILOSOPHY

Department of Electronic, Electrical and Computer Engineering
College of Engineering and Physical Sciences
The University of Birmingham

September 2009

Abstract

This study investigates the Internet's resilience to instabilities caused by the mismatch of its topological state and routing information. A first numerical analysis proves that the Internet possesses unused path diversity which could be employed to strengthen its resilience to failures. Therefore, a new routing framework called Chain Routing, which takes advantage of such path diversity, is proposed. This novel idea is based in the mathematical concept of complete order, which is a binary relation that is irreflexive, asymmetric, transitive and complete. More important is the fact that complete orders, when represented as a graph, are the most connected digraph that does not contain any cycles. Consequently, a complete order could be applied to route information from a source to a destination with the guarantee that cycles will not develop in a path. A second numerical analysis demonstrates the feasibility of implementing Chain Routing as part of a routing protocol. Finally, an analysis is presented on how network stability could be maintained if a routing protocol integrates complete orders in time and topology.

This work is dedicated to Gaby.

Acknowledgements

I would like to thank my supervisor, Costas Constantinou, for providing the expertise and guidance needed to fulfil this research: It is nice to have a good supervisor, but it is even better to have someone who inspires you for life. I also need to thank Alexander Stepanenko (Sanya) for the ideas and support he provided for this project.

This PhD was mostly funded by CONACYT and the University of Birmingham, hence I am in debt with both, Mexico and the United Kingdom.

I would like to thank my personal friends: David Delgado, Juan Carlos Cuevas, María Barilla, Rene Brueckner and Aida Hidalgo. I also need to thank my friends at the Communications Department who provided me with their support and help: Amalia Tsanaka, Debra Topham, Keita Rose, Paul Kiddie and Yuriy Nechayev.

In a more personal level, I would like to thank my parents, David Arjona and Bricia Villicaña, because with their love and care I have become who I am. ¡Todo mi amor para ustedes dos! Special thanks to my parents in law, Pedro Silva and Tayde Maceda, because they have always offered their unconditional support.

The serious problems and delays to this research were caused by the greatest joy and inspiration I received: my son Pablo. ¡Tu mamá y yo te adoramos!

Finally, I give special thanks to my Muse, my best friend and my love: when you smile life is colourful, flavoursome and delightful. ¡Te amo, Gaby!

Contents

List of Figures

List of Tables

List of Abbreviations

AS	Autonomous System [1.1]
ASN	Autonomous System Number
ATM	Asynchronous Transfer Mode
BDP	Bounded Divergence Protocol
BFS	Breadth-First Search (algorithm)
BGP	Border Gateway Protocol [2.1]
BRT	BGP Routing Table (data)
CAIDA	The Cooperative Association for Internet Data Analysis
CCR	Congestion Chain Reaction
CIDR	Classless Interdomain Routing
CRDS	Chain Routing Data Structure [6.2.1]
EGP	Exterior Gateway Protocol
EPIC	Enhanced Path-vector Routing Protocol
FESN	Forward Edge Sequence Numbers
HLP	Hybrid Link State and Path Vector Protocol
ICMP	Internet Control Message Protocol
IGP	Interior Gateway Protocol
IP	Internet Protocol
IS-IS	Intermediate System to Intermediate System

ISP	Internet Service Provider
MED	Multiple Exit Discriminator (parameter)
MIRO	Multi-path Interdomain Routing Protocol
MOAS	Multiple Origin Autonomous System
MRT	Multi-threaded Routing Toolkit
NLRI	Network Layer Reachability Information (UPDATE message)
OSPF	Open Shortest Path First
PPR	Preferred Path Rule [2.1]
PRO	Persistent Route Oscillations [2.2.2]
R-BGP	Resilient BGP
RCN	Route Cause Notification
RIB	Routing Information Base
RIP	Routing Information Protocol
RIPE	Réseaux IP Européens
RIR	Regional Internet Registry
RIS	Routing Information Service
RPSL	Routing Policy Specification Language
RRC	Remote Route Collector
SONET	Synchronous Optical Networking
SPVP	Simple Path-Vector Protocol
TORA	Temporally-Ordered Routing Algorithm
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

Chapter 1

Introduction

The Internet has become an essential part of today's economic, academic and social activities. This global network is bound together by a single routing protocol, the Border Gateway Protocol or BGP, which means that the Internet's reliability depends on the strengths and weaknesses of this routing protocol. This thesis presents the efforts made to analyse the current functionality offered by BGP, and introduces a new routing paradigm that is arguably better suited to utilise the Internet's topology.

This research starts by studying the problems and difficulties that BGP experiences as it routes information through the Internet. Such analysis reveals that, despite many efforts to improve BGP's resilience and performance, this protocol is not able to exploit the richness of path diversity available in the Internet and also, in special circumstances, may become unstable.

To overcome these two problems a new routing mechanism is proposed. Such a mechanism is the product of research on the topology of the Internet, the restrictions imposed by policies and the mathematical tools used to model, understand and exploit these relationships. This new proposal, called Chain Routing, is a routing framework designed to take advantage of the path diversity available in the network while, at the

same time, avoids the development of unstable behaviour.

1.1 Routing in the Internet

In order to perform routing in a network that experiences continuous growth, the Internet needed to follow a **hierarchical routing** model in which individual computers belong to small networks that also belong to larger networks that interconnect to each other in the cyberspace. This hierarchical model requires that each of the larger networks which form the Internet could be perceived as a single entity called an Autonomous System (AS). Because ASs are the main blocks that form the Internet and because this study analyses the topology and connectivity of these ASs, the following precise definition applies in this thesis:

Definition 1 (from RFC 4271 [1])**.** *"An **Autonomous System (AS)** is a set of routers under a single technical administration, using an interior gateway protocol (IGP) and common metrics to determine how to route packets within the AS, and using an inter-AS routing protocol to determine how to route packets to other ASs."*

Each AS usually belongs to a single organisation or company, like Level 3, British Telecom, or MCI. Their **interior gateway protocols (IGP)** are meant to provide good performance at the cost of limiting the size of the network in which they operate. Examples of such protocols are RIP, OSPF and IS-IS. Meanwhile, **exterior gateway protocols (EGP)** need to support policy-based routing and cannot limit the size of, or even know, the entire topology of the Internet. IGPs are also known as **intra-AS routing protocols**; likewise EGPs are also called **inter-AS routing protocols**.

Currently, the only EGP in use is the **Border Gateway Protocol**, or **BGP**. Two important characteristics that have contributed to its dominance are its ability to work

without global coordination and its flexibility to define policies. The first characteristic allows ASs great independence at selecting routes to destinations while, at the same time, helps to maintain scalable overheads for these same destinations. Meanwhile, policies provide a means to modify BGP's basic routing functionality in order to allow operators to select routes that are better suited to the AS because of other less obvious reasons. For example: traffic managment or meeting commercial agreements. How well BGP manages to perform adaptive routing in the Internet, specially when failures occur, will be further discussed in Chapter 2.

1.2 Routing with imperfect topological information

Most routing problems originate when the network's topology changes and routers fail to correctly interpret this change as either transient or permanent topological modifications or a set of inconsistent messages [2–16]. Therefore, this study starts with the premise that *the main cause of any instability is the inconsistency or imperfection of the topological information that may exist between any router and the current state of the network*. If we know that fault and change are given events in a network, is it possible to design a protocol that anticipates and deals with these inconsistencies? The main aim behind this research is to determine if it is possible to design a routing protocol that is resilient to imperfect topological information.

Previous research activities dealing with fault-tolerant routing have concentrated on how to correct the inconsistencies that may occur in a network for a specific routing protocol, such as BGP, or OSPF. These studies aim to define lower and upper bounds to different metrics that may help to reduce the effects of the faults under study [4–8, 12–15]. Others have focused their attention on the topological structure of the Internet without providing further consideration on how instabilities originate, or the ef-

fects they may produce on the network [3, 17–21]. This has inspired a large amount of research on instabilities without considering the network's topology, as well as a considerable number of studies on Internet topology that do not analyse fault-tolerant mechanisms.

Most studies that discuss network instability and inconsistent topological information in the Internet largely focus on BGP because of the following two reasons: (1) BGP is the only routing protocol used to route traffic between ASs; (2) this routing protocol was designed to operate without knowing the complete topology of the Internet. Because of this second reason, it is possible to say that, in principle, BGP performs routing with imperfect topological information. Although nobody can deny that BGP is well-suited to perform routing in the Internet, many authors have pointed out that this protocol is prone to instabilities, oscillatory behaviour and sub-optimal routing solutions. Not only that, but because BGP should only use a single path between a source and a destination at any given time, it does not allow the proper implementation of traffic engineering. A comprehensive analysis of these issues is specifically addressed in Chapter 2, and some of the ideas that have been proposed to solve these problems are reviewed in Chapter 3.

The opportunity to develop adaptive routing in the Internet without producing instabilities is still open. This dissertation proposes and explores the reasons why a first step to a solution could rest on the use of the mathematical concept of complete order, also known as a chain.

1.3 Chain Routing

Communication networks are usually modelled in the literature as a(n) (undirected) **graph** $G = (V, E)$, which is formed by a set V of **vertices** joined by a set E of

undirected **edges**. This study will demonstrate in Chapter 4 that, in order to model how BGP propagates destinations in the Internet, it is appropriate to use a **directed graph** or **digraph** $D = (V, A)$. This graph also consists of a set V of vertices but, in opposition to undirected graphs, these are joined through a set A of **arcs** that possess a specific direction, or order.

Chapter 4 also describes a numerical analysis designed to obtain a realistical topological representation of the Internet using digraphs. This representation is then used to find the connectivity available, but which remains unused, in this network. This numerical experiment suggests that it is possible to design a routing mechanism that is more resilient than BGP by taking advantage of the currently unexploited path diversity of the Internet.

Further research on mathematical graphs uncovered **partial orders** as a type of digraph suitable to model path diversity in a network. This digraph models mathematical relations that possess the following properties: irreflexivity, asymetry and transitivity. A partial order that is complete (the relations exist for all the vertices in the digraph) is called a **complete order**, **total order**, **linear order**, or simply a **chain**. A very important property of chains is that they are the maximal digraphs that do not contain any cycle. Complete orders will be analysed in Chapter 5.

By taking advantage of the directionality and the abscence of cycles that chains possess, it is possible to develop a routing framework that is more stable and consistent than the current approach followed by BGP. This new routing mechanism is called **Chain Routing**, and along with its increased stability, it allows easy implementation of traffic engineering and avoids the development of oscillatory behaviours. The cost incurred for providing all these features is an increased coordination between the ASs that would belong to the proposed routing framework. Chain Routing is further developed and analysed in Chapter 6.

The interactions between Chain Routing and the most important Internet instabilities are analysed in Chapter 7. This chapter also provides the conditions under which network stability could be maintained, and provides some suggestions and remarks for the future implementation of the proposed routing framework. Final considerations as well as the following steps that may be needed to develop Chain Routing are discussed in Chapter 8.

1.4 Scope of this study

As in other studies of similar nature, there are some assumptions that needed to be made as well as some limitations to the scope of this investigation.

The most important assumption made in this study is that instabilities in the Internet are caused by the inconsistency between the current state of the network and the information that their routers possess and use to exploit its topology to forward data reliably to its destination. This assumption caused some factors to be excluded from this study. The most important one is arguably network security, because in the context of this research there is no difference between a network failure caused by a security problem (malicious attack) and a legitimate network fault.

Another important simplification made for this study is considering ASs as independent and isolated nodes which form the Internet. Although this simplification allows to use network models which are easier to manage, they also fail to include the subtle interactions that may exists between IGPs and EGPs.

Finally, the Chain Routing program, developed to prove that this routing mechanism is a feasable idea, is just a first step towards the implementation of the routing framework proposed in this thesis, and it was not further refined because of the large amount of time required to realise a full implementation of this system.

Chapter 2

Routing in Today's Internet

The fact that the current version of BGP is still in use after more than a decade may be considered a proof of its robustness as a routing protocol. Still, there are many concerns about the ability of this routing protocol to continue performing in a network that continues to grow at an impressive rate [22]. A brief description of BGP and an analysis of the characteristics that are relevant to this study are included in section 2.1. Then, an analysis of the associated problems that have been discovered in the Internet is provided in section 2.2. Finally, section 2.3 gives some examples of potential difficulties that could occur if the issues discussed in this chapter are not addressed.

2.1 The Border Gateway Protocol

The current version of the Border Gateway Protocol, BGP-4, was orignally deployed in 1993 [23] under RFC 1771 [24], and was subsequently replaced in January 2006 by RFC 4271 [1].

BGP-4 is a complex routing protocol and its comprehensive description is outside the scope of this study. The objective of this section is to describe BGP's features that

have a direct relation to the propagation of destination address advertisements in the Internet. For further information on BGP it is advisable to consult RFC 4271 [1] and books specialised in this routing protocol, like the one by Zhang and Bartell [25].

BGP has been classified as a path-vector routing protocol that announces **Classless Interdomain Routing (CIDR)** prefixes as destinations. Two routers that share a BGP session are called **BGP peers**, and only four different types of messages are needed to start and mantain a session between BGP peer routers:

OPEN This is the first message a BGP router sends to its peer to start a BGP session.

KEEPALIVE This is a simple message that must be sent periodically to mantain connectivity between peers. When a router stops receiving this message it assumes that it has lost the active session with its peer.

NOTIFICATION This message is used when an error or an exception occurs. After this message is produced, the TCP connection is closed and new TCP and BGP sessions need to start again.

UPDATE This is the message used to withdraw unfeasable routes or to announce new destinations. The latter are contained in the **Network Layer Reachability Information (NLRI)** section of this message.

Each BGP router initially announces to its peers the set of CIDR prefixes owned by the AS where the router belongs. The neighbour ASs, through their own BGP routers, will learn these destinations and may retransmit them to their neighbours depending on their own internal policies.

Besides the NLRI section, the UPDATE message also contains a **Path Attributes** section. These attributes apply to the destinations included in the NLRI section and influence the decision of how peer routers process the announcement:

ORIGIN Defines who originated the path information.

AS_PATH The path of ASs that have transmitted the destination, it is also the list of ASs that will carry the data packets back to the AS that originally announced the destination. This attribute is what allows BGP to avoid creating loops in the paths used to transmit data.

NEXT_HOP The IP address of the router, at the neighbour AS, that will be used as the next hop to reach the destination.

MULTI_EXIT_DISC The **Multiple Exit Discriminator (MED)** is used to control how traffic is distributed between a pair of ASs that share more than one entry point. This parameter allows an AS to define a degree of preference between these links and to pass this preference to its neighbour.

LOCAL_PREF The Local Preference value is assigned by the network administrator to use a preferred route over others. This attribute should only be used by BGP routers at the same AS and should not be passed to BGP routers at other ASs, except when the ASs form a confederation.

ATOMIC_AGGREGATE This attribute indicates that the announced destination has been made less specific by another BGP router. In other words, the destination is a collection of different destinations that have been grouped under the same announcement.

AGGREGATOR The ASN and IP address of the router that performed the route aggregation.

BGP routers store the CIDR prefixes and its attributes in the **Routing Information Base (RIB)**, which is a database that contains:

Adj-RIBs-in All the announcements received from neighbour routers.

Loc-RIB The paths that have been selected to reach a destination.

Adj-RIBs-out The collection of announcements sent to neighbour routers.

Even though there could be more than one available path in the RIB to reach a destination, BGP will only select and propagate its preferred path. This property of the BGP algorithm is defined in this study as:

Definition 2. *The **Preferred Path Rule (PPR)** is the property of the BGP routing protocol that mandates that only one path can be used to reach a destination.*

The preferred path to a destination is selected among other feasable paths according to the strict order described bellow:

1. Select the path with the largest LOCAL_PREF value.
2. Select the path with the shortest AS_PATH.
3. For ASs that have more than one entry point, select the path that has the lowest MED value; ASs with a single entry point should be considered to have the lowest possible value.
4. Select the path with the smallest cost to reach the NEXT_HOP router. This is also called *hot potato routing*.
5. Select the path received through the peer router with the lowest IP address value.

Routes that have been selected using the previous set of rules may be announced to neighbouring ASs depending on the configured export policies. This means that BGP uses a double selection process: one for the routes received that could be adopted, and another for the routes adopted that may be annnounced.

2.2 BGP and the Internet

This section provides some insight on the problems arising from the application of BGP to the Internet. Special attention is given to the literature that describes how instabilities, which originated by normal interactions between BGP routers, start and propagate through the network.

By the end of the 1990s, two distinct lines of research developed which were trying to study BGP instabilities: Gao and Rexford [11] classified them as (1) measurement-based studies of BGP protocol traffic (subsection 2.2.1) and (2) theoretical analysis of BGP convergence properties (subsection 2.2.2). Then, both lines of research are reunited by Gao and Rexford [11] and, starting from this point, most studies provide individual ideas trying to solve the problems defined by the initial studies. This most recent strand of research, starting with [11], is discussed in subsection 2.2.3.

2.2.1 Measurement-based studies of BGP protocol traffic

This line of research starts with two articles: the first one by Paxson [2] in 1996 and the other by Govindan et al. [3]. Although they have the merit of being the earliest studies on network instability highlighting the problems faced in a very young Internet, these studies focused on finding and quantifying the problems experienced by the Internet without analysing the causes that produced these instabilities.

The subsequent article in 1997 by Labovitz et al. [4] was the first comprehensive study of Internet routing instability. In this paper, the authors described one type of instability being *route flapping* which occured when there was a sudden change of network reachability and topology information. They found that, at the time this research was conducted, there was an excessive amount of pathological routing information which did not reflect real topological changes. Some of this pathological information

was originated by a particular implementation of BGP which did not maintain the state on the information advertised to the router's peers. These routers were sending duplicate announcements because they were not keeping a record of routes previously sent to its neighbours. This implementation was called *stateless BGP* and the number of pathological messages decreased significantly after it was modified to send routing information to its peers only when there was a real topology change.

These researchers were also the first ones to describe the **congestion chain reaction (CCR)** pathology suffered by BGP: a congested BGP router may not process a KEEPALIVE message from a neighbour router before the neighbour's timer expires. When this happens, the neighbour router flags the congested route as being down, which may also trigger more congestion in other parts of the network because there are fewer routes available. Even though Labovitz et al. [4] did not propose a solution to this problem, this paper exemplified how inconsistent topological information could originate in the Internet.

Later, Malan and Jahanian [5] used Windmill, a passive network protocol performance measurement tool to demonstrate that it is possible to reduce the CCR pathology by using the UDP protocol to send KEEPALIVE messages. The logic behind this theory is that, in a congested network, a UDP datagram has greater chances of reaching a neighbour router before a TCP datagram, this could allow KEEPALIVE messages to arrive before the timer that withdraws congested routes expires.

In [6], Labovitz et al. continued the study started in [4]. This later article reported that the number of pathological routing information messages in the Internet had significantly diminished after manufacturers eliminated the stateless BGP implementation; this solution was, in fact, the main recommendation the authors had proposed in their previous paper. They also reported that half of the oscillations observed in their experiment stemmed from intra-AS routing protocol vs. BGP configuration errors. This later

finding indicated that configuration errors produced a significant proportion of network inconsistencies in the Internet.

Finally, Labovitz et al. [7] attempted to characterise the type, origin, frequency and duration of Internet's failures. They compared these failures to the ones observed in telephone networks, and concluded that the Internet exhibits significantly less availability and reliability than the telephone network. Although the authors reported the physical source and frequency of regional backbone failures, they did not analyze the original cause of these failures or the effects they produced in the network.

The articles reviewed in this subsection, which sought to measure BGP's performance in a real environment, provide a good snapshot of the state of the Internet at the end of the 1990s. They also help to understand the challenges that BGP needs to meet in today's networks.

2.2.2 Theoretical analysis of BGP convergence properties

This second strand of research was started by an influential technical report by Varadhan et al. [8]. In this report the authors demonstrated that it is possible to create **persistent route oscillations (PRO)** in the network when conflicting BGP policies override the shortest-path logic followed by the original path-vector protocol. The result is that a group of BGP routers cannot converge to a unique solution to reach a destination because of the conflicting policies. This causes the routers to start selecting different routes in an endless cyclic behaviour. Although the researchers admitted this problem had not been found in a real-life environment, they also believed that other factors, like BGP's configuration mechanisms, could increase the probability of this problem to occur in the future.

A proposed solution to the PRO problem was the NSF-sponsored *Routing Arbiter*

project by Govindan et al. [9], which was a routing policy coordination architecture that analyzed BGP relationships between providers. Although this project did not fulfill its promise to become a global coordinator of routing policies, it started the development of the Routing Policy Specification Language (RPSL) and the RIPE Database. Both of these developments have been the basis for the analysis done by this thesis, but further discussion on them needs to be postponed until Chapter 4.

Griffin et al. [10] classified the Routing Arbiter project as a static solution to PRO in BGP. They demonstrated that this kind of solution to BGP's convergence problem is either NP-hard or NP-complete to achieve. The reason for this is due to the exponential number of factors and choices that a BGP system may create to find a stable solution. This paper might have played an important role on why the Routing Arbiter project did not become a global coordinator of routing policies.

2.2.3 Modern studies of BGP convergence properties

The following studies build upon the research described in the previous sections and each one tries to address specific aspects of the BGP protocol operation. The publications described in this section become more involved and complex than the earlier ones and rely on either computer modelling or data analysis.

Gao et al. [11] made a first attempt to create a stable routing mechanism with incomplete topological information by developing guidelines that guarantee route convergence between ASs. Stability is obtained by ensuring that the network follows a hierarchical model, which eliminates some of the potentially problematic alternative path options. Gao et al. [11] argued that most of their proposed guidelines are already regular practice of most ISPs, which effectively implies that many alternative paths are actually restricted by common ISP's policies. The idea proposed by these researchers

may work in a real-life environment, but at the cost of creating an Internet with fewer alternative routes and a predominantly tree-like structure.

Following a different line of research, Labovitz et al. [12] examined the latency in Internet's path failure, failover and repair mechanisms due to the convergence properties in inter-domain routing. The authors performed this study by injecting bogus failure messages and collecting data on how paths changed at different points in the Internet. The results from this study relevant to this thesis are summarised below:

- Routing information associated with replacing a shorter path will usually propagate faster than routing information associated with replacing a longer path. This happens because shorter paths are accepted or rejected immediately, while longer paths have to wait while the router evaluates shorter alternative paths before finally accepting the announced longer path.

- The *MinRouteAdver* timer is usually beneficial for routing convergence. This property was further explored by Griffin et al. [14].

- Convergence time usually diminishes if the sender node checks for loops before sending a routing advertisement to its neighbour. This technique is called *sender-side loop detection* and was also further discussed by Griffin et al. [14].

The authors' conclusion is a clear warning on the state of BGP and the tasks that may be needed to improve its performance:

> "We can certainly improve BGP convergence through the addition of synchronization, diffusing updates and additional state information, but all of these changes to BGP come at the expense of a more complex protocol and increase router overhead" [12].

Labovitz et al. [13] extended the previous study by demonstrating that the period of time that multi-homed ASs use to propagate a path failure scales linearly with the length of the longest possible backup path for that route. In other words, the larger the network, the longer it is going to take for it to converge. The solution they proposed in order to optimise the convergence time is to minimise the longest paths in the network. However, decreasing the length of possible paths will negatively impact the connectivity of the network, because some longer routes will be ignored.

Complementing the studies by Labovitz et al. [12, 13], Griffin et al. [14] used a network simulator to analyze the influence that the following three BGP parameters have on convergence time:

- The MinRouteAdver timer: This timer limits the rate at which a router announces new routes to its neighbours.
- Sender-side loop detection: Mechanism that checks for routing loops before sending a routing message to a peer, instead of checking when a message is received.
- Withdrawal rate limiting: This timer, similar to MinRouteAdver, is used for sending withdrawal messages to peer routers.

Griffin et al. [14] discovered that distinct topologies experience different behaviours depending on the values used for these parameters, and that the MinRouteAdver timer is the most significant of the three for network convergence time optimisation. According to their simulations, an optimal MinRouteAdver timer value is highly beneficial in networks with large number of alternative paths, while setting a smaller timer than the optimal value quickly becomes harmful for network stability.

These researchers found in this same study that withdrawal rate limiting can result

in either a gain or a loss on network convergence time. They also found that sender-side loop detection consistently reduces convergence time when paths are withdrawn, although the benefits are small. On the other hand, sender-side loop detection has little effect on convergence time when paths are reinstated.

Griffin et al. did not specify in [14] how to obtain the optimal MinRouteAdver value for reducing the network convergence time, but Brian Premore's doctoral thesis [26] revealed more details about the results published in the article: the optimal MinRouteAdver timer value was found by trial and error, and it may not be possible to obtain this value using calculations because it changes significantly with the network topology.

An influential article by Mao et al. [15] demonstrated that the BGP's route flap damping mechanism [27] can increase the convergence time of many networks by triggering the route suppression mechanism. The researchers explained the root cause of this problem:

> "The intuition for this [increased convergence time] comes from the work of Labovitz et al. [12], who showed that a single route withdrawal can result in other routers exploring a sequence of alternate paths before deciding that the destination is unreachable. In this paper, we show that this kind of exploration causes what we call secondary flaps that can trigger the suppression threshold of the route flap damping algorithm" [15].

The experimental results obtained by Mao et al. [15] revealed that the smallest clique network in which withdrawal messages could trigger route suppression is a clique of size five. Other types of topologies, including pyramid-like and Internet-based, were also tested. Their withdrawal messages produced different types of failures and needed different minimal number of nodes to trigger suppressions. These results

demonstrated why simulations involving just a few nodes (3 or 4) cannot provide a true idea of how a new implementation will behave on a real network.

Another type of inconsistent routing information in the network, **multiple origin AS (MOAS)**, was described by Zhao et al. [16]. MOAS happens when two different ASs announce the same CIDR prefix, which may cause information intended for an AS to be erroneously routed to a different AS. This type of failure, also called *IP prefix hijack*, is common in the Internet; unfortunately in [16] the authors did not provide a solution to this problem.

A recent article by Chin [28] elaborated on the scenarios that cause MOAS and found that most of them are intentional and valid. Chin also found that unintentional MOAS conflicts are usually due to misconfigurations or spammers. This latter type of instability has a root cause which falls outside the scope of this study; however it is still important to consider the consequences of two different ASs unintentionally announcing the same destination prefix in the Internet.

2.3 Examples and analysis

Although Varadhan et al. [8] and Govindan et al. [9] could not provide any real-life examples of PRO, it is now accepted that this phenomenon develops when a BGP's MED parameter creates a conflict when selecting the preferred path between ASs [29, 30]. This problem was further analysed in a paper by Griffin and Wilfong [31], where the authors concluded that it is difficult to avoid creating the scenarios that develop these path oscillations; however, they offered some clues on how to debug and solve this instability.

A fine example of how instabilities may propagate and cause harm in the Internet is an outage which happened in April 25, 1997 [32]: On that day, AS7007 propagated

announcements received from a downstream ISP which caused other ASs to believe that AS7007 possesed the best routes to many important destinations in the Internet. This event caused major Internet backbone failures for a couple of hours and, because the routes were propagated to ASs around the globe, it continued to cause problems in other sections of the Internet for several more hours.

It is reasonable to believe that CCR, PRO and the other stability issues that have been mentioned in this chapter are of real concern and could cause disruptions in the Internet. The following chapter analyses the attempts that other researchers have done to prevent these network instabilities.

Chapter 3

New Ideas for the Same Internet

The network instabilities introduced in the previous chapter have motivated many researchers to develop different tools and methods to solve BGP's stability and resiliency problems. Individual BGP enhancements that have been previously proposed are analysed in section 3.1, while alternative routing protocols, which could replace BGP, are discussed in section 3.2. Finally, section 3.3 asserts that, before any solution to the Internet's instabilities could be proposed and implemented, first it is necessary to understand the topological structure that this network possesses, because it is likely that the strategies needed to solve this problem depend on this network's topology.

3.1 New ideas for BGP

Two different lines of research to solve BGP's instability problems have evolved. The first one has focused its efforts in solving BGP's PRO problem (subsection 3.1.1), while the second one has used the results derived from the Internet's messaging conflicts, analysed by Labovitz et al. [12, 13] and Mao et al. [15], to pursue techniques that would allow faster BGP convergence to topological changes (subsection 3.1.2).

3.1.1 Solving persistent route oscillations in BGP

In 1999 Griffin et al. [33–35] introduced the simple path vector protocol (SPVP) as a tool to "capture the underlying semantics of any path-vector protocol such as BGP". In these papers the authors asserted that, in order to prevent PROs, it is best if BGP attempts to solve the *stable paths problem* instead of the *shortest path problem*, which is the main objective of most other routing protocols.

Griffin et al. [33–35] also developed a tool called the *dispute wheel*, which is a mathematical representation of the long range interactions that develop when nodes in a BGP system impose their policies over the network. In [35] they proved that if there is no dispute wheel present, SPVP is guaranteed to converge to a single solution. In order to implement this tool they proposed to use a new optional attribute, called *history*, at the moment of deciding how to route records. However, there is a price to pay for using the history attribute: it also exports attributes to other ASs.

Cobb and Musunuri [36] provided in 2004 a different solution to the PRO problem by using a cost metric that increases when BGP cannot reach a stable solution. BGP's cyclic behaviour is broken when an AS, whose metric has increased beyond a set threshold, picks an *escape path*. The authors named their solution the Bounded Divergence Protocol (BDP).

Finally, Ee et al. [37] provided in 2007 a method based on Griffin's dispute wheel to avoid route oscillations. Their solution uses a *precedence metric* to find if the network has developed a cyclic behaviour. The authors claimed that this solution provides greater flexibility than Griffin's history attribute and that it could also solve the instabilities originated by the use of the MED BGP paramater. Unfortunately, their claims require strengthening through further testing and analysis before this new method could be accepted as a definitive solution to the PRO problem.

The SPVP proposed by Griffin et al. [33–35] and the BDP proposed by Cobb and Musunuri [36], may successfully solve BGP's PRO problem, but these solutions do not support a simple method to determine which AS or group of ASs originated this instability. They also could not determine how to find and avoid the development of dispute wheels when the instabilities are caused by the MED parameter. Perhaps the solution proposed by Ee et al. [37] could help to solve BGP's cyclic behaviour, but further testing and analysis of this mechanism are needed before a definitive veredict can be drawn.

3.1.2 In pursuit of faster BGP convergence

In [38], Pei et al. developed *consistency assertions* for BGP that compare similar routes and identify the *a priori* unfeasible ones. By using this technique, the authors expected to reduce the network convergence time and eliminate unnecessary route changes. One of the most insightful ideas introduced by this paper is that, although withdrawal messages always eliminate routes to a destination, there is an operational difference between eliminating a route because of a network failure or because of a BGP policy: the withdrawal of a route because of a network failure reflects a real topological change that should be learned and used by the affected nodes; on the other hand, a route withdrawal because of policy does not reflect a real topological change and different backup routes should be used to reach the same destination. This article demonstrated how BGP can sometimes contradict itself and create unstable states, e.g.: some route withdrawals created by policy could be invalidated by alternative routes that enter the same path through a different router.

Bremler-Barr et al. [39] found that inconsistent routing information could float around the network for long periods of time and called this type of false information

ghosts. The authors claimed that this is a variant of the *count to infinity* problem that naturally develops in distance-vector protocols, and concluded that faster convergence could be achieved by speeding withdrawal messages of unfeasable paths or, as they called it, *flushing* the ghosts.

Zhang et al. [40] studied the relationship between BGP's convergence delay and the effect this has on packet delivery in the network. They found that some fast convergence solutions may cause more packets to be dropped because the path-withdrawal message travels faster than the reinstate message, and this is specially problematic when the failure is transient. They concluded that the relationship between route convergence and packet delivery is more complex than anticipated and advocated the need to take this into consideration when proposing solutions to this problem. They also asserted that during routing convergence periods it is possible that transient loops might occur.

In a more recent paper, Pei et al. [41] employed a new kind of parameter, called the root cause notification (RCN), to help BGP improve its convergence time. The RCN parameter is formed of the identification number of the node that found the path failure and a sequence number which increments each time an event occurs. The authors compared the performance of RCN against consistency assertions [38] and ghost flushing [39], and concluded that in most cases their solution works best.

A similar solution to RCN is the enhanced path-vector routing protocol (EPIC) developed by Chandrashekar et al. [42]. In this article the authors claim that BGP's delayed convergence is caused by the excesive message exchange between routers which develops while the network tries to converge after a failure; they called this transient state, *path exploration*. Then, the authors proposed to use Forward Edge Sequence Numbers (FESN) that would help to identify the specific link that fails and also the order of the failures. This last parameter enables the identification of duplicated or delayed messages that should be ignored by the router. Since the FESN parameter would

be specific per link failure it will allow identification of the paths that are affected by the routing message received.

The use of the FESN [42] parameter creates an *absolute order in time* of the messages that are produced in a network. The importance of this mechanism lies in that this could be considered as the first attempt to create a **complete temporal order** in the routing messages exchanged in the Internet. The combination of this parameter with the **complete topological order** which is proposed in the following chapters may become a powerful scheme to guarantee stability in the Internet.

Finally, Lijun et al. [43] proposed to modify BGP's route flap damping mechanism to obtain faster path convergence. They introduced two new parameters that allow to identify paths that need to be ignored by BGP's damping mechanism: *neighbouring nodes supression* and *invalid routes damping*, and called this solution modified route flap damping (MRFD). The advantage of MRFD lies in its simplicity, but it may lack the robustness that EPIC or RCN provide.

Neither of the faster BGP convergence parameters discussed in this subsection can guarantee that PRO will not develop in the Internet, which means that these parameters could only be effective as long as the network can reach a stable state. Nobody has actually tried to study the effects of these techniques in a system that is oscillating.

3.2 Alternative routing protocols

In comparison to the literature on enhancements for BGP, the studies on alternative inter-AS routing protocols is relatively scarce. This subsection reviews only two of them.

Subramanian et al. [44] proposed a new inter-AS routing protocol that could substitute BGP: HLP or hybrid link-state and path-vector protocol. The main objective

behind this new protocol is to keep a balance between hierarchical information and policy privacy by mixing two different kinds of routing protocols under the same routing framework. The authors also wanted to fix BGP's main deficiencies, which according to them are: poor scalability (the Internet is growing faster than BGP's capabilities), route instability, route oscillations, long convergence times and poor fault isolation properties. HLP uses link-state mechanisms for routing within hierarchies, usually specified by provider-to-customer relationships, and a path-vector architecture for routing between hierarchies, or peer-to-peer relationships. According to the authors the benefits of using this structure are:

> "The link-state component improves convergence and reduces churn within a hierarchy, while the path-vector component preserves global scalability by hiding internal route updates across hierarchies (thereby sacrificing global visibility)" [44].

The authors acknowledgeed that HLP may never replace BGP, but they offered this protocol as a first step to find a better EGP.

Another routing protocol, proposed by Xu and Rexford [45], is the multi-path interdomain routing protocol (MIRO), which is a significant modification to BGP that would allow distant ASs to negotiate and use paths with lower preference. Although MIRO is similar to other ideas proposed earlier (source routing and overlay networks), its routing strategies could allow an increase of path diversity in the Internet. On the other hand, the authors also warned that new convergence problems could emerge because MIRO overrides the hierarchical guidelines introduced by Gao et al. [11].

None of the solutions described in section 3.1 addressed the reduced path diversity that BGP experiences because of its Preferred Path Rule (section 2.1); MIRO may lend an opportunity to increase the Internet's path diversity, although its functionality could

also create new instabilities. Conversely, HLP provides a stable routing solution, but it heavily relies on a rigid commercial structure, and the authors did not show how to address other complex relations that may exist between ASs: it is possible that, because of a link failure, an AS that used to be a client to another AS could become a provider.

3.3 The Internet's topology

The proposals described in sections 3.1 and 3.2 have focused on how to improve the current routing framework without addressing the topological characteristics of the network in which the routing protocol needs to function. This thesis advocates rethinking the routing issues in topological terms and find how well connected the Internet really is, how efficient is BGP at exploiting this connectivity, and what could be done to reduce the gap between these two measures without sacrificing the stability of this network.

By analysing the topological structure of the Internet, it may be possible to understand the connectivity that this network has, and to measure the inherent resilience that this connectivity provides. But in order to measure the resilience of the Internet, first it is necessary to provided a definition for this term:

Definition 3. *The **resilience** of a network is the capacity to maintain its connectivity and communication after a failure has occurred.*

It is also important to make a distinction with two other closely related terms, safety and robustness, which apply to BGP and were introduced by Jaggard and Ramachandran [46]:

Definition 4. *A routing solution is **safe** if cyclic behaviours (PRO) do not develop.*

Definition 5. *The **robustness** of a network is the ability to find a new safe routing solution after a failure has occurred.*

Hence, resilience could be measured as the connectivity a network experiences and how well it manages to stay connected when the topology changes, while robustness is the ability to find a solution that quickly converges to this new topology. But in order to find a suitable metric for the Internet's resilience, first it is necessary to characterise the topological structure of this network.

The first attempt to describe the Internet's topology comes from Faloutsos et al. [17], who discovered that *the Internet follows a power-law distribution in the degree of its nodes* (ASs). A **power-law** is a mathematical expression of the form:

$$x \propto y^a$$

where x and y are variables and a is a defined constant. The **degree** of a node (or vertex) is the number of edges incident to the node and was represented by y in the equation. Then, the authors calculated the frequency of each degree, represented by x, which is just the number of nodes with a specific degree, and plotted the frequency versus the degree in a log-log scale. For three different datasets obtained at different times, the authors found almost identical values of $a = -2.2$. The significance of these findings is that the distribution of the Internet at the AS level cannot be described by a random graph, because it follows a distribution where ASs with low degree are more frequent.

The findings by Faloutsos et al. [17] were complemented by an extensive review article by Albert and Barabási [18], in which they analysed different complex networks, including the Internet. In this article they defined the networks which follow a power-law degree distribution as **scale-free networks**, because power law distributions are

free of a characteristic scale. The authors also explained that scale-free networks in real environments are usually the result of two types of mechanisms: growth and preferential attachment. Evidently, the Internet is a network that possesses both characteristics. They also compared the resilience of scale-free networks versus random networks against node failures, and concluded that, in general, scale-free topologies are more resilient but are also more vulnerable when the most highly connected nodes are removed. Based on the analysis presented in this article [18], the authors asserted that scale-free networks are "exceptionally resilient against random node failures".

Tangmunarunkit et al. [19] studied two different types of topology generator tools: *degree-based* topology generator tools which try to mimic the Internet's power-law distribution, and *structural* generators which imitate the Internet's hierarchical structure. Contrary to the authors' expectations, they discovered that degree-based generators produced better Internet-like topologies than structural generators. Another finding made by the authors of this article is that the network's resilience decreased when policy routing was taken into account; unfortunately, the policies used to obtain these results were based on simple rules which do not reflect how policies are applied in reality by most ASs.

In a more recent paper by Mahadevan et al. [20] there is an analysis of the Internet topologies generated by three different projects which generate different types of data sources:

- Skitter [47] is a project which stores the paths used to transmit packets in the Internet. This routing information is limited in the sense that paths that are not currently being used will not be included in its data.

- Route Views [48] stores BGP routing tables and updates obtained from collectors at eight different locations. Although the routing information may be considered

complete for the BGP routers that are monitored at each collector, the announcements received at these routers have already been filtered by their peers according to BGP's PPR.

- The RIPE Database [49] is a collection of the different routing policies that have been provided by various European ASs. Since the contents of this database are compiled manually and on a voluntary basis, it is not possible to confirm the accuracy of its routing information, specially since the Internet never stops evolving.

Furthermore, Skitter and Route Views produce global topologies, while the RIPE Database only stores routing information for Europe. On the other hand, from these three data sources, RIPE Database provides greater number of links between ASs and also greater information on how policies may influence the final routing decisions. Mahadevan et al. [20] found that the topology produced by the RIPE Database does not follow a power-law distribution, this is due to the large number of links this source provides.

The previous studies demonstrated that the Internet has a structure that can be modelled mathematically, but recently Oliveira et al. [21] demonstrated that previous attempts to find a valid Internet topology have failed, and that the connectivity found by these studies is lower than the one really experienced in the Internet. Unfortunately, Oliveira et al. [21] did not provide a solution to the problems they found with past topological models. Even though they promise to address this issue in a future paper, it was not yet published at the time of writing this thesis, hence the impossibility to use their results for this research project. Since the following chapter intends to define a resilience metric for the Internet based on its connectivity, there is a possibility that future findings by these authors could have positive implications for the analysis and

proposals introduced later in this thesis.

The following chapter employs the ideas and topological studies described in this chapter to analyse how the Internet's connectivity is affected by BGP and the application of its policies, and to devise an experiment which would allow to measure the connectivity and resilience of a section of the Internet.

Chapter 4

Measuring the Internet's Resilience

This chapter starts by analysing how destination address advertisements originate and propagate between ASs and, by considering BGP's functionality and its policies, how information packets could flow back to a destination (section 4.1). Although these findings are useful, they still cannot provide a metric of the Internet's resilience, therefore section 4.2 proposes the use of connectivity definitions from graph theory to measure this characteristic. In order to apply this newly proposed metric, a numerical experiment to measure the resilience of the European core of the Internet is devised and described in section 4.3. Then, section 4.4 describes each one of the routing information sources used in the numerical analysis: The RIPE Database, RIPE RIS, Route Views and Skitter. Section 4.5 provides details on the software developed to process the collected data. Finally, in section 4.6 there is an analysis of the results obtained and a discussion of their significance for the overall resilience of the Internet.

4.1 BGP policies create a digraph

Communication networks, like the Internet, are usually modelled as a **graph** $G = (V, E)$ in which the vertices V represent routers or ASs and the edges E describe the communication links between them. Figure 4.1 is an example of such a graph; this is an undirected graph because the communication links in a network are usually bidirectional.

In the specific case of BGP, routing information to a destination (or a group of destinations) is transmitted from an AS to its neighbour ASs through the existing communication links. Because policy is an essential characteristic of any EGP, some ASs may choose not to communicate some destinations to certain ASs. Still many routes may be able to propagate through the Internet. It then becomes necessary to represent the propagation of this routing information as a **directed graph** or **digraph** $D = (V, A)$ which is just an orientation of G:

Definition 6. *The graph that represents how an AS, i, propagates its destination advertisements through a network is called the* ***announcement digraph of AS i,*** *$D_{annc}(i)$. The announcement digraph is only restricted by the policies that some ASs impose over their neighbours for transmitting the reachability of destination i.*

Figure 4.2 is an example of an announcement digraph for AS 12 ($D_{annc}(12)$). This digraph is not real in the sense that it does not factor BGP's PPR, which modifies how routing information is actually transmitted between ASs. The announcement digraph may be considered as a union of all the permited alternatives to transmit a destination through the network before BGP's functionality is imposed.

As each AS selects its preferred route to reach a destination (or group of destinations), many of the redundant routes in the announcement digraph will be ignored in order to comply with BGP's PPR. Each AS will choose which routes to export (or an-

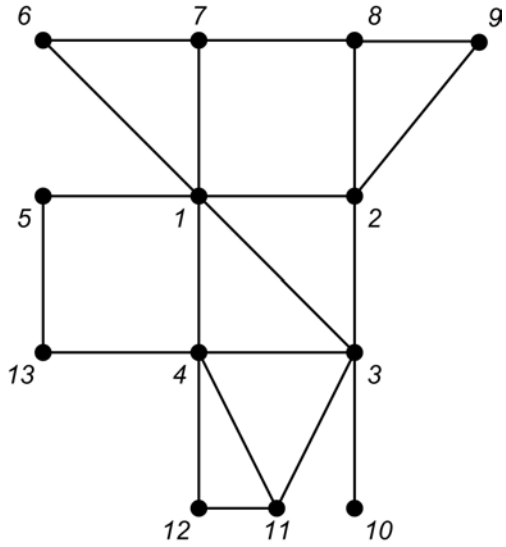


Figure 4.1: Communication network

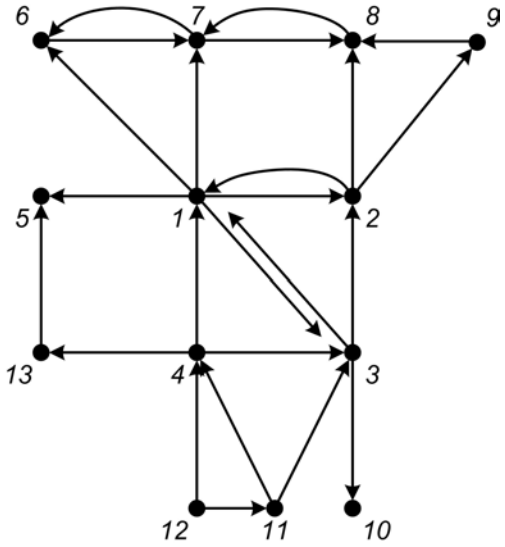


Figure 4.2: Announcement digraph for AS 12, $D_{annc}\,(12)$

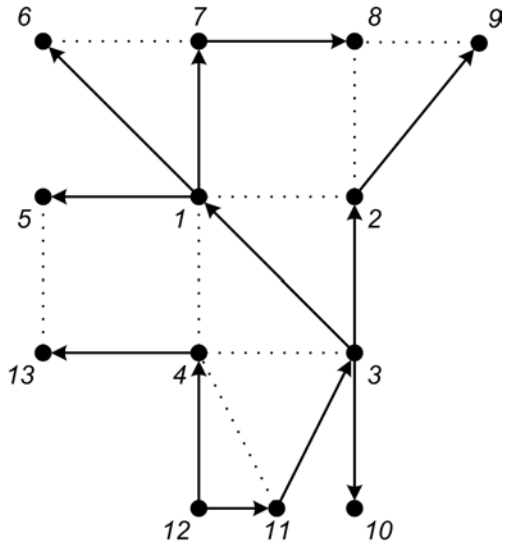


Figure 4.3: BGP digraph for AS 12, $D_{BGP}\,(12)$

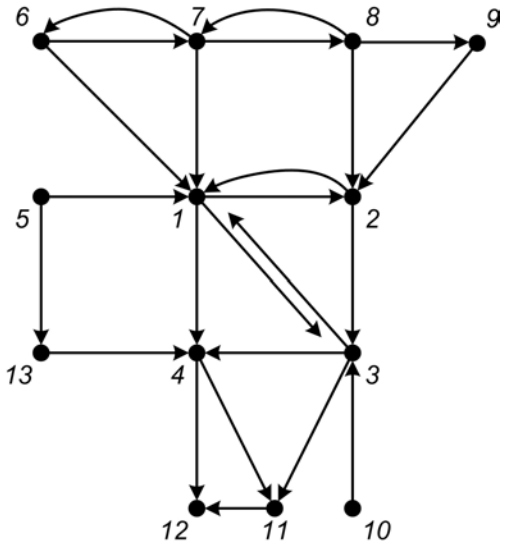


Figure 4.4: Destination digraph for AS 12, $D_{dest}\,(12)$

nounce) and which routes to import (or accept) depending on the policies defined by the network administrators. Figure 4.3 is an example of how BGP functionality modifies the announcement digraph for a destination and creates rigid paths which form an **oriented tree** or **arborescence**. In this thesis, this graph is called the **BGP digraph of AS i,** $D_{BGP}\,(i)$. This digraph is equivalent to what other authors have called a **solution** to a BGP system [10, 33, 34], because when this arborescence is defined all the ASs in the network converge to the same solution.

Because the BGP digraph only provides one path between each AS and the destination AS i, when a path is lost the ASs that cannot reach the destination need to find an alternative path by consulting their own routing tables and selecting a new preferred path. This period in which the ASs are processing and building a new solution is what causes the transient BGP instabilities that were described in section 3.1.2.

In a digraph, the operation of **reversing** an arc means to replace an arc ab with an arc ba that has the opposite direction. When we reverse all the arcs in a digraph we obtain the **converse** of the same digraph. In this study, the converse of $D_{annc}\,(i)$ represents the paths that other ASs may use to reach the destination that was originally advertised. This will be called the **destination digraph of AS i,** $D_{dest}\,(i)$. Figure 4.4 is an example of the destination digraph obtained from Figure 4.2.

By comparing the original network topology (Figure 4.1) to the announcement digraph and BGP digraph (Figures 4.2 and 4.3), it is evident that the original network has more alternative paths than the announcement digraph and, subsequently, they both have more options than the final BGP digraph; but still there is no clear way to quantify this diversity. In the following section a metric to define diversity and resilience in the Internet is proposed, which is based on the digraphs that have just been introduced.

4.2 The connectivity of a digraph

In order to find the connectivity of a digraph, first it is necessary to define some terms. The following mathematical definitions come from the book by Bang-Jensen and Gutin [50]:

Definition 7. *A digraph D is **strongly connected** or **strong** if for every vertex of D it is possible to reach any other vertex in D.*

Definition 8. *A **strong component** of D is a maximal induced subdigraph of D which is still strong.*

According to theorem 7.2.2 in [50], for a digraph to be strong it is necessary that there is at least one cycle in this digraph. These concepts of strongness are useful to this study because every AS in the Internet should be able to send IP packets to all other ASs in this same network.

Definition 9. *A group of **arc-disjoint (s,t)-paths** is a set of paths connecting vertex s to vertex t through intermediate vertices, in which none of the paths traverse the same arc more than once.*

Definition 10. *A group of **internally disjoint (s,t)-paths** is a set of paths connecting vertex s to vertex t where neither the arcs nor the vertices are common in any of these paths.*

As an example, the digraph depicted in Figure 4.5 has a maximum of 2 arc-disjoint (s,t)-paths (examples are indicated with patterned lines), but only 1 possible internally disjoint (s,t)-path, because any path from s to t needs to pass through vertex x. Meanwhile, the graph depicted in Figure 4.6 has a maximum of 3 arc-disjoint (s,t)-paths (examples are also indicated with patterned lines), and 2 internally disjoint (s,t)-paths, which must pass through vertices y and z.

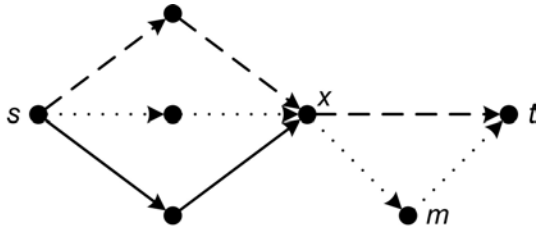


Figure 4.5: 2 arc-disjoint (s,t)-paths

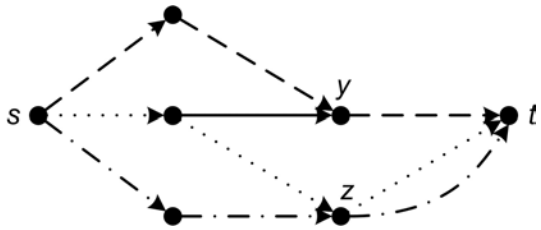


Figure 4.6: 2 internally disjoint (s,t)-paths

Sections 1.5 and 7.3 in [50] define the global connectivity and strongness of a digraph using the following two alternative concepts:

Definition 11. *The **arc-strong connectivity** of a digraph is the minimum number of arc-disjoint (s,t)-paths for every pair (s and t) of vertices in the digraph.*

If a digraph has an arc-strong connectivity of 3, it means that any vertex in this graph has at least 3 different arc-disjoint paths to every other vertex.

Definition 12. *The **vertex-strong connectivity** is the minimum number of internally disjoint (s,t)-paths for every pair (s and t) of vertices in the digraph.*

When a digraph has a vertex-strong connectivity of 2, it means that any vertex in this graph has at least 2 internally disjoint paths to every other vertex.

Although both previous definitions measure a digraph's strongness, vertex-strong connectivity is a measure of how resilient a digraph is when vertices (and their adjacent arcs) are eliminated from the digraph, while arc-strong connectivity defines how

resilient a digraph is when only arcs are eliminated from the digraph. When these two concepts are translated in terms of an inter-AS network, vertex-strong connectivity measures the resilience of a network when a complete AS fails, while arc-strong connectivity only measures the resiliency when the communication links between ASs fail. But which one of these two definitions is more suitable to quantify connectivity between ASs in the Internet?

A digraph with a certain value of vertex-strong connectivity will be more resilient than another digraph that shares the same value for arc-strong connectivity [51]. Unfortunately, this higher level of connectivity can only be achieved, for vertex-strong connectivity, by introducing new arcs between existing vertices which, in the context of a network, is equivalent to adding new communication links between ASs. Conversely, arc-strong connectivity could be increased by using parallel arcs which, also for a network environment, may represent a higher preference of an existing link or the introduction of a backup link. Hence, increasing arc-strong connectivity does not imply that the network topology needs to be modified, while this is the case when increasing vertex-strong connectivity.

It is also necessary to consider that ASs, represented by vertices in these digraphs, are usually under the administration of an ISP, which usually employs a team of engineers to monitor and assure that their network is always operational. Therefore the disappearance of a vertex, or the failure of an entire AS, is a very unlikely event, significantly less probable than a link failure event.

Given the improbability that an AS fails and disappears from the topology and that increasing the vertex-strong connectivity in a network is not a trivial task, it is plausible that the arc-strong connectivity is a more suitable metric for the connectivity of an announcement digraph.

Still, it seems that other alternatives may deserve further analysis. For example, it

is possible to count the number of different paths that an AS may use to reach other ASs in the network. Although such a metric would be valid, it does not really reflect the number of useful alternative paths between two different ASs. As an example, Figure 4.5 shows a network with two arc-disjoint paths between s and t, but which has six different paths that could be used to reach vertex t. However, when any of the two pair of links xt, xm or xt, mt fail, the connectivity disappears. The fact that this digraph could be disconnected by removing at least two arcs is reflected by the arc-strong connectivity value (2), while the other metric (6) may lead to the false perception that other alternative paths could exist. In conclusion, arc-strong connectivity provides a better approach to how resilient a network is to link failure.

Having established that arc-strong connectivity is the most apropriate metric to define the path diversity available in a network, and that by increasing this diversity, an increase in the resilience of the network could also be obtained. The next step would be to find the arc-strong connectivity of the Internet before and after BGP's restrictions have been applied to the network. If the difference between these two metrics is significant it demonstrates that there is an opportunity to improve the path diversity and the resilience of this network.

The connectivity of the Internet after BGP has applied the PPR can be determined by looking at Figure 4.3. In this digraph it is possible to see that there is only one arc-disjoint path between AS 12 and any other ASs in this network. This explains the necessity of BGP to process and announce an alternative route every time the topology changes.

On the other hand the announcement digraph in Figure 4.2 has different arc-disjoint values for different ASs. For example, there are two arc-disjoint paths between ASs 12 and 4, but only one between ASs 12 and 10. In the Internet, ASs that only have one or two links to their own ISPs do not have many choices when routing to different

destinations, but ASs that are closer to the network core will have greater path diversity that could be exploited to increase the resilience.

In order to find if ASs closer to the Internet core really have greater path diversity, a numerical experiment, which builds announcement digraphs for the core ASs of the European Internet, was devised. Such an experiment and its results are the main topic of the remainder of this chapter.

4.3 The core of the European Internet

In order to obtain announcement digraphs that have an arc-strong connectivity greater than one and which also result in increased path diversity, it was necessary to restrict this study to ASs that possess good connectivity. This is why ASs closer to the core were selected; ASs at the edge of the Internet tend not to have enough path diversity, with just a few exceptions [21]. Also, because the RIPE Database only provides routing data for Europe, this numerical analysis also had to be restricted to include only European ASs. Therefore, it was decided that the 50 largest European ASs should provide enough connectivity to produce announcement digraphs with good path diversity.

The first step to obtain the European core's connnectivity, was to define the ASs that form this core. CAIDA [52] maintains an AS ranking project [53] which was used to build a list of the 50 largest European ASs as of May 24, 2006. This initial list is shown in Tables 4.1 and 4.2:

#	AS	Organisation
1	1299	TeliaNet Global Network, Sweden
2	3320	Deutsche Telekom AG (DTAG), Germany
3	702	MCI EMEA, Netherlands
4	3303	Swisscom Enterprise Solutions Ltd, Switzerland
5	1257	TELE2 Group, Sweden

Table 4.1: Top 50 European ASs (Part 1)

#	AS	Organisation
6	13237	European Backbone of LambdaNet, Germany
7	8220	COLT Telecommunications, Netherlands
8	286	KPN Internet Backbone AS, Netherlands
9	3257	Tiscali Intl Network, Italy
10	1273	Cable & Wireless, UK
11	16150	Port80 AB, Sweden
12	8928	Interoute Communications Ltd, UK
13	8342	RTComm.RU AS, Russia
14	5413	PIPEX, UK
15	5511	France Telecom
16	12956	Telefonica Backbone AS, Spain
17	6762	Telecom Italia Sparkle
18	15412	Flag Telecom Global Internet AS, UK
19	5400	BT European Backbone, UK
20	3300	Infonet-Europe, UK
21	20965	The GEANT IP Service, UK
22	3292	TDC Data Networks, Denmark
23	2856	BTnet UK Regional network
24	3246	Song Networks, Finland
25	6805	Telefonica Deutschland AS, Germany
26	8210	Nextbone - Telenor Networks' international backbone, Norway
27	3301	TeliaNet Sweden
28	8434	Telenor AB, Sweden
29	5462	Telewest Broadband, UK
30	6878	T-Systems International GmbH, Germany
31	31399	DaimlerChrysler Autonomous System, Germany
32	6667	EUnet Finland Backbone AS / Saunalahti Group, Finland
33	1103	SURFnet, Netherlands
34	2603	NORDUnet, Norway
35	680	DFN-IP service G-WiN, Germany
36	2200	Renater, France
37	3269	TELECOM ITALIA
38	3215	France Telecom Transpac
39	9121	TTnet Autonomous System, Netherlands
40	786	The JANET IP Service, UK
41	8404	Cablecom GmbH, Netherlands
42	20485	JSC Company TransTeleCom, Netherlands
43	9035	Wind Telecomunicazioni spa, Italy
44	1267	Infostrada S.p.A., Italy
45	3352	Internet Access Network of TDE, Spain
46	6830	UPC Distribution Services, Europe
47	3216	Golden Telecom, Moscow, Russia
48	5568	RBNet, Russia
49	20562	Open Peering Initiative, Amsterdam, Netherlands
50	25462	ReTN.net Autonomous System, Netherlands

Table 4.2: Top 50 European ASs (Part 2)

4.3.1 Incomplete routing data

As the analysis of the routing information obtained from these ASs progressed, some were discovered to provide unusable information; hence they were excluded from the list. The reasons why each one of these ASs were deleted from the list are provided below:

AS2200: Renater (Réseau National de télécommunications pour la Technolgie l'Enseignement et la Recherche) is a network that connects many research facilities in France. Unfortunately this network exports just a few destinations to its neighbours, which are very few in any case.

AS2856: BTnet is part of British Telecom's (BT) main backbone. According to Robtex this AS is used to route within the UK [54]. While verifying connectivity for different ASs it was found that BTnet usually appears disconnected from the rest of Europe, probably because it obtains most of its routing information directly from BT's AS.

AS3300: Infonet was acquired by BT in early 2005 to enhance BT's prescence in North America and Asia, and to offer an increased variety of telecommunication services [55, 56]. According to the RIPE Database, Infonet's Network Management Center (NMC) is located in Belgium and the company also claims that they provide global services, but in most routing data analysed their AS was disconnected from the rest of Europe.

AS3320: Deutsche Telekom AG The network administrators of the second largest AS in Europe decided that it was in their best interest not to provide any routing information to the RIPE Database. Although this AS may be important in obtaining

a complete picture of the European topology, it is not possible to infer their destinations and links without their routing information.

AS20562: Open Peering Initiative is a small ISP which provides low-cost interconnectivity to Dutch companies. Its main objective is to bypass larger service providers to provide cheaper services within Holland [57]. Besides its poor connectivity with the rest of Europe, this AS is the second smallest of the original list.

AS25462: RETN is a Russian ISP [58] and is also the smallest AS of the list; it also suffers from poor interconnectivity with the rest of Europe.

Besides these six deleted ASs, there were incongruencies in the data that AS5462 (Telewest Broadband) was providing, so this AS was substituted. In addition, since Deutsche Telekom did not provide routing information, it was necessary to complement the list by using a global ISP which exceptionally provided its routing information in the RIPE Database. Both cases are further discussed in the following two subsections.

4.3.2 AS substitution

Telewest Broadband (AS5462) provided Internet and cable services in parts of Great Britain until it was acquired in 2005 by NTL (AS5089), which was a direct competitor providing similar services [59]. In 2006 NTL-Telewest acquired Virgin Mobile and changed its commercial name to Virgin Media which now offers Internet, phone, cable and mobile services in the UK [60].

By experimenting with both ASs, it was found that AS5462 (Telewest) had overtaken the routing functionality provided by AS5089 (NTL), so it was decided to replace the former by the latter in the list.

4.3.3 A global ISP

By studying the routing information of some disconnected ASs, it was found that many European ISPs connect to the rest of the world through Level 3 Communications (AS3356), which is an American ISP which provides global Internet services. Fortunately, they provided their routing information to the RIPE Database.

To verify the influence that this AS has on the rest of Europe, AS3356 was added to the list and it was observed that the connectivity of the other ASs in the list increased. This lead to the conclusion that, in order to mitigate the connectivity probems caused by deleting Deutsche Telekom, it was necessary to include AS3356 into the final list.

Processing the routing information for Level 3 was not a straight-forward process: additional characters were found in its routing destination data. These were codes to specify that they do not accept prefixes beyond a defined length. A custom script, Preprocess_AS3356.pl, had to be created to remove the additonal characters from AS3356's routing information.

4.3.4 The final European list

After performing the modifications explained in subsections 4.3.1, 4.3.2 and 4.3.3, the final list includes only the forty-five (45) European ASs, shown in Table 4.3.

Henceforth, this list will be refered as the **Top-45**. Routing data was collected from different sources for each AS in this list. The data was harvested in two different days to avoid the possibility of taking samples on a non-typical day. The dates selected randomly were November 29 and December 19, 2007. The following section (4.4) describes the nature of the Top-45's routing data and how it was used to produce announcement digraphs.

#	AS	Organisation
1	1299	TeliaNet Global Network, Sweden
2	702	MCI EMEA, Netherlands
3	3303	Swisscom Enterprise Solutions Ltd, Switzerland
4	1257	TELE2 Group, Sweden
5	13237	European Backbone of LambdaNet, Germany
6	8220	COLT Telecommunications, Netherlands
7	286	KPN Internet Backbone AS, Netherlands
8	3257	Tiscali Intl Network, Italy
9	1273	Cable & Wireless, UK
10	16150	Port80 AB, Sweden
11	8928	Interoute Communications Ltd, UK
12	8342	RTComm.RU AS, Russia
13	5413	PIPEX, UK
14	5511	France Telecom
15	12956	Telefonica Backbone AS, Spain
16	6762	Telecom Italia Sparkle
17	15412	Flag Telecom Global Internet AS, UK
18	5400	BT European Backbone, UK
19	20965	The GEANT IP Service, UK
20	3292	TDC Data Networks, Denmark
21	3246	Song Networks, Finland
22	6805	Telefonica Deutschland AS, Germany
23	8210	Nextbone - Telenor Networks' international backbone, Norway
24	3301	TeliaNet Sweden
25	8434	Telenor AB, Sweden
26	5089	NTL Group Limited, UK
27	6878	T-Systems International GmbH, Germany
28	31399	DaimlerChrysler Autonomous System, Germany
29	6667	EUnet Finland Backbone AS / Saunalahti Group, Finland
30	1103	SURFnet, Netherlands
31	2603	NORDUnet, Norway
32	680	DFN-IP service G-WiN, Germany
33	3269	TELECOM ITALIA
34	3215	France Telecom Transpac
35	9121	TTnet Autonomous System, Netherlands
36	786	The JANET IP Service, UK
37	8404	Cablecom GmbH, Netherlands
38	20485	JSC Company TransTeleCom, Netherlands
39	9035	Wind Telecomunicazioni spa, Italy
40	1267	Infostrada S.p.A., Italy
41	3352	Internet Access Network of TDE, Spain
42	6830	UPC Distribution Services, Europe
43	3216	Golden Telecom, Moscow, Russia
44	5568	RBNet, Russia
45	3356	Level 3 Communications, U.S.

Table 4.3: Top-45 European ASs

4.4 Data Sources

Four different data sources are described in this section: The RIPE Database, RIPE RIS, Route Views and Skitter. But only three different data types were processed because RIPE RIS and Route Views produce similar types of routing information.

4.4.1 The RIPE Database

The RIPE Database [49] and RIPE RIS (subsection 4.4.2) are projects mantained by RIPE NCC [61]. This organisation is one of the five Regional Internet Registries (RIR) which provide registration services and resource allocation for the whole Internet. Specifically, RIPE NCC services all ASs located in Europe, the Middle East, Central Asia and Russia.

The RIPE Database stores IP addresses, AS numbers, organisations associated with these resources, their contact details and their routing information. This information is available in RPSL [62] format, which is a standard language created to represent routing policies and instructions.

Almost every organisation registered at RIPE NCC provides their routing information and policies through a series of export and import rules using RPSL. A very simple example of an export rule is:

```
export: to AS786 announce AS-KPN
```

The AS generating the previous rule announces a group of ASs, called AS-KPN, to its neighbour AS786. Conversely, AS786 may contain a matching import rule similar to:

```
import: from AS1299 accept AS-KPN
```

This means that AS 786 will accept the AS-KPN list of ASs from its neighbour AS1299. The list of ASs announced between neighbours (AS-KPN) allows the determination of the destinations available from one AS through another. Henceforth, these lists will be called using the generic name **AS-List**.

The RIPE Database allows users to specify their own AS-Lists. Some ASs will restrict the number of destinations they pass or export to a specific neighbour by defining a smaller AS-List; likewise, some ASs may restrict the number of destinations they will accept by using a smaller AS-List when defining their import rule.

By analysing the list of ASs which are passed from one AS to the next, it is possible to create the announcement digraph that originates at a particular AS and propagates from a neighbor AS to the next one until the announcement of the original AS reaches all the other ASs in the Top-45 list.

To perform this analysis it was necessary to produce a set of Perl scripts that would read the routing information for each AS in the Top-45 list, compare their export and import rules, and then create a set of announcement digraphs which represent how announcements may propagate to all the other ASs.

Because the RIPE Database's data is updated manually, it is common to find incomplete and old information and there are even claims that some users modify their information on purpose [20, 21]. Hence the Perl scripts were designed to verify that the connectivity claimed by each AS matched the information provided by its neigbours. If a route could not be verified by the scripts, it was assumed to be false.

Details of the Perl scripts developed, as well as their functionality are provided in section 4.5, and an analysis of the results obtained from this data is presented in section 4.6.

4.4.1.1 Handling of inaccurate and incomplete information

The main problem in processing routing data from the RIPE Database was that some ASs do not provide accurate routing information. This is due to the fact that the information stored in this database is created and loaded manually. The following list explains the specific problems found and the corresponding solutions adopted:

AS1299: TeliaNet was the upstream provider for AS20485 (TransTeleCom) according to TransTeleCom's routing information, but AS1299 did not include itself in the group of ASs announced to AS20485. This configuration problem was probably originated by TeliaNet, but since there is no way to confirm this hypotesis, it was best to leave these two ASs disconnected.

Another inconsistency between TeliaNet and AS8342 (RTComm) originated because this second AS did not explicitly said that it was connected to AS1299, although the opposite was true. Since there is no way of verifying which configuration was correct, both ASs were kept disconnected.

AS3303: Swisscom decided to condense its export routes to its peers by using the following abstraction as its only export rule:

```
export to AS-SWCMPEERS announce AS-SWCMGLOBAL
```

A custom script (Preprocess_AS3303.pl) was created to translate Swisscom's condensed export rule into a group of detailed export rules that could be processed by the Perl scripts developed.

AS8210: Nextbone had an announcement digraph which seemed to be almost completely disconnected from the rest of Europe. Closer inspection revealed that this AS was exporting the AS-TELENOR list, and this list does not contain AS8210.

Since this is not a reasonable setup, as it means that Nextbone announces to its neighbours all the routes it knows except for destinations contained within Nextbone's AS, it was decided that the best solution to this problem was to manually add:

1. AS8210 to each neighbour's export rules in AS8210's data file.
2. AS8210 to the import rules of each neighbour's data file.

AS8220: COLT Telecommunications had routing inconsistencies with four different ASs. The following list describes each problem found with COLT's routing data, and a summary of the changes made in order to fix these problems is provided in Table 4.4.

- AS3215 (France Telecom domestic IP backbone) imported the AS-List AS-COLTFR, but this did not include AS8220. Since there was no way to confirm why this inconsistency was happening or who was causing it, it was decided to keep these two ASs disconnected.
- COLT imported routing data from AS3356 (Level 3 Communications) via the AS-LEVEL3 list, which was not recognised as a valid AS-List by the RIPE Database. Since it is impossible to deduce the original intention of this announcement, it was decided to allow COLT to import only AS3356. This decision seems to contradict the original reason for including Level 3 in the Top-45 list, but given the lack of any other type of routing information, it was concluded that using the smallest AS-List that could be transferred between these ASs was the fairest possible solution.
- AS3352 (TDE: Telefonica Data España) accepted the AS-TDECOLT list which was an invalid AS-List.

- AS6805 (Telefonica Deutschland) accepted the invalid list AS-CO.

Although there seems to be no particular reason why the two last AS-Lists are not defined in the RIPE database, it is possible to conclude that the purpose of them was to allow AS8220 to pass routing information to Telefonica's ASs. Hence it was decided to replace AS-TDECOLT and AS-CO with AS-COLT on the import rules for these two ASs.

AS	Original AS-List	Modified AS-List
AS8220	AS-LEVEL3	AS3356
AS3352	AS-TDECOLT	AS-COLT
AS6805	AS-CO	AS-COLT

Table 4.4: Modifications related to COLT

AS9035: Wind Telecomunicazioni had a configuration problem similar to AS8210 (Nextbone): AS9035 seemed to be almost completely disconnected from the rest of Europe, but its exporting list (AS-INFOSTRADA) did not contain AS9035. In order to solve this problem it was necessary to manually add AS9035 to the import rules of each neighbour's data file.

AS20965: GEANT had particular instructions in the RIPE Database to create, use and announce a special list of ASs called community attributes. The advantage of using these commands is that it is possible to define routing attributes in a more flexible and transparent form. The disadvantage is that it creates routing instructions which are more complex to read and decode by the scripts developed. Hence, a tailored set of procedures were added to the scripts in order to decode the community attributes used by GEANT.

4.4.1.2 Additional Inconsistencies

Smaller inconsistencies, mostly typos and old labels, were probably caused because the RIPE Database is updated manually. These are listed in Table 4.5. The columns in this table provide: the AS where the inconsistency was found, the original AS-list that caused the problem and the new label used.

AS	Original AS-List	Modified AS-List
AS286	AS12956:AS-TDATANETEU	AS-TDATANETEU
AS702	AS-1273 AS-TELIA AS-TELEDK	AS1273 AS-TELIANET AS-TDCNET
AS786	RS-UK RS-UK-CUSTOMER AS-RTCOMM-LINX-EXP AS-TELEDK AS5413 AS-PIPEX	AS702:RS-UK AS702:RS-UK-CUSTOMER AS8342:AS-RTCOMM-LINX-EXP AS-TDCNET AS5413:AS-PIPEX
AS1103	AS-TELEDK	AS-TDCNET
AS1257	AS-TELEDK	AS-TDCNET
AS3257	AS-NTL	AS-NTLI
AS3292	AS-LNCETNOD	AS-LNCNETNOD
AS3303	AS-TELEDK	AS-TDCNET
AS5089	AS-TISCALI-BACKBONE AS-PORT80-GLOBALTRANSIT AS-RTCOMM-AS	AS-TISCALI AS-CUSTOMERS AS8342:AS-RTCOMM-LINX-EXP
AS6805	AS-XARANE AS-CABLEINE AS-20562 AS-TELEDK AS-NTL	AS-XARANET AS-CABLEINET AS20562 AS-TDCNET AS-NTLI
AS6830	AS-NTL AS-JANET	AS-NTLI AS-JANETPLUS
AS8434	AS-TELEDK	AS-TDCNET
AS8928	AS-FLAG AS-RTCOMM-LINX-EXP AS-TELEDK	AS-FLAGP AS8342:AS-RTCOMM-LINX-EXP AS-TDCNET
AS12956	AS-RTCOMM-LINX	AS8342:AS-RTCOMM-LINX-EXP
AS13237	AS-TELEDK	AS-TDCNET
AS15412	AS-BTTOLLINX AS-JANET AS-PIPEX	AS-BTTOLINX AS-JANETPLUS AS5413:AS-PIPEX

Table 4.5: Additional inconsistencies fixed

4.4.1.3 Unresolved Inconsistencies

There were inconsistencies in the RIPE Database that could not be resolved, because after analysing the routing data, there was no clear solution to the problems they were causing. Table 4.6 provides a list of these inconsistencies and a probable solution, which in the end was not used:

AS	Original AS-List	Probable Fix
AS286	AS-KPNEURO	AS-KPN
AS702	AS-SONGNETWORKS	Delete
AS786	AS-AUCSPEERS AS-TELENORDIASE	Delete AS-TDCNET
AS2603	AS-SONERA	Delete
AS6830	RS-UUNETAT AS-LNCPARIX	 Delete
AS8928	AS-LNCBNIX AS-LNCSFIN	Delete Delete

Table 4.6: Unsolved inconsistencies

4.4.2 RIPE RIS

The RIPE Routing Information Service (RIS) [63] is a RIPE NCC project that collects and stores BGP's routing data. RIS has Remote Route Collectors (RRCs) at fifteen Internet Exchanges. These RRCs peer with local operators to collect their routing information. In total, over 600 RRCs peer sessions are tracked. Table 4.7 provides a list of the RRCs from which routing data was collected, as well as their physical locations.

RIPE RIS provides two different sets of routing information:

- RD: This is a snapshot of the current BGP routing table.
- AD: This is a record of the BGP updates received.

Since all the destinations should be available in the BGP routing table, it was concluded that the RD data should be enough to produce the desired Internet topologies.

RRC	Name	Location
rrc00	RIPE NCC	Amsterdam
rrc01	LINX	London
rrc02	SFINX	Paris
rrc03	AMS-IX, NL-IX, GN-IX	Amsterdam
rrc04	CIXP	Geneva
rrc05	VIX	Vienna
rrc06	NSPIXP2	Otemachi
rrc07	Netnod	Stockholm
rrc10	MIX	Milan
rrc11	NYIIX	New York
rrc12	DE-CIX	Frankfurt
rrc13	MSK-IX	Moscow
rrc14	PAIX	Palo Alto
rrc15	PTTMetro-SP	Brazil

Table 4.7: RIPE RIS Remote Route Collectors

Data collected from the RRCs is stored in Multi-threaded Routing Toolkit (MRT) format, so it had to be translated to a text format which could be easily read and parsed by Perl scripts. Fortunately RIPE NCC provided a program called libbgpdump [64], which converts MRT data to text format. The resulting text file contains a series of routing entries similar to the following example:

```
TIME: 05/09/03 04:01:59
TYPE: BGP4MP/MESSAGE/Update
FROM: 198.58.5.254 AS3727
TO: 198.58.5.34 AS3936
ORIGIN: IGP
ASPATH: 3727 2914 6730 8640
NEXT_HOP: 198.58.5.254
ATOMIC_AGGREGATE
AGGREGATOR: AS8640 195.141.213.58
COMMUNITY: 2914:420 2914:2000 2914:3000 3727:380
ANNOUNCE
  195.28.224.0/19
```

The ASPATH parameter describes an ASs path that may be used to transmit data from this AS (parameter TO) to a destination AS (parameter FROM). This means that it is possible to deduce how routing messages travel through the Internet by using the ASs

in ASPATH in reverse order. A series of Perl scripts were created to process these paths and produce a set of announcement digraphs for the Top-45 ASs. Further discussion on these scripts is provided in section 4.5

The MRT routing information produced by RIPE RIS has identical format to the data produced by the Route Views project (section 4.4.3), therefore their data was combined and processed simultaneously using the same scripts. Henceforth, the combined information of these two projects will be regarded as **BGP Routing Table (BRT)** data.

There are two reasons why BRT data cannot produce a complete Internet topology:

1. Every AS knows routes that may not be propagated to the next AS because of its own set policies. In other words, many paths will not be propagated because of BGP's PPR rule.

2. Paths stored at each RRC are specific to them, which means there are routes that will never be recorded at the RRCs because they are not intended to reach any of the routers included in these projects

In conclusion, although BRT data provides valid propagation paths between ASs, there will be many other paths that are not detected by the RRCs. Hence BRT data could be used to complement the information provided by the RIPE Database, but it cannot be the sole source of routing information.

4.4.3 Route Views

The Route Views project [48] is supported by the University of Oregon Advanced Network Technology Center [65]. Just like RIPE RIS, it collects and stores BGP's routing data from several RRCs at nine Internet hosts, but only eight data archives stored full BGP routing tables. Table 4.8 provides a list of the data archives from which routing data was obtained:

Host
route-views2.oregon-ix.net
route-views.eqix.routeviews.org (Equinix Ashburn)
route-views.eqix.routeviews.org (Equinix)
route-views.isc.routeviews.org
route-views.kixp.routeviews.org
route-views.linx.routeviews.org
route-views.wide.routeviews.org
route-views6.oregon-ix.net

Table 4.8: Route Views data archives used

Like RIPE RIS, Route Views provides two different sets of routing information in MRT format:

- RIBs: This is a snapshot of the current BGP routing table. This is analogous to RIPE RIS's RD data sets.
- UPDATES: This is a record of the BGP updates received. This is analogous to RIPE RIS's AD data sets.

Just as with RIPE RIS's data, the analysis was limited to the RIB data and the RIPE NCC's libbgpdump program [64] was used to convert from MRT to text format. The converted data was processed simultaneously with RIPE RIS's data and further details on the scripts used are provided in section 4.5.

The results produced by the BRT data will be presented and discussed in the final section of this chapter (4.6).

4.4.4 Skittter

The Skitter project [47] is supported by CAIDA [52]. This tool uses Internet Control Message Protocol (ICMP) messages to build a topology of the Internet according to the ASs that are visited when this message is propagated.

Skitter is able to register data flows from one AS directly onto the next one, but it cannot reveal the policies that apply between these ASs, where the data was originated, nor how it will reach its final destination. This means that this tool is able to create a topology that represents only how the Internet is passing data; it does not enable definition of paths that might be used to reach distant ASs. Therefore the announcement digraphs produced using these data only contain links from the source AS to a list of immediate neighbours.

Similar to BRT data, Skitter will ignore sections of the Internet where it cannot collect data. Therefore, its view of the Internet's topology is also incomplete. The script used to process the Skitter data is discussed in section 4.5, and the results are also analysed in section 4.6.

4.5 Processing the collected data

Several Perl scripts were developed to parse and process each one of the different data described in the previous section (4.4). The information of each data file needed to be converted into a format that could be easily analysed and displayed, so adjacency matrices were selected.

The **adjacency matrix** of a digraph is a square matrix of dimensions $V \times V$ (the **order** or number of vertices in the digraph) in which each column (row) of the matrix represents the head (tail) of an arc. Each element of this matrix is either 1, if the arc that corresponds to the row-column pair is present in the digraph, or 0 otherwise.

Three adjacency matrices, which represent the announcement digraph of each data type, were created for each Top-45 AS. The connectivity of these digraphs increased according to the order: Skitter, BRT data and RIPE Database, and although many elements were common in all three matrices, many others did not.

Data type	File name	File type
RIPE Database	get_as_data.sh	UNIX script
	Preprocess_AS3303.pl	AS specific script
	Preprocess_AS3356.pl	AS specific script
	DGPS.pl	main Perl script
	DG_CLASS.pm	Perl class module
	IP_CLASS.pm	Perl class module
BRT data	libbgpdump.c	C program (modified)
	AMG2.pl	main Perl script
	RD_CLASS.pm	Perl class module
Skitter	skitter.pl	main Perl script
combined	JOINMAT.pl	Perl script
	ORMAT.pl	Perl script
	WHOSWHO.pl	Perl script

Table 4.9: Scripts to process topologies

Since these matrices represent different views from the same network, it was concluded that in order to obtain the closest aproximation to the Internet's announcement digraphs, which would apply if BGP does not impose the PPR, it was necessary to combine their information by performing a union of the directed edges from all three data sources.

The scripts used to process each data source type, and also to combine their information and produce meaningful results are listed in Table 4.9, and their code has been included in Appendix B. A brief explanation of each script's functionality is provided below:

RIPE Database This data was downloaded from the whois.ripe.net server using the UNIX script get_as_data.sh. Because of the issues described in subsections 4.3.3 and 4.4.1.1, the data files for AS3303 and AS3356 needed to be modified (preprocessed) by the Perl scripts Preprocess_AS3303.pl and Preprocess_AS3356.pl. Then all the data files were processed simultaneously by the DGPS.pl Perl script, which generates a data structure of destination groups and produces announcement matrices as output.

BRT This data was downloaded from the Internet pages of the project that generated them (RIPE RIS and Route Views). The libbgpdump [64] program was modified to produce a list of paths formed only of the Top-45 ASs, which are the ones that can be used to build the required announcement digraphs. Then, this data was transformed into adjacency matrices using the AMG2.pl Perl script.

Skitter The Perl script provided by CAIDA, asadj2graph.pl [66], was modified to produce simple ajdacency matrices under the name skitter.pl.

combined The adjacency matrices from the three previous data types were combined, by performing logical OR operations of all their edges, using the JOINMAT.pl Perl script. Two other additional Perl scripts were developed to perform a simple logical OR operation between any two adjacency files (ORMAT.pl), and to create an adjacency file that shows which data source originated and also which edges are common in the three input files (WHOSWHO.pl).

4.6 Analysis and results

The 45 announcement digraphs $D_{annc}(i)$ obtained by using the scripts described in the previous section represent how destination advertisements could propagate from a **source AS**, labeled i, to the other 44 ASs in the European Internet core. Most of them were connected digraphs where AS i is the origin. Many contained cycles and, in some, it was even possible to return to i. Just 9 digraphs had some ASs that could not be connected from i. We presume that these ASs should be obtaining announcements through ASs which have not been included in the Top 45 list, like Deutsche Telekom. Figure 4.7 shows AS8220's announcement digraph for data collected on December 19, 2007. The announcement digraphs for the Top-45 in both collection dates has been

included in Appendix A.

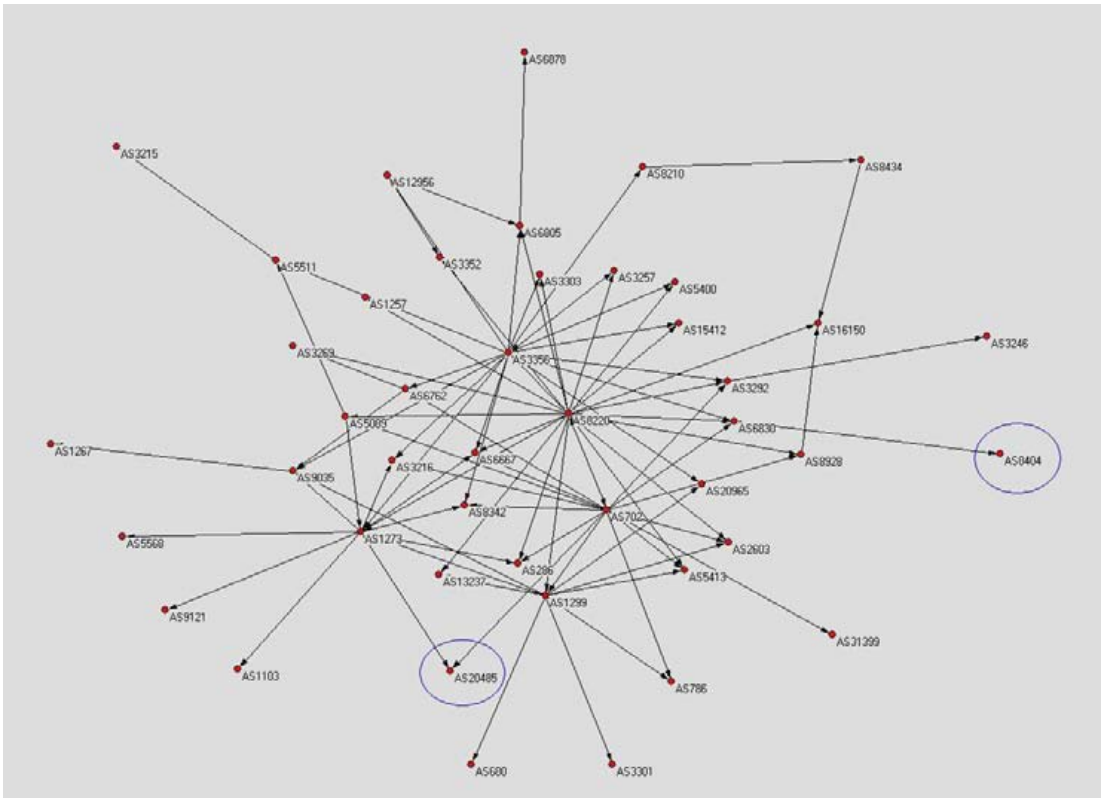


Figure 4.7: Announcement digraph for AS8220 (December 19 ,2007)

As in the other announcement digraphs obtained (Appendix A), the one shown in Figure 4.7 has two types of vertices or ASs:

- Vertices which receive announcements only through a single directed arc; these vertices are called **stubs**, e.g.: AS8404.
- Vertices that are connected through more than a single arc; also called the **core** of the digraph, e.g.: AS20485.

Stub vertices do not really have options at the time of choosing a path back to the origin because they only receive a single announcement through a neighbour AS. On the other hand, core vertices have more than one path back to the origin, hence only the core's connectivity properties may produce results that represent path diversity.

The arc-strong connectivity of these core digraphs was obtained by implementing a series of Matlab scripts. The main script, called AMA.m, obtains the arc-strong connectivity of each strong component (Definition 8, section 4.2) of a digraph using the following two steps: First, find all the strong components of the digraph using the algorithm developed by Tarjan and elaborated in [67]; then calculate the arc-strong connectivity of each component using the following proposition and the preflow-push algorithm by Goldberg and Tarjan (section 3.6.3 in [50]):

Proposition 1 (from [50], p. 356)**.** *For any directed multigraph $D = (V, A)$ with $V = \{v_1, v_2, \ldots, v_n\}$ the arc-strong connectivity of D, $\lambda(D)$, satisfies*

$$\lambda(D) = min\left\{\lambda(v_1, v_2), \ldots, \lambda(v_{n-1}, v_n), \lambda(v_n, v_1)\right\}$$

where $\lambda(s, t)$ is the maximum number of arc-disjoint (s,t)-paths in D.

The Matlab script used to find the stong components, also called strong blocks, of a digraph is called strong_components.m; the one that calculates the arc-strong connectivity is named Fnd_Arc_Strg_Conn.m and preflow_push.m implements the preflow-push algorithm. These Matlab scripts have been included in Appendix B.

The analysis provided by these algorithms revealed that each digraph has several strong blocks, but usually only one of these components per graph had an arc-strong connectivity greater than zero. Notice that by definition a vertex that is not strongly connected to any other vertex is a strong block of its own. Therefore, most of the components discovered by Tarjan's algorithm are just single vertices.

Table 4.10 shows how many strong components each digraph had and the arc-strong connectivity (Definition 11, section 4.2) obtained for the best connected component in each AS. From this table it is possible to see that most announcement digraphs have a 1 arc-strong core. This is a strong component of vertices or ASs which can propagate

the announcements sent by the originating AS through at least one arc-disjoint route amongst themselves. Some ASs' announcement digraphs have 2 arc-strong cores, so in this case it is possible to create an even stronger block with 2 arc-disjoint routes. On the other hand, there are a few cases in which the arc-strong connectivity is zero for the whole announcement digraph. This does not mean that the digraph is disconnected, just that there are no cycles present in the digraph that would allow forming strong components (theorem 7.2.2 in [50]).

AS	strg. comp.	arc-strong	AS	strg. comp.	arc-strong
AS1299	12	1	AS3301	12	1
AS702	11	1	AS8434	9	1
AS3303	13	1	AS5089	9	2
AS1257	13	0	AS6878	11	1
AS13237	12	1	AS31399	10	1
AS8220	15	1	AS6667	11	1
AS286	12	1	AS1103	11	1
AS3257	13	1	AS2603	11	1
AS1273	12	1	AS680	12	1
AS16150	9	1	AS3269	13	1
AS8928	10	1	AS3215	14	1
AS8342	12	1	AS9121	8	1
AS5413	11	1	AS786	10	1
AS5511	13	0	AS8404	12	1
AS12956	9	1	AS20485	10	2
AS6762	12	0	AS9035	8	1
AS15412	11	1	AS1267	7	1
AS5400	14	1	AS3352	13	1
AS20965	12	1	AS6830	9	1
AS3292	11	2	AS3216	10	2
AS3246	11	1	AS5568	10	1
AS6805	11	2	AS3356	14	0
AS8210	9	1			

Table 4.10: Arc-strong connectivity analysis of the Top 45 ASs

By using the converse of each announcement digraph $D_{annc}(i)$ to obtain the corresponding destination digraph $D_{dest}(i)$ (section 4.1), it is possible to calculate how many arc-disjoint paths each AS could use to reach i. The result of this analysis for each of the Top-45 ASs for November 29, 2007, is shown in Tables 4.11 and 4.12 at

the end of this chapter.

The last column in table 4.12 shows that on average each AS knows about 2 different arc-disjoint paths to any other AS. The maximum number of arc-disjoint paths between any two ASs obtained is 6, and the minimum is 0. Most probably, these isolated ASs are connected through ASs which were not included in the Top-45 list, such as Deutsche Telekom.

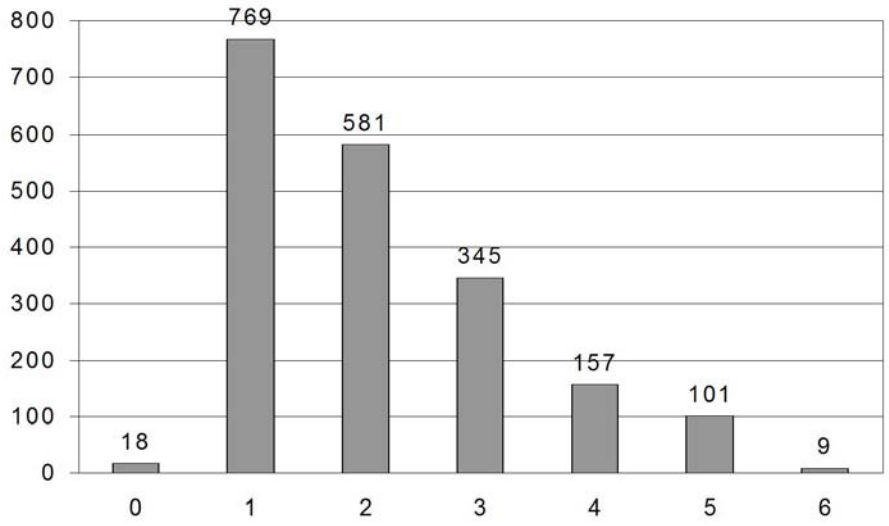


Figure 4.8: Number of arc-disjoint paths between any two ASs for the Top-45

The frequency of the number of arc-disjoint paths in Tables 4.11 and 4.12 are displayed in Figure 4.8. This Figure shows that in most cases (almost $2/3$) there is more than one arc-disjoint path between any two ASs, although the most frequent situation is to have only one path between ASs. In total, there are 1193 occurrences of more than one arc-disjoint paths. This is proof that there is a large number of extra paths that could be used for backup or to shape the flow of information in the network, if a protocol capable of exploiting these paths were to be employed. Hence, the most important result obtained from this numerical experiment is the fact that there is great connectivity that is not being exploited by BGP, even after the application of local policies is

considered.

It is important to remark that the data on which the results of this chapter are based, are not entirely accurate. Still the results obtained by this numerical analysis concur with Oliveira et al. [21], which demonstrated that the Internet has greater connectivity than has been previously reported. Also, many of the inconsistencies discovered in section 4.4.1 were solved by using the least connected possible options. Therefore, it is probable that the inaccuracies in this experiment's data, if corrected, could increase the path diversity of this network. Finally, lack of routing data for ASs outside Europe meant that many other highly connected ASs could not be included in this numerical analysis. By including these ASs, it is expected that a larger and better connected network would be obtained compared to what was observed in this numerical experiment.

The numerical experiment described and analysed in this chapter demonstrates that, at its core, the Internet possesses a richness of path diversity, even after policies have been applied to the connectivity of this network. Since section 4.1 demonstrated how BGP only uses one preferred path to reach each destination, it cannot take advantage of this path diversity. The following two chapters presents a proposal on how to exploit this connectivity in order to increase the resilience and the robustness of the Internet. These new ideas are still firmly based on graph theory.

Orig AS	AS1299	AS702	AS3303	AS1257	AS13237	AS8220	AS286	AS3257	AS1273	AS16150	AS8928	AS8342	AS5413	AS5511	AS12956	AS6762	AS15412	AS5400	AS20965	AS3292	AS3246	AS6805	AS8210	AS3301
AS1299		1	3	1	1	1	3	1	2	1	2	3	2	1	2	1	1	1	2	3	1	2	1	1
AS702	2		2	2	4	4	3	3	2	6	4	3	5	2	3	3	2	3	2	3	4	3	2	1
AS3303	2	1		3	3	5	4	3	2	4	3	5	4	3	3	4	2	4	2	5	4	4	2	1
AS1257	3	1	2		2	1	3	1	2	1	2	3	3	1	2	1	2	1	2	3	2	2	2	2
AS13237	2	2	4	3		3	3	3	3	3	2	4	3	1	3	2	2	2	2	4	2	2	1	1
AS8220	2	1	2	1	2		2	1	3	3	2	3	4	2	1	2	2	2	2	3	2	3	1	1
AS286	3	1	2	1	2	3		1	1	3	2	2	3	0	2	2	2	1	2	3	2	3	2	1
AS3257	2	2	4	3	3	3	4		2	2	2	4	3	2	3	2	3	2	2	4	2	3	2	1
AS1273	2	3	3	2	2	3	3	2		2	2	3	3	2	3	2	2	3	2	4	2	2	1	1
AS16150	2	3	4	2	3	4	4	2	2		2	4	3	3	3	2	4	4	2	4	5	3	3	3
AS8928	4	4	3	3	2	2	3	3	3	2		4	3	3	3	3	2	4	2	4	2	3	1	1
AS8342	3	2	3	1	2	2	2	2	2	3	3		4	3	3	2	2	2	2	4	3	2	2	1
AS5413	2	2	3	3	2	2	3	2	3	2	2	4		1	2	2	2	3	2	3	1	3	1	1
AS5511	3	1	2	1	1	1	2	1	2	2	3	3	2		2	1	1	1	2	2	2	2	1	1
AS12956	3	3	4	3	1	2	4	3	2	1	2	3	2	3		3	1	4	2	5	1	2	1	1
AS6762	2	1	2	1	1	0	3	1	2	1	2	3	2	1	2		2	1	2	3	1	2	1	1
AS15412	2	2	3	3	2	3	3	3	2	2	2	3	3	2	2	3		3	2	3	3	3	2	1
AS5400	2	1	2	1	1	2	2	1	2	3	2	3	3	1	2	1	2		2	3	1	2	2	1
AS20965	1	2	2	1	1	1	2	2	2	1	2	2	2	2	2	2	1	2		2	1	2	1	1
AS3292	2	2	4	3	3	3	4	3	2	4	3	5	3	1	3	2	3	2	2		2	4	2	2
AS3246	1	2	3	3	3	2	3	2	2	3	2	3	3	1	2	2	3	2	2	2		3	2	2
AS6805	2	2	3	1	2	1	3	2	2	2	2	2	3	2	2	2	2	2	2	3	2		2	1
AS8210	0	1	2	1	1	0	1	1	1	2	1	1	2	0	1	0	2	1	1	2	2	2		0
AS3301	1	1	1	2	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	2	1	1	
AS8434	2	2	3	1	2	1	2	2	1	2	2	2	2	2	2	2	2	2	2	3	4	2	2	2
AS5089	6	6	5	3	3	4	5	6	4	4	5	4	5	5	6	5	2	4	2	5	2	3	2	1
AS6878	2	2	2	2	2	2	2	2	2	2	2	2	2	1	2	1	2	2	2	2	2	2	2	1
AS31399	3	3	3	3	2	3	3	3	3	3	3	3	3	3	3	3	3	3	2	3	2	3	3	1
AS6667	2	3	3	2	2	2	3	2	2	2	2	3	3	3	3	3	2	1	2	4	3	3	3	1
AS1103	2	1	3	2	2	1	3	2	2	2	2	3	3	1	2	1	2	2	3	4	2	3	2	1
AS2603	3	2	3	2	2	1	3	2	2	2	2	3	2	1	2	2	1	1	3	4	2	2	1	2
AS680	3	2	3	1	2	4	3	2	2	2	2	3	2	1	2	2	1	2	3	4	1	3	1	1
AS3269	2	2	2	1	1	1	2	1	1	1	2	2	2	1	1	1	1	1	2	2	1	1	1	1
AS3215	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
AS9121	5	5	5	5	3	5	5	5	5	4	4	5	5	5	5	5	5	5	2	5	2	4	3	1
AS786	4	2	3	3	2	2	4	4	4	2	2	4	3	2	3	2	2	3	3	4	2	2	2	1
AS8404	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
AS20485	4	5	5	5	6	5	5	4	5	5	4	5	5	4	5	4	4	4	2	5	6	4	5	1
AS9035	4	4	4	3	1	1	4	4	4	1	2	3	3	4	4	3	1	3	2	4	1	2	1	1
AS1267	2	2	3	2	2	2	3	3	2	1	3	2	2	2	2	2	1	2	2	2	1	2	1	1
AS3352	1	1	1	1	1	2	1	1	1	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1
AS6830	5	3	4	4	2	4	4	4	4	3	2	3	3	5	5	4	2	4	2	5	1	3	2	1
AS3216	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	4	4	2	5	5	4	5	1
AS5568	4	5	5	5	6	5	6	4	5	5	4	4	5	4	5	3	3	4	3	5	5	3	5	2
AS3356	2	1	1	0	2	2	3	1	1	3	3	3	3	1	1	1	1	1	2	2	2	2	1	1

Table 4.11: Paths from an originating AS to other ASs for November 29, 2007 (Part 1)

Orig AS	AS8434	AS5089	AS6878	AS31399	AS6667	AS1103	AS2603	AS680	AS3269	AS3215	AS9121	AS786	AS8404	AS20485	AS9035	AS1267	AS3352	AS6830	AS3216	AS5568	AS3356	Average
AS1299	1	1	1	1	3	1	2	1	1	1	2	1	1	3	4	1	1	2	4	1	1	1.61
AS702	3	3	1	1	4	3	1	3	2	1	1	4	1	3	4	1	1	3	3	1	2	2.59
AS3303	3	3	1	1	5	3	1	3	2	1	1	4	1	2	4	1	1	4	3	1	4	2.77
AS1257	1	0	1	1	4	2	2	1	1	1	1	2	1	2	4	2	1	2	3	1	1	1.77
AS13237	2	2	1	1	4	2	2	2	1	1	1	2	1	4	3	1	1	3	3	1	2	2.20
AS8220	1	2	2	1	4	1	3	1	2	2	1	2	1	2	4	1	2	3	3	1	2	2.00
AS286	1	2	1	1	3	2	1	2	1	0	1	3	1	2	3	1	1	3	2	1	1	1.75
AS3257	1	3	1	1	5	2	1	3	2	1	1	1	1	2	4	2	2	3	3	1	1	2.30
AS1273	2	4	1	1	3	1	1	1	1	1	1	2	1	2	4	1	1	3	3	1	1	2.05
AS16150	3	3	1	1	4	2	3	2	1	1	1	3	1	3	3	1	1	4	3	1	1	2.59
AS8928	2	2	1	1	4	1	2	2	2	1	1	1	1	2	4	1	2	4	3	1	2	2.36
AS8342	3	2	1	1	3	1	1	1	1	1	1	3	1	3	3	1	1	2	4	1	2	2.09
AS5413	1	3	1	1	4	1	1	1	1	1	1	3	1	2	3	1	1	3	3	1	1	1.95
AS5511	2	2	1	1	4	1	2	1	1	1	1	2	1	3	4	1	1	2	3	1	1	1.70
AS12956	2	3	1	1	3	1	2	1	1	1	1	2	1	2	4	1	1	3	3	1	2	2.11
AS6762	1	0	1	1	3	1	2	1	1	1	2	1	1	2	4	1	1	2	3	1	1	1.55
AS15412	3	2	1	1	3	1	1	1	1	1	1	2	1	3	3	1	1	3	2	1	2	2.09
AS5400	2	1	1	1	3	1	1	2	1	1	1	3	1	3	4	1	2	2	3	1	1	1.77
AS20965	2	1	1	1	2	1	1	2	1	1	1	2	1	2	2	1	1	2	2	1	2	1.52
AS3292	4	2	1	1	4	2	2	2	1	1	1	3	1	4	4	1	1	2	3	1	2	2.43
AS3246	3	2	1	1	4	2	2	2	1	1	1	3	1	3	2	1	1	2	3	1	2	2.09
AS6805	1	3	1	1	3	1	1	2	2	1	1	2	1	3	2	1	1	3	2	1	2	1.86
AS8210	1	1	1	1	3	1	1	0	0	0	1	1	1	3	1	1	1	2	2	1	1	1.11
AS3301	2	1	1	1	1	1	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1.11
AS8434		2	1	1	3	2	2	1	1	1	1	2	1	2	2	1	1	2	2	1	2	1.82
AS5089	2		1	1	5	3	1	1	1	1	1	3	1	4	4	1	1	5	3	1	4	3.20
AS6878	1	2		1	2	2	1	2	2	1	1	2	1	2	2	1	2	2	2	1	2	1.75
AS31399	1	3	2		3	2	1	3	1	1	1	2	1	2	3	1	1	3	3	1	3	2.39
AS6667	3	1	1	1		1	2	1	1	1	1	2	1	2	4	1	1	2	3	1	4	2.11
AS1103	1	0	1	1	4		2	2	1	1	1	2	1	2	3	1	1	2	3	1	2	1.89
AS2603	2	1	1	1	4	2		2	1	1	1	2	1	3	3	1	1	2	3	1	2	1.93
AS680	2	2	1	1	3	2	2		1	1	1	3	1	3	4	1	1	4	3	1	3	2.09
AS3269	1	0	1	1	2	1	1	1		1	1	1	1	2	2	2	1	1	2	1	1	1.30
AS3215	1	0	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1	1	1	1	1.00
AS9121	3	4	1	1	5	3	2	5	1	1		2	1	3	4	1	2	5	3	1	5	3.55
AS786	2	2	1	1	4	2	2	2	1	1	1		1	4	4	1	1	3	3	1	2	2.36
AS8404	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1	1	1	1	1.00
AS20485	3	5	1	1	5	3	2	3	1	1	1	5	1		4	1	2	5	5	3	5	3.73
AS9035	2	0	1	1	3	1	2	1	2	1	1	1	1	2		1	1	3	3	1	4	2.18
AS1267	2	2	1	1	2	1	2	1	2	1	1	2	1	2	2		1	2	2	1	2	1.77
AS3352	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1		1	1	1	1	1.07
AS6830	2	2	1	1	4	2	1	2	1	1	1	2	1	2	4	1	1		3	1	2	2.59
AS3216	4	5	1	1	5	3	2	3	1	1	1	5	1	4	4	1	2	5		3	5	3.80
AS5568	5	5	1	1	5	2	2	2	1	1	1	5	1	4	4	1	1	5	5		4	3.66
AS3356	3	2	1	1	4	1	2	2	1	1	2	2	1	3	4	1	2	2	3	1		1.80

Table 4.12: Paths from an originating AS to other ASs for November 29, 2007 (Part 2)

Chapter 5

Developing a Model Based on Complete Orders

The Internet's unexploited path diversity, which was measured in the previous chapter, led to the proposal and analysis of different routing models which would be able to take advantage of this connectivity. Section 5.1 provides a brief summary of the models which were initially proposed and were then found to be unsuitable to address this issue. Subsequently, a novel mathematical model based on complete orders was developed. Section 5.2 describes complete orders and later analyses some of its properties which were needed to apply this order in a communications network.

5.1 Initial proposals to increase the Internet's resilience

This section describes the first attempts we made at finding a routing framework which is more resilient and stable than BGP. Although these three proposals were discarded, they describe the thought processes through which this research developed, provide the necessary background to understand how the final framework was formulated, and

allow for its comparison with other similar ideas. The three original models described in this section were conceived in order to increase the resilience of the Internet; each one was carefully analysed, but none was found suitable to be applied to this network.

The proposals presented in this section tried to develop an *abstraction model* which functions as an alternative representation of the network's topology in which routing tasks could be simplified. The main advantage of developing an abstraction model is that it could create a hierarchical structure that would help to define the sections of the network that are affected when failures occur; this in turn may also be used to find the most convenient strategy to overcome failures and adapt to changes in the network's topology.

5.1.1 The arc-strong neighbourhood model

The first and simplest model attempted was based on an original concept, the **arc-strong neighbourhood**, which was used to identify a group of vertices (or ASs) that share a given arc-strong connectivity (section 4.2) value amongst themselves. A possible abstraction could be established by replacing an entire arc-strong neighbourhood with a single abstracted vertex. This vertex would then represent a highly connected set of ASs. An example of this model is shown in Figure 5.1, where digraph (A) depicts a group of vertices, enclosed inside the dotted line, with an arc-strong connectivity of 2; while digraph (B) illustrates how the 2 arc-strong neighbourhood has been abstracted using vertex X_1.

Routing in the arc-strong neighbourhood model could be accomplished by letting the ASs included in the neighbourhood to coordinate amongst themselves to develop better routing strategies. This coalition of ASs will need to act as a group of siblings in which routing information would be completely shared. However, AS's administrators

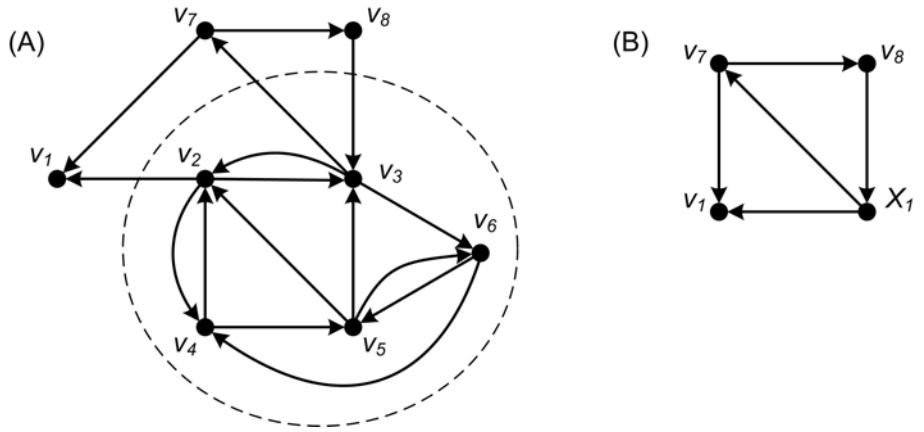


Figure 5.1: The arc-strong neighbourhood model

usually assert their independence because of commercial reasons, hence sharing private routing information is an unrealistic request to ask for most ASs.

Another problem with this type of model is that arc-strong neighbourhoods are still prone to develop Persitent Route Osscilations (PRO) (sections 2.2.2 and 3.1.1). Finally, finding the boundaries of the arc-strong neighbourhood would be, in practice, a very difficult task because networks are constantly modifying their connectivity.

5.1.2 The semicycle model

This model tries to stop the development of cyclic behaviour in the network, like PRO, by using the concept of (directed) **semicycles** used to reach a destination. Therefore, a group of arc-disjoint paths that have the same source s and destination d, are abstracted as a semicycle sd which includes all the paths from s to d. By repetitively performing this operation, it is possible to build a network abstraction like the one illustrated in Figure 5.2.

This figure also shows a new concept called the Varc or virtual arc, which is just an abstraction of a path between two ASs in which there are intermediate ASs whose only function is to transmit packets between the previous and the next AS. The Varc concept

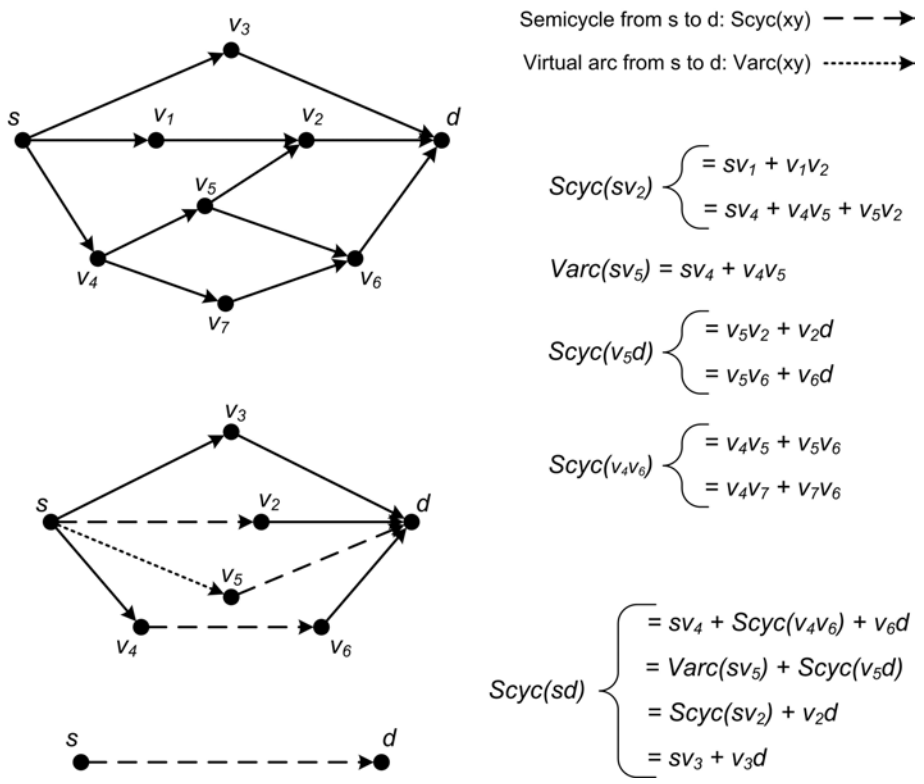


Figure 5.2: The semicycle model

did not disappear with the semicycle model because it became useful to the new routing framework which will be introduced later in this chapter; therefore a formal definition of virtual arcs will also be postponed.

The semicycle model was partially implemented in a C++ program, but during testing it was discovered that topologies that are almost completely connected cannot be uniquely abstracted using this method. An example of how this problem develops is illustrated in Figure 5.3, where it is possible to verify that there is no definitive way to determine which of the intermediate vertices (v_1, v_2 or v_3) should be used to build the abstraction model. For example, if v_1 is chosen as the intermediate node, there are 5 paths (Figure 5.3 (B)) that may be used to define semicycles from s to v_1: sv_1, $sv_2 + v_2v_1$, $sv_3 + v_3v_1$, $sv_2 + v_2v_3 + v_3v_1$ and $sv_3 + v_3v_2 + v_2v_1$; and there are also 5 paths (Figure 5.3 (C)) that could be used between v_1 and d: v_1d, $v_1v_2 + v_2d$, $v_1v_3 + v_3d$,

$v_1v_2 + v_2v_3 + v_3d$ and $v_1v_3 + v_3v_2 + v_2d$. Notice how some of these paths overlap, and how other similar sets can also be defined if either v_2 or v_3 are selected as the intermediate nodes.

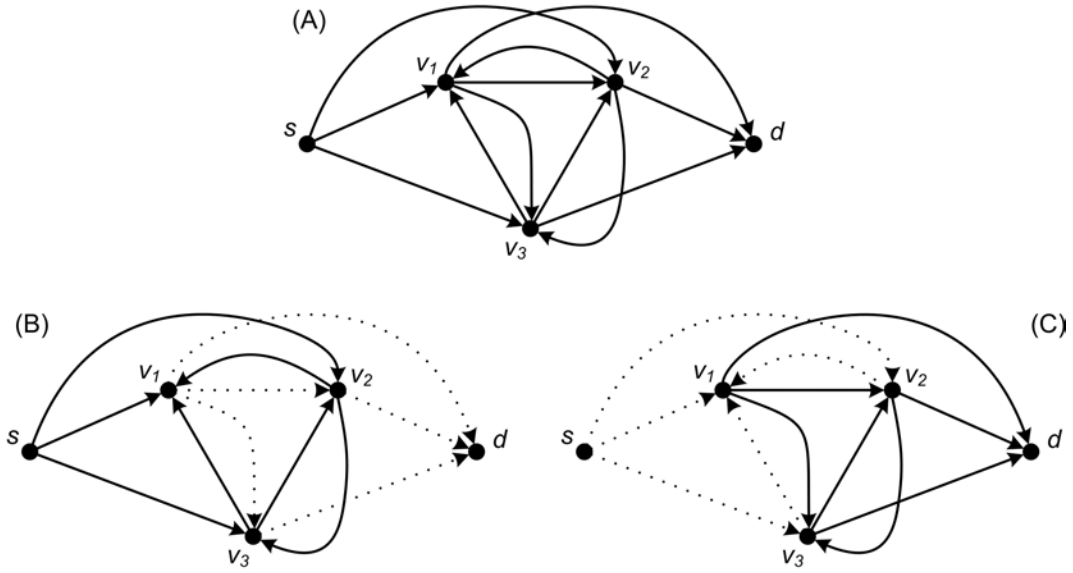


Figure 5.3: A highly connected topology and possible paths through v_1

Highly connected topologies, similar to the one illustrated in Figure 5.3, were found while testing how the C++ program would abstract the digraphs obtained in Chapter 4; hence there is proof by counter-example that such topologies arise in real environments and that, because of the ambiguity while trying to abstract these networks, this model is not amenable to algorithmic abstraction to a unique solution.

5.1.3 The dimensional model

The fact that the highly connected topology depicted in Figure 5.3 could be classified by using path-sets, in which either v_1, v_2 or v_3 were selected as intermediate vertices to the destination, led to the idea that a path-set in a graph could be used to form a *vector space* which may represent a coherent routing solution to reach the destination. Then,

in order to avoid BGP's cyclic behaviour, each vector space would be formed by a set of paths which did not form cycles.

Vector spaces are usually characterised by their dimension:

> "A vector space has *dimension* x if it can accommodate a maximum of x linearly independent vectors" [68].

Therefore, it was thought that each of the path-sets (v_1, v_2 or v_3) that could be used to define routes in Figure 5.3 (A) may somehow be used to define dimensions which could then represent strategies to solve routing in a network and increase its resilience to failures.

Initial research in vector spaces uncovered some relations between graph theory and dimensional mathematics. An example of such relation is the following theorem by Schnyder [69]:

> "A graph is planar if and only if the dimension of its incidence partially ordered set is at most 3."

Unfortunately, further research in this same field did not produce any direct relation between the dimensions of a vector space and a graph that could represent a communication network, as it did not prove possible to define a suitable notion of linear independence for paths.

On the other hand, Schnyder's theorem and other similar studies uncovered two graph-theoretic structures called *partial orders* and *complete orders*, the latter being just a subset of the former. These two mathematical concepts became the foundations for the new routing framework which is introduced in the following section.

5.2 Partial and complete orders

The models analysed in the previous section failed to provide a representation of a network which could exploit its path diversity and increase its resilience. A problem common to all these models is that they did not facilitate the definition of a unique order to transmit packets between ASs: this lack of order could later cause cyclic behaviours and instability in the network.

This section introduces the mathematical concept of *ordered sets* and provides the background needed to demonstrate the correctness of the routing framework proposed in section 6.1. Subsection 5.2.1 defines and analyses partial orders and subsection 5.2.2 does the same for the more specific concept of complete orders.

5.2.1 Partial orders

This section begins by stating some mathematical definitions which apply to digraphs and are required to describe partial orders.

Definition 13 (from [50], p. 10)**.** *A **walk** in a digraph D is an alternating sequence $W = v_1a_1v_2a_2v_3 \dots v_{n-1}a_{n-1}v_n$ of vertices v_i and arcs a_j from D such that the tail of a_i is v_i and the head of a_i is v_{i+1} for every $i = 1, 2, \dots n-1$.*

Definition 14 (from [50], p. 11)**.** *If the vertices of a walk W are distinct, W is a **path**.*

Definition 15 (from [50], p. 11)**.** *If the vertices $v_1, v_2 \dots v_{n-1}$ of a walk W are distinct, $n \geq 3$ and $v_1 = v_n$, W is a **cycle**.*

Since there is an implicit need to eliminate the influence of cycles in the announcement and destination digraphs introduced in Chapter 4, it is important to define an efficient way of removing cycles and, at the same time, produce results that can be used to develop a resilient routing framework capable of exploiting path diversity. For

this reason, the most generic and basic concept from which other definitions will arise is that of **acyclic digraph**, which is just a digraph that has no cycles. The following theorem by Harary et al. [51] states some useful properties of acyclic digraphs:

Theorem 1 (from [51], p. 268)**.** *The following properties of a digraph D are equivalent:*

1. *D has no cycles.*
2. *Every strong component (Definition 8, section 4.2) of D consists of one vertex.*
3. *Every sequence of D is a path.*
4. *It is possible to order the points of D so that its adjacency matrix is upper triangular.*
5. *It is possible to assign levels n_i to the vertices v_i in such a way that if the arc $v_i v_j$ is in D then $n_i < n_j$.*
6. *The **transitive closure** of D (D^t) is a **partial order**.*

This last statement contains two new concepts (in bold typefase) that also need to be defined. The first one, is just an operation that applies to any digraph:

Definition 16 (from [51], p. 118)**.** *The **transitive closure** D^t of a given digraph D is the minimal transitive digraph containing D and has the same set of vertices as D (Figure 5.4).*

In order to precisely define the second concept (partial order), first it is necessary to formally describe the concept of binary relation: a **binary relation** R on a set S is a subset of S^2 ($R \subseteq S \times S$). Hence, xRy is equivalent to $(x, y) \in R$, and the notation $not(xRy)$ means $(x, y) \notin R$.

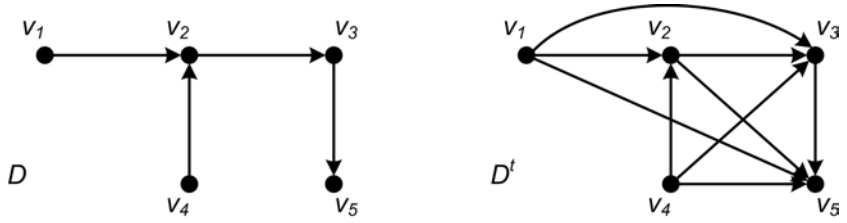


Figure 5.4: Transitive closure D^t of a digraph.

By applying these concepts to graphs, it is possible to define a digraph $D = (V, A)$ as a binary relation on the vertex set V, where each of the existing arcs $v_x v_y$ in the digraph is equivalent to xRy. Therefore, properties for binary relations can be applied to digraphs and vice versa.

The following binary relations R on a set S are common in the mathematical literature [70] and also apply to digraphs:

- R is *reflexive* if xRx for every $x \in S$. For example, Figure 5.5 (A).
- R is *irreflexive* if $not(xRx)$ for every $x \in S$. Like all graphs in Figure 5.5, except for (A).
- R is *symmetric* if $xRy \Rightarrow yRx$ for all $x, y \in S$. For example, Figure 5.5 (B).
- R is *asymmetric* if $xRy \Rightarrow not(yRx)$ for all $x, y \in S$. For example, Figure 5.5 (C) and (D).
- R is *transitive* if $(xRy, yRz) \Rightarrow xRz$ for all $x, y, z \in S$. For example, Figure 5.5 (C).
- R is *negatively transitive* if $xRy \Rightarrow (xRz$ or $zRy)$ for all $x, y, z \in S$. For example, Figure 5.5 (D).
- R is *complete* if $x \neq y \Rightarrow (xRy$ or $yRx)$ for all $x, y \in S$. Like all graphs in Figure 5.5, except for (A).

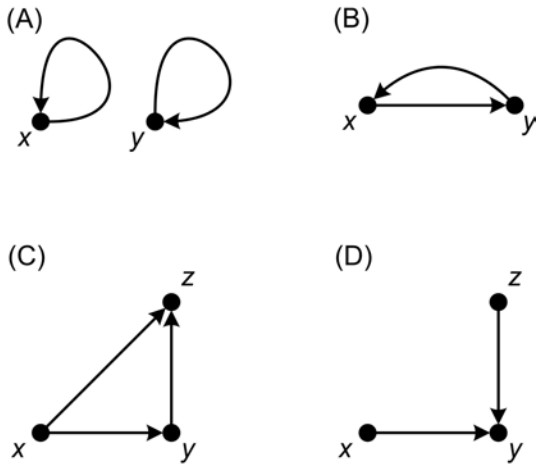


Figure 5.5: Binary Relations

Based on the properties of binary relations, it is possible to define a partial order:

Definition 17. *A binary relation R on a set S is a **partial order** if it is irreflexive, asymmetric and transitive.*

Because partial orders are based on binary relations, the smallest possible partial order must possess at least 2 elements in its set S. Since binary relations also apply to digraphs, a digraph that is irreflexive, asymmetric and transitive is also called a partial order. An example of a partial order digraph is provided in Figure 5.6 (A).

The last binary relation in the list, the *complete* binary relation, is analogous to saying that the relation must exist between any two members of the set S. If the relationship fails to exist between any two members, then it is said that the relation is incomplete. The complete binary relation is needed to define a complete order, which is a more specific type of partial order:

Definition 18. *If a partial order relationship is also complete, it is called a **complete order**, **total order** or **linear order**.*

Two digraphs that have a one-to-one correspondence between their vertices and their arcs are called **isomorphic**. All complete orders with the same number of vertices

are isomorphic. An example of a complete order of 5 vertices is shown in Figure 5.6 (B).

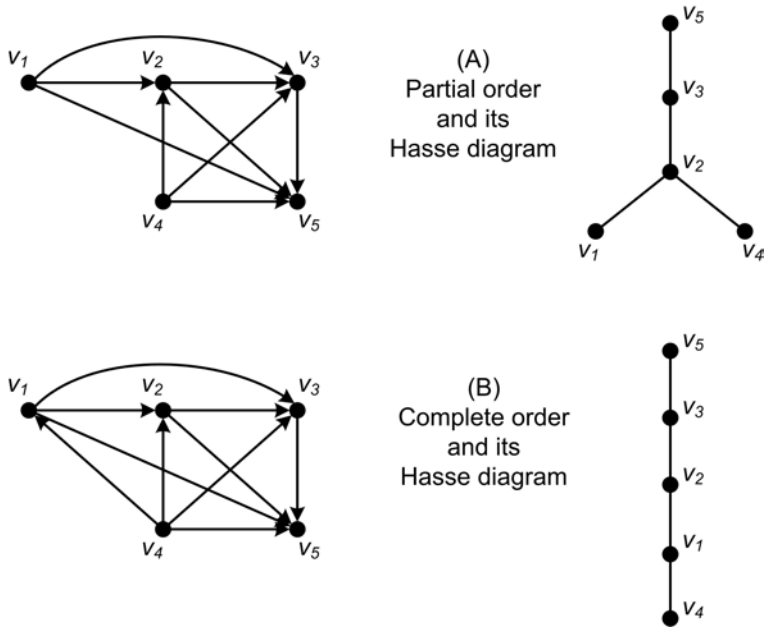


Figure 5.6: Hasse diagram examples

The notation used in the literature to define a *partial order on a set* S is $(S, \prec)$ [70] and is also sometimes called a **poset** (partially ordered set). A common representation of posets is the **Hasse diagram**. To construct this diagram from a digraph D, first order and draw the vertices of D in vertical order such that x is below y if xRy, then draw all the digraph's arcs which should have an upward direction if the order was done correctly, then delete all arcs that could be implied by the transitive property and finally delete the direction indicators on the arcs. Examples of Hasse diagrams for a partial and a complete order are shown in Figure 5.6.

Figure 5.6 also shows why a complete ordered subset of a partial order is called a **chain**. For example, in the Hasse diagram of the partial order the vertices v_1, v_2, v_3 and v_5 form a chain, denoted $C(v_1, v_2, v_3, v_5)$. The reason why Hasse diagrams are an appropriate and valid representation of partial orders is detailed in property 5 of

Theorem 1.

Finally, the **height** $H(S, \prec)$ of a partial order $(S, \prec)$ is one less than the number of vertices in a maximum length chain in $(S, \prec)$. So in Figure 5.6 the height of the partial order is 3, because both maximum length chains ($C(v_1, v_2, v_3, v_5)$ and $C(v_4, v_2, v_3, v_5)$) have 4 vertices; and the height of the complete order is 4.

5.2.2 Complete orders

Since complete orders are a subset of partial orders, the former inherit all the characteristics of the latter, but complete orders also have their own additional set of properties. The following theorem provides some of the most significant characteristics of this type of order.

Theorem 2 (from [51], p. 281)**.** *The following statements for an acyclic digraph D with n vertices are equivalent:*

1. *D is **maximal**.*
2. *Given any 3 vertices of D, they can be labelled v_1, v_2, v_3 such that arcs v_1v_2, v_2v_3 and v_1v_3 are in D. In other words, any 3 vertices are transitive.*
3. *D has $n(n-1)/2$ arcs.*
4. *D is complete.*
5. *D is a **complete order**.*

In the previous theorem, the term **maximal** means that adding any new arc on the digraph forms a cycle. Therefore, a complete order is the maximal acyclic digraph that can be defined given a set of vertices. This is a crucial property for the purposes of

routing because it implies that *the digraph with the largest number of arcs that is still acyclic and provides the best connectivity is the complete order*.

A crucial property of complete orders which immediately applies to communication networks is:

Theorem 3 (from [51], p. 297)**.** *Every complete order has a* ***unique transmitter*** *and a* ***unique receiver****.*

This means that the most connected acyclic digraph between a unique transmitter (source) and a unique receiver (destination) is the complete order. Therefore, the previous theorem could apply to destination digraphs in which an AS uses the different paths, announced by its neighbours, to send information to a destination AS.

In order to provide a complete theoretical foundation of the routing framework proposed by this thesis, there are two important characteristics of complete orders that require further mathematical and logical analysis: the connectivity of complete orders (subsection 5.2.2.1), and the structural properties of such orders which are related to topological modifications (subsection 5.2.2.2).

5.2.2.1 The complete order's connectivity

The following theorem, which is an original contribution of this thesis, defines the path diversity available in a complete order between the transmitter and the receiver. Although it was argued, in section 4.2, that arc-strong connectivity and arc-disjoint paths are better suited to measure the Internet's path diversity, the following theorems also apply for internarlly disjoint paths, which are related to vertex-strong connectivity.

Theorem 4. *If C is a complete order with $n \geq 2$ vertices, then there are $n-1$ arc-disjoint (internally disjoint) paths from the transmitter to the receiver.*

Proof. This is proven by induction:

The smallest possible 2-vertex complete order has only 1 path: the arc v_1v_2. Now assume that a complete order with n vertices has $n-1$ arc-disjoint (internally disjoint) paths between the transmitter v_1 and the receiver v_n. Then, the transitive property implies that the complete order with $n+1$ vertices also has $n-1$ arc-disjoint (internally disjoint) paths between the transmitter v_1 and its receiver v_{n+1}. In addition to these, the arc v_1v_{n+1} provides an additional arc-disjoint (internally disjoint) path, because of the completeness property. This adds to a total of n arc-disjoint (internally disjoint) paths from v_1 to v_{n+1}. □

The number $n-1$ of arc-disjoint paths in the previous theorem applies only between the transmitter and the receiver of a complete order. This is not the same as the *arc-strong connectivity* discussed in section 4.2, which applies to the vertex set V of a digraph $D = (V, A)$. Because $n-1$ does not describe the connectivity of all the vertices in a digraph, this number could be greater than the arc-strong connectivity.

Also notice that $n-1$ is equal to the height of the complete order $H(C, \prec)$, which means that the maximum number of arc-disjoint paths and the height of a complete order are the same.

Theorem 4 also demonstrates that the smallest complete order that offers any path diversity has 3 vertices, and thus, just 2 arc-disjoint paths. In a communication network, these paths could be used to either distribute data traffic, or one path could be used as the main path, while the other stays as a backup path. Complete orders with more than 3 vertices offer more path diversity and complex traffic path arrangements for their corresponding networks. In order to analyse these arrangements, first it is necessary to find the **number of arcs that the arc-disjoint paths from the transmiter to the receiver will use** (u) by means of the following theorem, which is also an original

contribution of this thesis.

Theorem 5. *If C is a complete order with $n \geq 2$ vertices, then all of the $n-1$ arc-disjoint (internally disjoint) paths from the transmitter to the receiver uses exactly $u = 2n-3$ arcs.*

Proof. This is also proven by induction:

The smallest possible 2-vertex complete order uses only 1 arc for its only path: the arc v_1v_2. Now assume that a complete order with n vertices uses $2n-3$ arcs on its $n-1$ arc-disjoint (internally disjoint) paths between the transmitter v_1 and the receiver v_n. Most of these arcs follow a path from the transmitter to an intermediate node and then to the receiver, and there is just 1 direct path from the transmiter to the receiver.

Then, because of its transitivity, the complete order with $n+1$ vertices must also use $2n-3$ arcs on its arc-disjoint (internally disjoint) paths between the transmitter v_1 and the receiver v_{n+1} plus the arc v_nv_{n+1} and, because of its completeness, the arc v_1v_{n+1}. This forms a total of:

$$u = 2n - 3 + 1 + 1 = 2(n+1) - 3.$$

□

Then, finding the **number of unused or remaining arcs** (r) is a trivial task.

Corollary 6. *If C is a complete order with $n \geq 2$ vertices, there are exactly $r = (n-2)(n-3)/2$ arcs that are not used in all the $n-1$ arc-disjoint (internally disjoint) paths from the transmiter to the receiver.*

Proof. This result is obtained by substracting the number of arcs used by the $n-1$ arc-disjoint paths (Theorem 5) from the complete order's total number of arcs (Theorem 2):

$$r = \frac{n(n-1)}{2} - (2n-3) = \frac{(n-2)(n-3)}{2}.$$

□

An important property of the remaining arcs (r) is that they increase quadratically with respect to the number of vertices in the complete order (n), while the number of arcs used by the arc-disjoint paths (u) increases linearly. By inspection of Table 5.1, it is possible to see how these quantities change with the size of the complete order. This table demonstrates that from $n = 8$, r becomes larger than u; this means that from this point, more resources would be needed to store and manage remaining arcs than for used arcs. This may translate into ASs and their routers spending a larger percentage of their processing time dealing with chain coordination which is not directly related to the more efficient arc-disjoint paths.

n	u	r
2	1	0
3	3	0
4	5	1
5	7	3
6	9	6
7	11	10
8	13	15
9	15	21
10	17	36

Table 5.1: Complete order's size (n) vs. the number of used (u) and remaining (r) arcs in all the arc-disjoint paths from the transmitter to the receiver

Table 5.1 leads to the conclusion that, altough complete orders are an acyclic structure which offers good path diversity, they should be limited in size because of the large number of arcs that will not be needed to support arc-disjoint paths. Therefore, it must be considered that in practice, complete orders that grow beyond a **maximum allowed size** of 7 vertices could use too much memory and too many computational resources

to be considered a sensible solution for a communication network's routing needs.

5.2.2.2 The complete order's structural properties

The following theorems demonstrate the advantages and challenges of modifying the structure of complete orders. The next theorem demonstrates how easy it is to reduce the height of a complete order:

Theorem 7 (from [51], p. 297)**.** *If C is a complete order with at least 3 vertices, and if v is any vertex of C, then $C - v$ is also a complete order.*

This previous theorem demonstrates that, when a chain of n vertices needs to eliminate a vertex, it is possible to end with the chain of $n - 1$ vertices.

The next theorem and corollary analyse how many arcs are needed to increase the height of a complete order. Both are original contributions of this thesis:

Theorem 8. *If C is a complete order with $n \geq 2$ vertices, then the augmented complete order $C + 1$ which has exactly 1 more vertex than C requires n new arcs.*

Proof. C has $\frac{n(n-1)}{2}$ arcs (from Theorem 2). So $C + 1$ has $\frac{(n+1)n}{2}$ arcs.

Hence the difference in the number of arcs between these complete orders is:

$$\frac{n^2 + n}{2} - \frac{n^2 - n}{2} = n.$$

□

Corollary 9. *Increasing the number of vertices and height of a complete order by 1 requires at least 2 new arcs.*

Proof. Since the smallest possible complete order has 2 vertices, then 2 new arcs are needed to augment to the complete order of 3 vertices. □

The previous theorem and corollary demonstrate why a single arc is not enough to increase the height of a chain.

The following theorem explores the effects of combining two interlaced chains to produce a longer chain. However, first it is necessary to precisely define interlaced chains and their common arcs:

Definition 19. *A pair of chains are **interlaced** if the last vertices of one of these chains are also the first vertices of the other one, and the common vertices follow the same order in both chains.*

Definition 20. *The **common arcs** of two interlaced chains are the arcs that are common in both chains at their Hasse diagrams.*

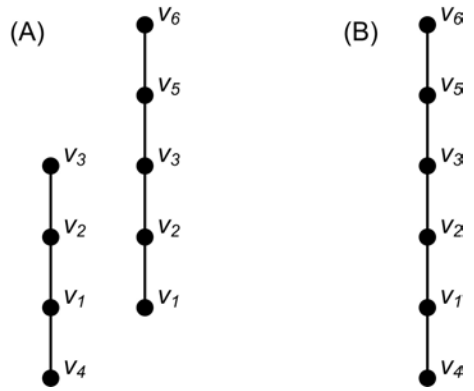


Figure 5.7: Two interlaced chains and the final combined chain

Figure 5.7 (A) shows the Hasse diagrams of two interlaced chains, where v_1v_2 and v_2v_3 are the common arcs. Part (B) shows the Hasse diagram of the combined chain. A special case of interlaced chains is the **cascaded chains**, in which the last vertex of a chain is the first vertex of the other and the number of common arcs is zero. The previous definitions lead to the following theorem and corollary:

Theorem 10. *When two interlaced chains with x common arcs and heights m and n are combined into a longer chain, the height of this longer chain is equal to $m + n - x$.*

Proof. The height of a chain is equal to the number of arcs in its Hasse diagram (see Figure 5.6), and the height of the combined chain must be equal to the height of the common arcs plus the height of the other two chains minus the common arcs:

$$x + (m - x) + (n - x) = m + n - x.$$

□

Corollary 11. *A chain which has been formed by combining two interlaced chains with x common arcs and heights m and n, needs $a = (m - x)(n - x)$ additional arcs.*

Proof. The total number of arcs needed in the combined chain is:

$$\frac{(m + n - x + 1)(m + n - x)}{2}$$

And the number of arcs already provided by the interlaced chains is:

$$\frac{m(m-1)}{2} + \frac{n(n-1)}{2} - \frac{x(x-1)}{2}$$

Then, the number of additional arcs a is obtained by substracting the later from the former:

$$\begin{aligned} a &= \frac{m^2 + n^2 + x^2 + 2mn - 2mx - 2nx + m + n - x}{2} \\ &\quad - \frac{m^2 + m + n^2 + n - x^2 - x}{2} \\ a &= x^2 + mn - mx - nx \\ a &= (m - x)(n - x) \end{aligned}$$

□

The previous theorem and corollary help determining the total number of arcs needed to complete a combined chain from two smaller interlaced chains, and they allow the same calculation for two cascaded chains.

This section has covered the mathematical and structural properties of complete orders, which provide the largest number of arc-disjoint paths between a unique transmitter and a unique receiver. A proposal on how to apply these concepts to a new routing framework is provided in the following chapter.

Chapter 6

Chain Routing: A Proposal to Increase the Internet's Resilience

This chapter describes a new routing framework, called Chain Routing, designed to be more resilient and robust than the current BGP implementation. By taking advantage of the Internet's potentially rich connectivity, Chain Routing may overcome several of the limitations of BGP and offers the possibility of increasing the Internet's resilience. This new proposal is fully described in section 6.1. Then, section 6.2 describes a numerical experiment used to test the feasibility of employing this framework in realistic topologies. Finally, section 6.3 discusses the main advantages and limitations of Chain Routing.

6.1 Chain Routing

As was described in Chapter 2, BGP routers need to select a path to reach each destination before announcing it to its neighbour routers, this is called the preferred path rule (PPR). PRO may develop because policies could interfere with BGP's path selection

process and cause cyclic behaviour between a group of routers that are each trying to define its own preferred path to a destination.

Chain Routing is a new routing protocol that uses complete orders (chains) as the basis for determining a set of valid routes to a destination. Such a protocol has two main advantages:

1. Chain Routing uses the highest number of possible acyclic directed routes between two nodes as its natural unit of path diversity. Note that whenever the terms *node* or *vertex* in a graph are mentioned, it is necessary to consider both of them as representing ASs in a network. Most of the time, the network is represented as a destination digraph (section 4.1).

2. The second Chain Routing advantage is that it requires a simple data structure to store several paths, though this would also mean that coordination between different ASs will be required to establish and maintain operational chains.

Since Chain Routing has been thought either as a replacement or as an enhancement to BGP, it will also need to operate as a **decentralised routing algorithm** ([71], p. 353) in which each AS will take its routing decisions based on the announcements and permissions that it has received from its neighbour ASs. However, contrary to the current BGP functionality, more coordination between ASs will be needed to define paths to a destination.

The following subsections provide a gradual introduction to the principles of this new routing protocol. Subsection 6.1.1 presents the basic ideas on how to apply complete orders in a communication network. Subsection 6.1.2 introduces a new structure needed to logically modify the network's topology so it is suitable to be represented as a complete order. Subsection 6.1.3 describes operations related to the incremental topological discovery and use of Chain Routing in a realistic network environment. Then,

subsection 6.1.4 shows how to combine the functionalities described in the previous subsections. Subsection 6.1.5 reflects on how to use this new framework to perform routing in the Internet. Finally, subsection 6.1.6 analyses other routing protocols that offer resilience solutions which are somehow similar to Chain Routing.

6.1.1 Basic Chain Routing principles

The main objective of Chain Routing is to build a complete order or chain (C), between a **source** s and a **destination** d, including as many intermediate vertices as necessary, provided the chain's *maximum allowed size* of 7 vertices is maintained as proposed in section 5.2.2.1. Vertices included in this set, other than s and d, will be called the **intermediate nodes** from s to d. Because of the completeness and maximality properties of complete orders that were discussed in section 5.2.2, a chain can be used to represent a *self-contained strategy* to reach d, even when some of the intermediate vertices or links fail.

As an example of how to apply Chain Routing, consider the complete order depicted in Figure 5.6 (B) as representing a real destination digraph from the source AS (v_4) to the destination AS (v_5). There are many possible paths v_4 may choose to send packets to v_5: it could balance the traffic load between the different $n-1$ arc-disjoint paths that exist between s and d, or it could prefer to use the direct link v_4v_5 and leave the other paths as backup. If v_4 chooses the latter option, this AS knows that if link v_4v_5 fails, it has 3 alternative paths through v_1, v_2 and v_3; and if any of these intermediate ASs also fails, v_4 still has 2 alternative paths to reach v_5. Notice that because the original graph is acyclic, it is possible to guarantee that, regardless of which vertex or arc fails, cyclic paths cannot develop in this network.

Because of Chain Routing's distributed nature, each AS will need to discover and

define its own set of chains that reach different destinations. However, regardless of these individual discovery tasks, each chain will still need coordination between the source and the intermediate nodes. Such cooperation is required to avoid instabilities and congestion in the network. For example, if v_4, at the complete order depicted in Figure 5.6 (B), wants to balance its traffic load between the 4 available arc-disjoint paths it has to v_5, each of the intermediate nodes (v_1, v_2 and v_3) will need to understand the routing strategy implemented at v_4 because, if they are not aware that v_4 is distributing loads between each intermediate node, v_1 could use its path through v_2 and create congestion problems for v_2. Many other conflicting scenarios are possible, hence the importance of maintaining coordination between the nodes that form a chain.

Since communication links and some policies between ASs are bidirectional in real network environments, in some circumstances more than one solution could be used to define a chain. In such cases, it is possible to take advantage of the mathematical properties of chains and assign levels to the ASs that will form the complete order (Theorem 1). These levels could be used to solve disputes in which there is more than one solution to the definition of a chain in a network. In such cases, the source s should be assigned to a level 1 and the nodes with highest preference would be assigned closer to the source's level until the maximum level n is given to the destination d.

6.1.2 Introducing virtual arcs

Some ASs may not have enough connectivity to allow defining a chain with other ASs, but they may still permit forming chains around them. To solve this connectivity problem, a structure that allows to bypass ASs by including them in an abstracted form is proposed. This new structure is called a Varc and is defined below:

Definition 21. *A **Varc** or **virtual arc** is a structure that represents a set of vertices and*

their adjacent arcs that follow a directed path from an initial vertex x to a final vertex y, where the directed path needs to be abstracted in order to allow x and y to be vertices of a chain.

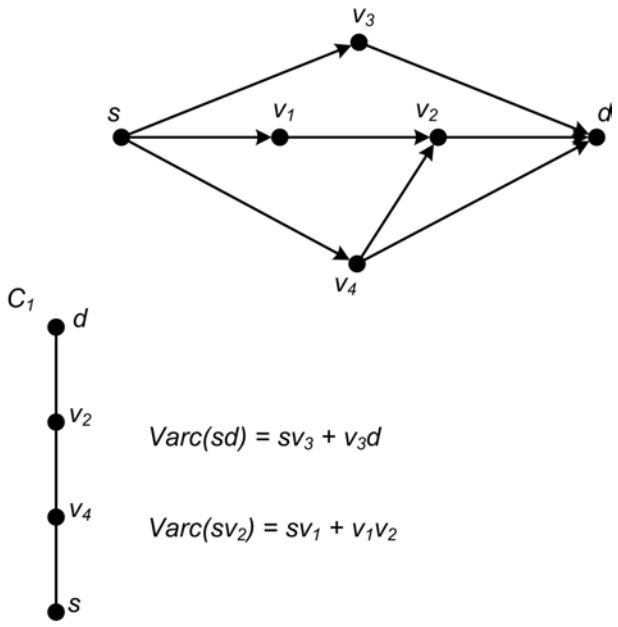


Figure 6.1: Simple Varc example

There are two type of Varcs in the Chain Routing protocol:

Simple Varc is used to abstract a single path with no options. This means a path that starts with a vertex of outdegree $\geq$2, followed by one or more vertices that can only be used to transit data between the previous and the next vertex, and terminating at another vertex of indegree $\geq$2. This type of Varc can be identified easily because there are no alternative paths between the initial and final vertices. Examples of simple Varcs are $Varc(sd)$ and $Varc(sv_2)$ in Figure 6.1.

Functional Varc is created in order to define a chain which needs to use the initial and final vertices of the Varc as its intermediate nodes. This type of Varc is usually

created incrementally and while the routing protocol is discovering the network's topology, hence it will be exemplified in the following subsection.

6.1.3 Incremental operations in Chain Routing

Since Chain Routing has been devised as a decentralised routing algorithm, this implies that this protocol will need to perform incremental topology discovery and also modify its routing parameters as the network's topology changes. This means that some of the chains and structures that this protocol finds may need to be modified when more topological information becomes available. Some of the strategies that have been designed to accommodate this type of functionality are described here.

As new vertices and their corresponding links are discovered, they will need to be recorded in a data structure. This data structure will need to accommodate three different types of **basic structures**:

arc This is the most basic structure that only records a single link between two ASs.

Varc Introduced in subsection 6.1.2. A Varc describes a path which, in terms of the data structure, is just a sequence of arcs.

chain Records a complete order. Chains are usually composed of arcs, but they can also include Varcs and other chains.

Every new link must be initially recorded as an arc in the data structure, but there are three other possibilities that must be considered for each new arc:

1. The arc may help to create a new or a longer chain. As stated in Corollary 9, a single arc is not enough to increase the height of a complete order. But it is possible that the discovered arc is the one needed to either complete a 3-vertex chain, or form a longer chain from a predefined one.

2. The new arc could allow to combine two interlaced chains into a longer one (Definition 19).

3. The arc could be included as part of a simple Varc. Notice that adding an arc to a simple Varc will not help to increase the connectivity of the network, just to obtain an abstracted structure that can later be used to build other chains.

Since Chain Routing's main objective is to form chains with the largest height, every discovered arc should be matched and tested in order to determine if it can be used as described above. Links that can be included as part of a simple Varc have the least preference of the three options because they do not increase the network's path diversity, but the other two options are only preferred depending on which one will form the longest chain.

The data structure and matching operations described above, help to keep a correspondence between the network's topology and the acyclic structures needed to implement Chain Routing. Sometimes this correspondence may not be obvious; in these cases the following strategy, called series of chains, helps to develop this correspondence when a unique solution cannot be attained:

Definition 22. *A **series of chains** is an operation in which a set of vertices in a path is used to define more than one chain, because there is not a preferred transitive node which could be used to build the complete order.*

An example of how to use the *series of chains* concept is provided in Figure 6.2. Here it is possible to see that s could use three different nodes to define a chain to d: x_1, x_2 and x_3. Since there is no other criteria that could be used to prefer one vertex over the other, and because the connectivity of any of these ASs may change in the future, a series of chains would allow to record the three different solutions that could

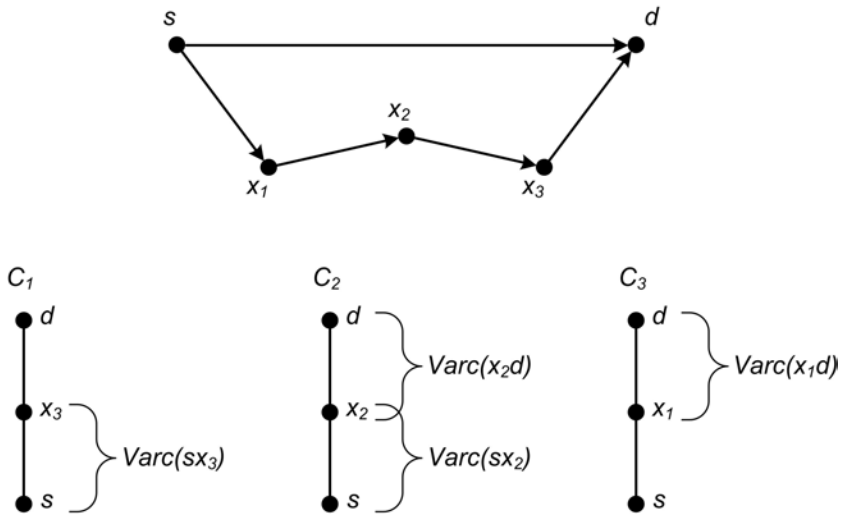


Figure 6.2: Series of chains

be used until a best one could be later selected. These three solution chains (C_1, C_2 and C_3) are also depicted in Figure 6.2. Notice that the Varcs defined for each chain are functional Varcs that allow to define chains, rather than reflecting the network's topology. At this point, any of these chains could be selected to route packets between s and d: after all, they all need to route information through the same set of nodes. Eventually, the connectivity of either x_1, x_2 or x_3 may change and any of these chains could be preferred over the others to become a definitive solution. For example, if x_2 discovers an alternative path to d (not depicted), then this vertex may become better connected than the others and C_2 could become the preferred solution.

6.1.4 Combined Chain Routing functionality

As was stated at the beginning of this section, the main operational objective of Chain Routing is to exploit complete orders to define felxibly alternative paths to a destination. Sometimes, it will not be possible to define a chain between source and destination because, at some point in the graph, the topology only allows one path between s and

d. This, in graph theory, is called a bridge:

Definition 23. *A **bridge** in a digraph is a path that, if deleted, disconnects the digraph.*

Bridges have no path diversity, but Chain Routing will still need to accommodate the existence of bridges by allowing to combine chains and paths to reach a destination. Strictly speaking, a bridge could be represented by either an arc or a simple Varc in the data structure; therefore Chain Routing must allow the concatenation of chains, Varcs and arcs to reach destinations that can only be reached through bridges. Figure 6.3 shows two examples of how bridges may appear in a destination digraph and how to use the basic structures to reach the destination.

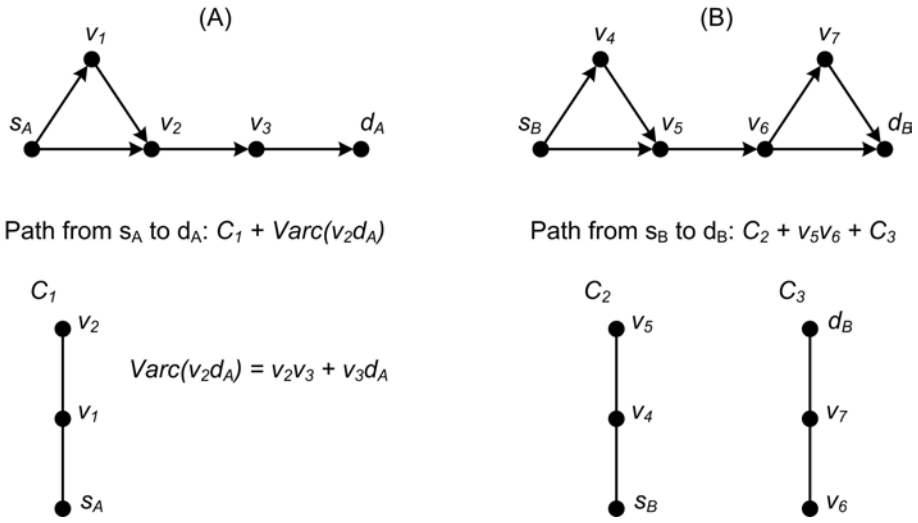


Figure 6.3: Examples of topologies with bridges

Chain Routing needs to record and keep track of bridges because of the following reasons:

1. The topology of the Internet is always changing, therefore it is plausible that a bridge may become part of a chain as the network evolves. For example, adding the arc v_1d_A in Figure 6.3 (A) will create the chain $C(s_A, v_1, d_A)$.

2. Traffic engineering could exploit bridges with high capacity. For example, it is possible that arc v_5v_6 in Figure 6.3 (B) has higher capacity than s_Bv_5 or $s_Bv_4 + v_4v_5$, but by distributing the traffic between these two paths it is possible to take advantage of the capacity provided by v_5v_6.

To demonstrate how to combine the ideas and strategies introduced in the previous sections, a final Chain Routing example is provided by Figure 6.4. This Figure shows how a destination digraph with 10 vertices may be represented by a chain of height 3. This is possible because, by nesting chains and Varcs, it is possible to abstract vertices and arcs of this destination digraph. Note that the main chain in this network, C_1, has defined v_4 and v_2 as the two intermediate nodes that can be used to take advantage of the 3 arc-disjoint paths that exist to reach d. Also note that C_1 only has a height of 3, which shows that even when the maximum allowed size of a chain is 7 vertices, that in itself does not mean that the number of nodes the destination digraph has is also restricted by this number. Nesting structures and virtual arcs allow to condense the network's topology and to focus attention on the ASs central to the resilience of the network.

In order to define the structures shown in Figure 6.4, a great degree of coordination between the ASs in this network is needed. For example, v_4 has four valid options when selecting an intermediate node to d: v_2, v_5, v_6 and v_7; but only v_2 allows the definition of C_1 as depicted. Therefore, s needs to coordinate with v_4 in order to obtain the complex solution shown in this picture. On the other hand, a much simpler solution could be produced if v_4 decides to hide some of its path information from s, for example: C_3 and $Varc(v_4v_6)$. This decision will help to create more compact data structures at the price of having less path diversity information of the network.

The mechanisms to perform the operations described above have not been completely defined because this research focuses on the correctness and application of

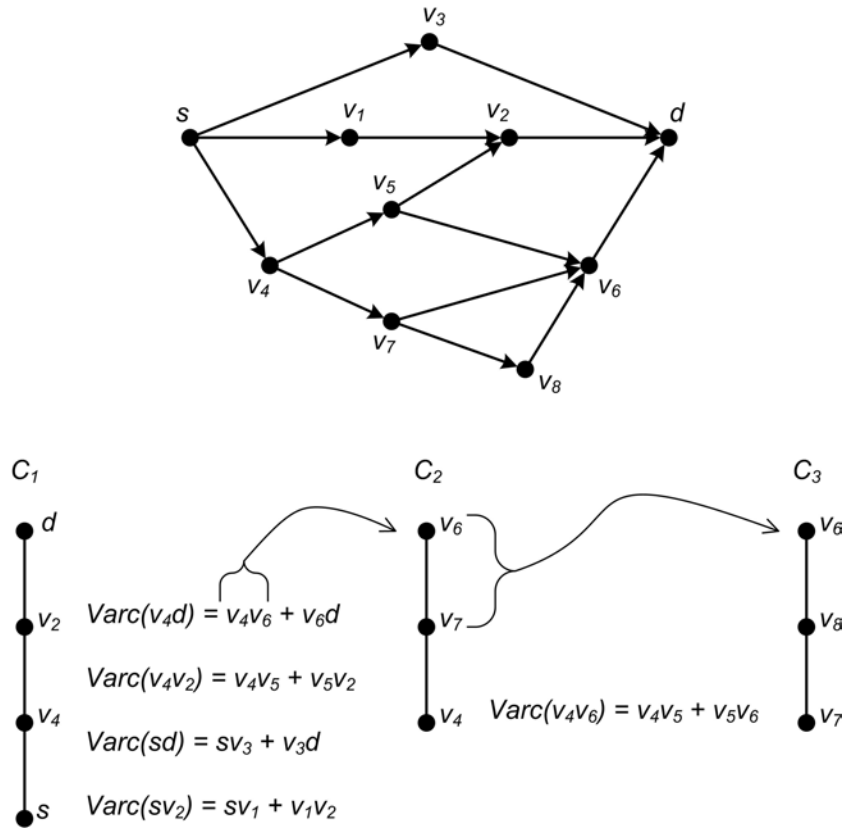


Figure 6.4: Chain Routing example

chains as a routing framework, rather than defining its specific implementation in a network. Nevertheless, some preliminary steps on how this implementation could be done are discussed later in section 6.3.2.

6.1.5 Routing with chains

Up to this point, this section has described how to use chains, virtual arcs and arcs to build acyclic structures that encode the path diversity available between s and d. How to employ this path diversity is the same as determining how to route information from s to d. Since the main objective of this thesis is not to fully define a routing protocol, but to determine if it is possible to perform stable routing in a network that is constantly modifying its topology, this section just analyses some of the routing strategies that

may be used in a final implementation of Chain Routing.

Before describing how chains could be employed to perform routing, first it is necessary to remark that Chain Routing must operate on two different timescales:

Immediate timescale is used to switch traffic within a chain to minimise the impact of a network failure. This timescale should act in the range of miliseconds.

Long-term timescale is used to search for longer or more stable chains that could offer a better solution than the currently selected chain. This timescale is a strategy to explore the topology of the network without causing instabilities to other ASs and it should act in the range of minutes.

At the immediate timescale, Chain Routing should use the selected chain to reach d for as long as there is any viable path to d, and only when no path is available should AS s switch to a different chain or path. While at the long-term scale, the source AS should constantly be looking for a more stable chain to reach d, but only when all the involved ASs have reached an agreement, AS s will be free to switch traffic to the new chain. This timescales mechanism would allow a very stable and reliable connectivity between s and d where only necessary changes are sudden.

There are many strategies that could be applied to route packets between s and d by using chains. The following subsections analyse two distinct potential approaches.

6.1.5.1 Routing by using maximal traffic distribution

The maximal traffic distribution approach uses all the arc-disjoint paths available to send information packets to the destination; this implies that a mechanism to distribute traffic is always functioning and that coordination between all the vertices in the chain is constant in order to avoid congestion. This approach takes full advantage of the

network's path diversity and should allow more traffic to travel between s and d. Conversely, if an entire AS fails or does not provide correct information about its traffic load or connectivity, this system may be prone to congestion problems and instabilities because more ASs are affected when a problem occurs.

6.1.5.2 Routing by using the best path

In this approach a *best path* is selected and used to reach the destination, while the other alternative paths are kept on standby and used when the best path becomes unavailable. Selecting the best path could be done by comparing a set of predefined characteristics from all available paths. This is not too different from current BGP functionality, and the only advantage is that alternative paths are always pre-computed.

The two problems with adopting the best path approach are that it does not support traffic engineering, and that most segments of the chain will normally not be used. Both problems are caused because this strategy does not exploit the network's path diversity.

Besides the two routing strategies described above, there are other possible approaches but in all of them there will always be a trade-off between the network's path diversity and coordination between the ASs that form the chain.

Chain Routing's strategies and rules have been described at a conceptual level without considering how to implement it in a real router or computer. What sort of behaviour can be expected from this routing protocol when it is implemented in a real network? A large amount of experimentation is needed before a practical implementation of Chain Routing can be fully considered. However, a first step towards answering this question has been accomplished by implementing a computer program that is able to discover chains in a digraph. Such a program is further described in section 6.2. However, it is important to first compare Chain Routing against other routing strategies that are also trying to increase resilience through the use of greater path diversity.

6.1.6 Chain Routing and other protocols

As it was described in the previous section, Chain Routing provides a structure that allows to know and possibly use alternative paths to reach a destination. As it should be expected, Chain Routing is not the first attempt to pre-compute paths in a network in order to minimise the impact of failures. Examples of other routing protocols that possess this characteristic are:

- Pre-configured cycles (p-cycles), which were proposed to provide ATM networks with path restoration times similar to the ones obtained for SONET networks [72]. Although it seems feasable to implement p-cycles in an inter-AS network, the cycles would need to operate in a bidirectional environment which, as has been demonstrated in section 4.1, cannot be implemented in a network that faces directed restrictions due to the policies applied between the ASs that form such network.

- The resilient recursive routing (R^3) protocol proposed by Constantinou et al. [73] provided an abstraction model based on cycles, which allows to perform resilient routing in an Internet-like network. Unfortunately, their model is also based in bidirectional networks.

None of these proposals have been designed to perform in a directed network which may not contain cycles. Therefore, they are not suitable to perform routing in inter-AS networks which are restricted by policies.

Another protocol which briefly explores the potential of using order in time and topology was introduced by Park and Corson [74]. Their temporally-ordered routing algorithm (TORA) was originally developed to perform routing in a wireless ad-hoc network, although its main principles could be applied in other types of networks.

TORA defines a temporary and a topological order, but it does not identify which nodes possess a transitive relation with other nodes, hence it does not establish a complete order beteween nodes. This results in a routing system that cannot take advantage of the path diversity available in the network, which is one of the main advantages of Chain Routing over BGP.

Finally, Kushman et al. [75] proposed Resilient BGP (R-BGP) as an enhancement to BGP which pre-computes an alternative route for each BGP destination. Their proposal also takes into consideration that an order in time, as described in [41] and [42], is needed to achieve faster network convergence. They also demonstrated that in order to guarantee resilience, the alternative path needs to be disjoint from the primary path. A similar mechanism but for Chain Routing will be described in section 7.1.1. Unfortunately, R-BGP does not advocate the development of traffic engineering and does not specifically require the definition of a topological order.

6.2 Implementing Chain Routing using C++

To prove that it is possible to define complete orders in a highly connected digraph a C++ program was developed: ChainRtg. This program tries to find as many chains as possible to the Top-45 destinations using the announcement digraphs produced in Chapter 4. Since it is intended that Chain Routing works as a decentralised routing algorithm, every AS in the Internet will need to discover and establish its own set of chains for all its known destinations. Thefore, a similar program to the one developed for this numerical analysis may run locally at each AS's router that needs to define such chains.

The ChainRtg C++ code has been included in Appendix C. This program finds chains in a digraph by using an adapted version of the Breadth-First Search (BFS)

algorithm [50]. The reason for employing BFS is because it discovers first the direct routes from a source vertex, followed by the paths that contain intermediate nodes. Therefore, the nodes that have already been visited by BFS could represent a transitive relation with a node that has been discovered before.

The following subsection (6.2.1) describes the data structure developed to accommodate the basic structures introduced in section 6.1.3. Then, subsection 6.2.2 shows how the BFS algorithm was modified and applied to discover chains. Subsection 6.2.3 provides the details of how to process the different types of chains discovered by the modified BFS algorithm, and subsection 6.2.4 explores the shortcomings of the algorithms used. Finally, subsection 6.2.5 analyses the results obtained from this numerical experiment.

6.2.1 The Chain Routing data structure

The **Chain Routing data structure (CRDS)** is responsible for storing and maintaining the data structure where the basic structures (chain, Varc and arc) are stored. This structure must reflect the topology of the network and also needs to consider that chains and virtual arcs are *potentially complex structures* that could be formed by either arcs or other complex structures.

The fundamental idea of the CRDS is that there are different **levels** where the different structures that form the network are recorded. For example, in Figure 6.3 the path to both destinations (d_A and d_B) are formed of a sequence of chains and bridges. Therefore the basic structures that need to be concatenated to describe how to reach the destinations are stored at the main level: level 0. At the next level, level 1, a description of each **segment** that forms a structure at level 0 is included, the only exception being the arc, which cannot be decomposed into any other basic structure. This recursive op-

eration where a complex structure at level n is decomposed into segments at level $n+1$ continues until every single arc that is part of a complex structure has been individually described.

Every entry in the data structure has a unique **Previous Level Index** which is used to link it to its ascendant entry at the previous level. Likewise, every segment of an entry in the data structure has a unique **Next Level Index** used to find its descendant at the next level. Entries at level 0 do not belong to other structures and have an invalid Previous Level Index (-1), and single arcs which do not have descendants have an invalid Next Level Index (-1) on its only segment.

An example of how to use the CRDS to store the chain C_1 depicted in Figure 6.4, is provided in Tables 6.1 and 6.2. In these tables, it is possible to see how C_1 has been stored at level 0 (vertex 0 represents the origin s and 9 is the destination d). At level 1, each of the segments that form C_1 is further described and, if necessary, decomposed at the following levels. The example shows how single arcs are just stored, while virtual arcs and chains are abstracted and described at the following levels. The structure is terminated at level 4, where the arcs belonging to C_3 are stored.

Because single arcs could become part of more than one complex structure (for example arc 4-5 which is part of two different Varcs at levels 1 and 3) it is necessary to implement a simple **Arc database** (Arc db) which contains the different locations of every single arc recorded in the Chain Routing data strcuture.

The Chain Routing data structure and its related Arc database show that this representation is indeed a very simple solution for storing the chains of a digraph. The following two subsections explain how the modified BFS algorithm searched for chains in a network and then recorded them in the CRDS.

Level 0	Number of Paths = 1		
	Path 0	Type = Chain	
	Chain length = 4	Segment count: 6	Prev Lev Idx = -1
	Segment 0	Arc = 0-4	Next Lev Idx = 0
	Segment 1	Arc = 0-2	Next Lev Idx = 1
	Segment 2	Arc = 0-9	Next Lev Idx = 2
	Segment 3	Arc = 4-2	Next Lev Idx = 3
	Segment 4	Arc = 4-9	Next Lev Idx = 4
	Segment 5	Arc = 2-9	Next Lev Idx = 5
Level 1	Number of Paths = 6		
	Path 0	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 0
	Segment 0	Arc = 0-4	Next Lev Idx = -1
	Path 1	Type = Varc	
	Chain length = 0	Segment count = 2	Prev Lev Idx = 0
	Segment 0	Arc = 0-1	Next Lev Idx = 0
	Segment 1	Arc = 1-2	Next Lev Idx = 1
	Path 2	Type = Varc	
	Chain length = 0	Segment count = 2	Prev Lev Idx = 0
	Segment 0	Arc = 0-3	Next Lev Idx = 2
	Segment 1	Arc = 3-9	Next Lev Idx = 3
	Path 3	Type = Varc	
	Chain length = 0	Segment count = 2	Prev Lev Idx = 0
	Segment 0	Arc = 4-5	Next Lev Idx = 4
	Segment 1	Arc = 5-2	Next Lev Idx = 5
	Path 4	Type = Varc	
	Chain length = 0	Segment count = 2	Prev Lev Idx = 0
	Segment 0	Arc = 4-6	Next Lev Idx = 6
	Segment 1	Arc = 6-9	Next Lev Idx = 7
	Path 5	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 0
	Segment 0	Arc = 2-9	Next Lev Idx = -1
Level 2	Number of Paths = 8		
	Path 0	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 1
	Segment 0	Arc = 0-1	Next Lev Idx = -1
	Path 1	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 1
	Segment 0	Arc = 1-2	Next Lev Idx = -1
	Path 2	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 2
	Segment 0	Arc = 0-3	Next Lev Idx = -1
	Path 3	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 2
	Segment 0	Arc = 3-9	Next Lev Idx = -1

Table 6.1: CRDS example for Figure 6.4 (Part 1)

	Path 4	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 3
	Segment 0	Arc = 4-5	Next Lev Idx = -1
	Path 5	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 3
	Segment 0	Arc = 5-2	Next Lev Idx = -1
	Path 6	Type = Chain	
	Chain length = 3	Segment count = 3	Prev Lev Idx = 4
	Segment 0	Arc = 4-7	Next Lev Idx = 0
	Segment 1	Arc = 4-6	Next Lev Idx = 1
	Segment 2	Arc = 7-6	Next Lev Idx = 2
	Path 7	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 4
	Segment 0	Arc = 6-9	Next Lev Idx = -1
Level 3	Number of Paths = 3		
	Path 0	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 6
	Segment 0	Arc = 4-7	Next Lev Idx = -1
	Path 1	Type = Varc	
	Chain length = 0	Segment count = 2	Prev Lev Idx = 6
	Segment 0	Arc = 4-5	Next Lev Idx = 0
	Segment 1	Arc = 5-6	Next Lev Idx = 1
	Path 2	Type = Chain	
	Chain length = 3	Segment count = 3	Prev Lev Idx = 6
	Segment 0	Arc = 7-8	Next Lev Idx = 2
	Segment 1	Arc = 7-6	Next Lev Idx = 3
	Segment 2	Arc = 8-6	Next Lev Idx = 4
Level 4	Number of Paths = 5		
	Path 0	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 1
	Segment 0	Arc = 4-5	Next Lev Idx = -1
	Path 1	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 1
	Segment 0	Arc = 5-6	Next Lev Idx = -1
	Path 2	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 2
	Segment 0	Arc = 7-8	Next Lev Idx = -1
	Path 3	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 2
	Segment 0	Arc = 7-6	Next Lev Idx = -1
	Path 4	Type = Arc	
	Chain length = 0	Segment count = 1	Prev Lev Idx = 2
	Segment 0	Arc = 8-6	Next Lev Idx = -1

Table 6.2: CRDS example for Figure 6.4 (Part 2)

6.2.2 Discovering chains in a digraph

The simplest way to identify chains in a digraph is by using the transitive relationship that must exist between the vertices that form the chain. Hence a vertex that can be reached through more than one path must posses a transitive relation with other vertices in this graph.

The ChainRtg program uses a modified version of the BFS algorithm (Algorithm 1). The input to this version is the adjacency matrix of an announcement digraph, just like the ones produced in Chapter 4. To keep track of the vertices that have been visited, this algorithm uses the *distance* array which will either contain a valid value, in which case the vertex has been visited before and a transitive relation exists, or an initial invalid value, which means the vertex is currently being visited for the first time.

When a vertex is visited for the first time it has to be added to the CRDS as either a single arc, or as part of a simple virtual arc (Section 6.1.2). On the other hand, vertices that have been visited before should be part of a chain and need to be further processed to determine the length and vertices that form this chain. This process will be further explained in subsection 6.2.3.

There is no output from Algorithm 1 because the main objective of this program is to store the topological information in the CRDS.

6.2.3 Adding chains to the data structure

Verifying and adding complete orders into the Chain Routing data structure (CRDS) is not a trivial task because each new link needs to be tested against the three possible scenarios described in section 6.1.3. The procedure responsible for performing this task is called *add_arc_to_Chain* (Algorithm 2) and its input is the *transitive arc* that has been discovered by the modified BFS program (Algorithm 1). In this section, this **input arc**

Algorithm 1: ChainRtg algorithm based on BFS

Input: The adjacency matrix of $D_{annc}\,(i) = (V, A)$
Output: No output needed from this algorithm

```
foreach v ∈ V do
    predecessor(v) ← nil;
    distance(v) ← nil;
end
distance(i) ← 0;
QUEUE ← i;
while QUEUE ≠ ∅ do
    x ← head of QUEUE (also delete x from QUEUE);
    foreach out-neighbour (y) of x do
        if distance(y) = nil then
            distance(y) ← distance(x) + 1;
            predecessor(y) ← x;
            Add y to the tail of QUEUE;
            if (y is the only out-neighbour of x) AND (x ≠ i) then
                Add a Varc to the data structure using arc xy and the arc formed by
                predecesor(x) and x;
            else
                Add a singe arc xy to the data structure;
            end
        else
            if predecesor(y) = nil then
                Arc xy is just a cycle back to the origin, ignore it;
            else
                Add a chain to the data structure using the transitive arc xy;
            end
        end
    end
end
```

is labelled $x_n y_n$ and, because vertex y_n has been visited before, the two vertices of this arc can be reached through at least a pair of different arcs, called the **predecessor arcs** of $x_n y_n$, which are labelled as $x_{n-1}x_n$ and $y_{n-1}y_n$.

In the most simple scenario $x_{n-1} = y_{n-1}$, which produces the simple 3-vertex chain $C(x_{n-1}, x_n, y_n)$ depicted in Figure 6.5 (A). However, more complex structures are possible in which any of the predecessor arcs could either be part of a Varc, such as in Figure 6.5 (C), or be part of a more complex structure that is not necessarily related to $C(x_{n-1}, x_n, y_n)$, for example Figures 6.5 (B) and (D). This means that both

Algorithm 2: Algorithm for procedure *add_arc_to_Chain*

Input: The start and end vertex of new arc $x_n y_n$
Output: Return **true** if the operation was succesful or **false** otherwise

```
x_{n-1} ← predecessor(x_n);                  // Usually, x_{n-1} = y_{n-1} and is also
y_{n-1} ← predecessor(y_n);                    // the source vertex of a chain
Xinst ← query the number of instances of x_{n-1}x_n in the Arc db;
Yinst ← query the number of instances of y_{n-1}y_n in the Arc db;
foreach Yinst do
    foreach Xinst do
        Xlevel ← query Arc db for level of arc x_{n-1}x_n;
        if Xlevel = 0 then
            x_{n-1}x_n is an Arc;
        else
            Xtype ← To what type of structure does x_{n-1}x_n belongs in the Arc db;
            switch Xtype do
                case Chain
                    x_{n-1}x_n is a Chain;
                end
                case Varc
                    x_{n-1}x_n is a Varc;
                end
                otherwise
                    return error;
                end
            end
        end
    end
end
```

predecessor arcs need to be tested against the network's already discovered topology in order to determine if it is feasible to define a simple 3-vertex chain that does not cause conflicts with other structures in the CRDS. Even then, the procedure needs to consider the possibility that $x_n y_n$ is actually an arc that allows to define a longer chain by combining two shorter chains, as exemplified in Figure 6.6.

Because each of the two predecessor arcs could be any of the three basic structures (chain, Varc or arc), there are nine different scenarios that could be tested while trying to define the correct position of the $x_n y_n$ arc in the CRDS. Algorithm 2 shows how procedure *add_arc_to_Chain* first checks the properties of predecessor arc $x_{n-1}x_n$ in an

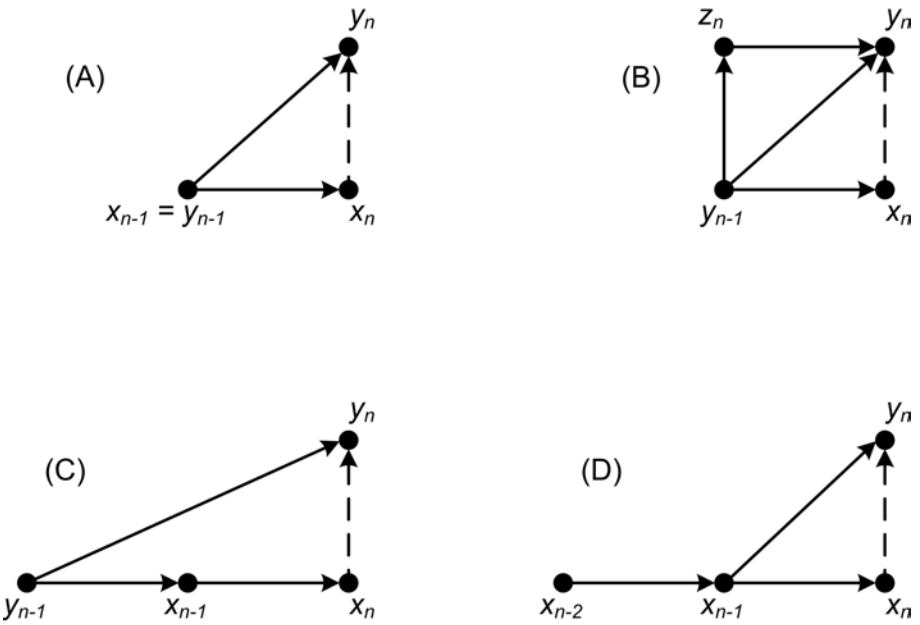


Figure 6.5: Some chain examples

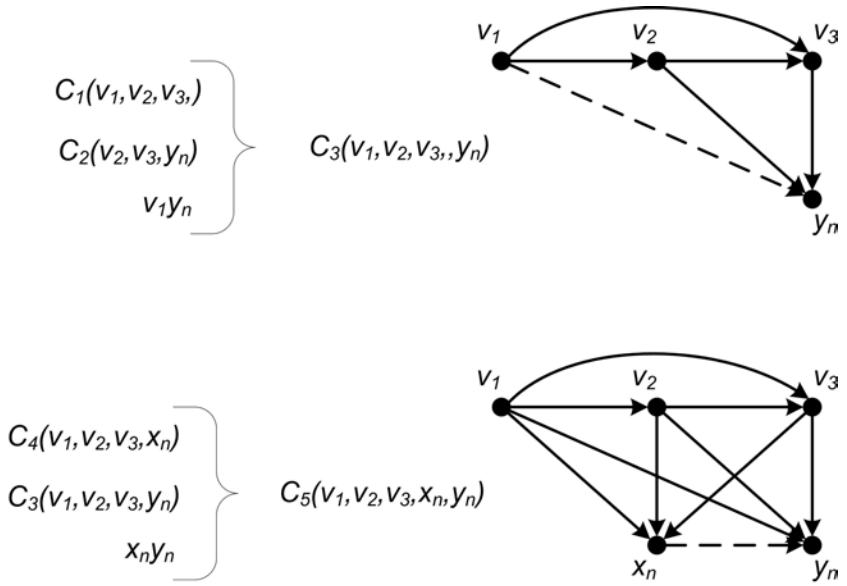


Figure 6.6: Combining two chains into a longer chain

attempt to correctly classify this arc as any of the Chain Routing's basic structures. The following subsections (6.2.3.1 to 6.2.3.3) explain the processing needed to determine where arc $x_n y_n$ should fit in the CRDS.

6.2.3.1 $x_{n-1}x_n$ is an arc

If the predecessor arc $x_{n-1}x_n$ is stored at the CRDS as only one arc that has not yet been included as part of a complex structure, then this means that this arc should have a unique instance in the CRDS at level 0. Hence determining that $x_{n-1}x_n$ is an arc is a trivial task.

Once it has been established that $x_{n-1}x_n$ is an arc, the following operations need to take place depending on which type of structure the predecessor arc $y_{n-1}y_n$ is:

invalid When $x_{n-1} \neq y_{n-1}$ it means that a more complex relation and structure may exist between the predecessor arcs. There are two options which need further analysis: either it is necessary to use Varcs to build the chain, as illustrated in Figure 6.5 (C); or it is necessary to create a **series of chains**, as defined in section 6.1.3.

arc This produces the chain $C(x_{n-1}, x_n, y_n)$, which is depicted in Figure 6.5 (A).

Varc Because y_{n-1} is the immediate predecessor of y_n, the Varc must be longer than what is needed to form the chain, similar to the example in Figure 6.5 (D), so the procedure needs to create another instance of $y_{n-1}y_n$ to build the $C(x_{n-1}, x_n, y_n)$ chain.

chain Since this algorithm's smallest chain has 3 vertices, the two other arcs ($x_{n-1}x_n$ and $x_n y_n$) cannot form a longer chain, because 3 new arcs are needed to form a chain with 4 vertices (Theorem 8). Hence this procedure needs to build a

$C(x_{n-1}, x_n, y_n)$ chain by creating another instance of $y_{n-1}y_n$. An example of this configuration is shown in Figure 6.5 (B).

6.2.3.2 $x_{n-1}x_n$ is part of a chain

When $x_{n-1}x_n$ is part of a chain there are two options that must be considered: First there is a possibility that, if $y_{n-1}y_n$ is also part of a chain, then the two chains could be combined to form a longer one. Then, it is also possible that $x_{n-1}x_n$ could be a segment of more than one chain, as it happens with arc $y_{n-1}y_n$ in Figure 6.5 (B). Therefore, it is necessary to test x_ny_n with as many combinations as possible, using a programming loop, until either a chain with more than 3 vertices can be defined or a suitable chain with Varcs can be found. If a complex structure cannot be found after completing this loop, then the algorithm will use the last basic structure tested to create a simple 3-vertex chain.

The following operations need to take place depending on what kind of basic structure predecessor arc $y_{n-1}y_n$ is:

invalid This is identical to what was described in subsection 6.2.3.1.

arc This is the case depicted in Figure 6.5 (B). If this is the last structure tested, define the 3-vertex chain $C(y_{n-1}, x_n, y_n)$.

Varc Similar to subsection 6.2.3.1. The procedure performs the same operation if this is the last structure tested.

chain First try to combine the two chains to which the predecessor arcs belong to form a longer chain. If this is not possible and if this is the last structure tested, define the 3-vertex chain $C(x_{n-1}, x_n, y_n)$.

6.2.3.3 $x_{n-1}x_n$ is part of a Varc

This case involves more operations because Varcs offer a larger number of options to create complex chains. Hence it is necessary to process every segment of the Varc against $y_{n-1}y_n$ in order to find if it is possible to build a chain. A loop in the program verifies segments of the Varc, starting from $x_{n-1}x_n$ and continuing to $x_{n-2}x_{n-1}$ and so on, until a suitable solution is found.

The following operations need to take place depending on the type of structure that predecessor arc $y_{n-1}y_n$ is:

chain/invalid Here both cases are the same because a more complex structure will be needed. The procedure tries to perform the same processing as for the invalid cases in subsections 6.2.3.1 and 6.2.3.2.

arc This is either the case depicted in Figure 6.5 (A), or the one in Figure 6.5 (B). In any case it is necessary to build a 3-vertex chain $C(y_{n-1}, x_n, y_n)$.

Varc This option will form a 3-vertex chain with either 1 or 2 Varcs.

Since simple Varcs do not offer path diversity, new instances of the same simple Varc may be used to define more than one chain; effectively using the Varc as if it were an arc. Conversely, a functional Varc, which was originally created to facilitate the definition of a chain, may not allow the definition of a new chain, therefore new instances of the arcs that formed the original functional Varc may be needed to define the new chain.

Up to this point, the main features of an algorithm that discovers chains in a network has been discussed. The code used to implement such algorithm has been included in Apendix C, and the results obtained from this implementation will be discussed shortly, but first it is necessary to describe its limitations.

6.2.4 Shortcomings of the algorithm used

The Chain Routing implementation described in this section is neither optimised nor complete. The main reason behind this assertion is that the ChainRtg program finishes after the modified BFS algorithm has completed its search for vertices in the digraph. But after this algorithm has concluded, there is a possibility to combine interlaced chains and nested structures to define longer chains. Therefore, in order to take advantage of the full path diversity of a network, it is necessary to implement post-processes which can discover the partial orders that could not be found before, because full topological information was not available. This development is beyond what can be achieved in the timeframe of this PhD programme, and it needs to be condsidered as future research.

There is also a need to explore if it is possible to apply or develop more efficient algorithms which would specialise on discovering chains, instead of nodes. This research activities are also beyond the scope of this thesis.

6.2.5 Results of the Chain Routing implementation

The software system described in subsections 6.2.1, 6.2.2 and 6.2.3 was implemented using four different C++ modules: BFSmod.cpp, Chainclass.cpp, Arcdbclass.cpp and queue.cpp. The code for these modules has been included in Appendix C. The final product is a program, named ChainRtg.exe, that uses a simple command-line interface to request the file that contains the adjacency matrix, $D_{annc}\,(i)$, and the position of the source vertex, i, in this matrix. The results of the analysis are stored in a text file, ChainRtg.out, which is similar to the listing shown in Tables 6.1 and 6.2, and contains all the available paths the program could find from vertex i to the other 44 destinations (vertices).

In practice, the ChainRtg program was able to process digraphs with 45 vertices in less than two seconds using a Dell Inspiron 6000 laptop, with a 2 GHz Intel Pentium processor, 2 GBytes of RAM and running the Microsoft Windows XP Home Edition operating system. Similar results were obtained when a Solaris workstation was used, thus the basic implementation of the Chain Routing algorithm is very fast for this matrix size (45×45). Moreover, the BFS algorithm used is known [50] to have a limited complexity of $O(n+m)$, where n is the number of vertices (order) and m is the number of arcs in the digraph, but the complexity of the functions that search chains within the BFS algorithm has the potential to become very large. Since the main objective of this numerical experiment is not to find the most efficient method to define chains, but that it is possible to do so in a digraph with path diversity (not a directed tree), no further analysis on the complexity of the implemented program will be provided in this thesis.

The digraphs for the Top-45 ASs (Chapter 4 and Appendix A) were processed using the ChainRtg program. Then, the results were analysed and the longest possible chain to each of the 44 announced ASs was recorded. The results of this analysis are displayed in Tables 6.3 and 6.4, and a description of what the entries in these tables mean follows:

number This is the height of the longest chain between the source AS and the announced AS. When the number is just 1, it indicates that there is only an arc between the ASs, but the announced AS is also part of a chain. This means that, although there is not much path diversity to the announced AS, it is still crucial to the connectivity of other ASs.

A This entry means that there is just an *arc* between the source AS and the announced AS. There is no further path diversity available.

B This means that there is a *bridge* between the source AS and the announced AS.

Examples of this configuration were presented in Figure 6.3. Although path diversity still exists, the prescence of the bridge indicates that the connectivity between the ASs is limited.

blank space Indicates that the AS does not receive an announcement; this is probably because these ASs are connected through an AS which was not included in the Top-45 list.

na The ChainRtg program failed to process the digraph because of a software error, and no results were obtained. These errors were caused by software bugs and can only be fixed by further improving the ChainRtg program.

By comparing Tables 6.3 and 6.4 with Tables 4.11 and 4.12, it is possible to see that there is a strong relation between the number of paths found in Chapter 4 and the height of the chains obtained by the ChainRtg program. This indicates that even a basic implementation of the Chain Routing framework is efficient at exploiting the path diversity of a digraph.

While the results for each digraph were being analysed, it was noticed that some chains conflicted with each other. An example of such a conflcit would be $C_1(a, b, c)$ and $C_2(a, c, b)$. This situation implies that, if Chain Routing is going to exploit the connectivity between a, b, and c, it will need to select and use only one of C_1 or C_2, otherwise cyclic behaviours could arise.

Since the height of a chain and the number of arc-disjoint paths are the same (Theorem 4), the results shown in Tables 6.3 and 6.4 demonstrate that, in most cases, there is a 2 arc-disjoint path to reach a destination. These tables also show how in some cases there is enough connectivity to build a 3 arc-disjoint, or even a 4 arc-disjoint path to a destination. This is proof that it is possible to use Chain Routing to increase the number of alternative paths and the resilience between source and destination.

Tables 6.3 and 6.4 also show that some AS can only use bridges (B) and arcs (A) to reach the other 44 destinations. This ASs were mostly relying on a better-connected AS to route to the rest of the network. This is probably what most ASs, which do not have many links to other ASs, experience in the Internet. This implies that ASs that are not highly connected will enjoy limited benefits from the Chain Routing framework.

Finally, Figure 6.7 shows the frequency of the chain-heights recorded in Tables 6.3 and 6.4. Since chains of height 1 and arcs (A) are similar, they are both counted under the same column (height=1) where arcs appear in darker colour. There were only 2 ocurrences of chains of height 4, and the most frequent chain, with a height of 2, had a count of 670.

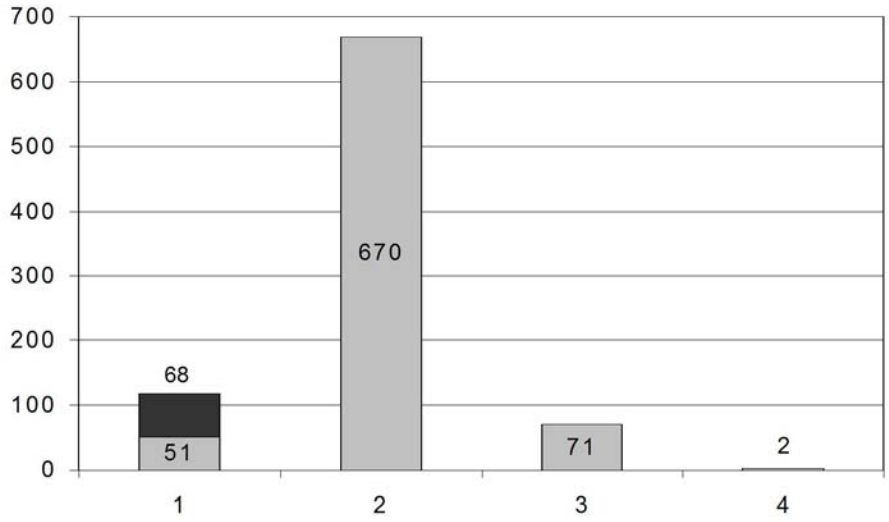


Figure 6.7: Number of chains per height value

Orig AS	AS1299	AS702	AS3303	AS1257	AS13237	AS8220	AS286	AS3257	AS1273	AS16150	AS8928	AS8342	AS5413	AS5511	AS12956	AS6762	AS15412	AS5400	AS20965	AS3292	AS3246	AS6805	AS8210	AS3301
AS1299	-	1	3	A	A	A	2	A	2	B	2	2	2	A	2	1	B	A	2	2	B	2	B	A
AS702	2	-	2	2	3	3	3	3	2	3	2	2	3	2	2	2	B	2	2	2	3	2	B	B
AS3303	2	1	-	2	2	4	2	2	2	3	3	3	3	3	2	2	2	2	2	2	2	3	2	B
AS1257	2	1	2	-	2	A	2	A	2	B	2	2	3	A	2	1	2	A	2	2	2	2	2	2
AS13237	2	2	2	2	-	2	3	2	2	2	2	3	2	1	2	B	2	2	2	2	2	2	1	B
AS8220	2	1	2	A	2	-	2	1	2	2	2	2	3	2	1	2	2	2	2	2	2	2	B	B
AS286	2	1	2	A	2	2	-	A	1	2	2	2	3		2	2	2	A	2	2	2	3	2	B
AS3257	2	2	2	2	2	2	3	-	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	B
AS1273	2	2	2	2	2	2	3	2	-	2	2	3	2	2	2	2	2	2	2	2	2	2	B	B
AS16150	1	2	3	B	2	3	3	2	B	-	2	2	2	2	B	B	2	2	B	2	3	2	2	2
AS8928	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na	na	na	na	na	na	na	na	na
AS8342	3	2	2	B	2	2	2	2	2	2	3	-	2	2	2	2	2	2	2	2	2	2	2	B
AS5413	2	2	2	2	2	2	3	2	2	2	2	3	-	1	2	2	2	2	B	2	B	2	B	B
AS5511	2	1	2	A	B	B	2	A	2	2	2	2	2	-	2	1	B	A	2	2	B	2	B	B
AS12956	2	3	3	3	B	2	3	2	2	B	2	2	2	2	-	2	B	2	2	2	B	2	B	B
AS6762	2	1	2	A	B		2	A	2	B	2	2	2	A	2	-	2	A	2	2	B	2	B	B
AS15412	B	B	3	2	2	2	2	2	2	2	2	2	2	2	2	2	-	2	B	2	3	2	2	B
AS5400	2	1	2	A	B	2	2	A	1	2	2	2	3	A	1	1	2	-	2	2	B	2	2	B
AS20965	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na
AS3292	2	2	3	2	2	2	3	2	2	3	2	2	3	A	2	2	2	2	2	-	2	3	2	2
AS3246	1	2	2	2	2	B	2	2	2	2	2	2	2	B	2	B	2	B	2	2	-	3	2	2
AS6805	2	B	3	B	2	A	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	-	2	B
AS8210		1	2	A	A		A	A	B	2	A	B	2		B		2	A	B	2	2	2	-	
AS3301	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	-
AS8434	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS5089	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS6878	2	2	2	2	2	2	2	2	2	2	2	B	2	B	2	1	2	2	B	2	2	2	2	B
AS31399	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS6667	2	2	3	2	2	2	3	2	2	2	2	2	2	2	2	2	2	A	2	3	2	2	2	B
AS1103	2	1	2	2	2	B	2	2	1	2	2	2	2	B	1	B	2	B	2	2	2	2	2	B
AS2603	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS680	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS3269	2	2	B	B	B	A	2	B	1	B	2	2	B	B	B	1	B	B	2	2	B	B	B	B
AS3215	B	B	B	B	B	2	B	B	B	B	1	B	B	1	B	B	B	B	B	B	B	B	B	B
AS9121	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS786	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS8404	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B	B
AS20485	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS9035	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS1267	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS3352	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS6830	3	3	4	3	2	2	3	2	2	2	2	2	2	3	3	2	2	2	2	2	2	2	2	B
AS3216	3	2	2	2	2	2	2	2	2	2	2	3	2	2	2	2	2	2	2	2	2	2	2	B
AS5568	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS3356	2	1	A		2	2	2	A	1	2	2	2	2	A	1	1	A	A	2	2	2	2	1	B

Table 6.3: Longest chain from an originating AS to other ASs (Part 1): A=arc, B=bridge, number=chain height, na=error

Orig AS	AS8434	AS5089	AS6878	AS31399	AS6667	AS1103	AS2603	AS680	AS3269	AS3215	AS9121	AS786	AS8404	AS20485	AS9035	AS1267	AS3352	AS6830	AS3216	AS5568	AS3356
AS1299	B	A	B	B	2	B	2	A	B	B	2	A	B	2	2	B	B	2	2	B	1
AS702	2	2	B	A	2	2	A	2	2	B	B	2	B	2	2	B	B	3	2	B	2
AS3303	2	2	B	B	2	2	B	2	2	B	B	2	B	2	2	B	B	2	2	B	3
AS1257	B		B	B	3	2	2	B	B	B	B	2	B	2	2	2	B	2	2	B	1
AS13237	2	2	B	B	2	2	2	2	B	B	B	2	B	2	2	B	B	2	2	B	2
AS8220	B	2	2	B	3	B	2	B	2	2	B	2	B	2	2	B	2	2	2	B	2
AS286	B	2	B	B	2	2	B	2	B		B	2	B	2	2	B	B	2	2	B	1
AS3257	B	2	B	B	3	2	B	2	2	B	B	B	B	2	2	2	2	2	2	B	1
AS1273	2	2	B	B	2	A	B	B	B	B	A	2	B	2	2	B	B	2	2	A	1
AS16150	3	2	B	B	2	2	2	B	B	B	B	2	B	2	2	B	B	2	2	B	1
AS8928	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS8342	2	2	B	B	2	B	B	B	B	B	B	2	B	2	2	B	B	2	2	B	2
AS5413	B	2	B	B	2	A	B	B	B	B	B	2	B	2	2	B	B	2	3	B	1
AS5511	2	2	B	B	3	B	2	B	B	A	B	2	B	2	2	B	B	2	2	B	1
AS12956	B	2	B	B	2	B	2	B	B	B	B	2	B	2	2	B	A	2	2	B	2
AS6762	B		B	B	2	B	2	B	A	B	2	B	B	2	3	B	B	2	2	B	1
AS15412	2	2	B	B	2	A	B	B	A	B	B	2	B	2	2	B	B	2	2	B	2
AS5400	2	A	B	A	2	B	A	2	B	B	B	2	B	2	2	B	2	2	2	B	1
AS20965	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS3292	2	2	B	B	2	2	2	2	B	B	B	2	B	2	2	B	B	2	2	B	2
AS3246	2	2	B	B	3	2	2	B	B	B	B	2	B	2	2	B	B	2	2	B	2
AS6805	1	2	A	B	3	A	B	2	2	B	B	B	B	3	2	B	B	2	2	B	2
AS8210	1	A	B	B	2	A	A				B	A	B	2	B	B	B	2	2	B	1
AS3301	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS8434	-	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS5089	na	-	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS6878	2	2	-	B	2	2	B	2	2	B	B	B	B	2	2	B	2	2	B	B	2
AS31399	na	na	na	-	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS6667	2	A	B	A	-	A	2	B	B	B	B	2	B	2	2	B	B	2	2	B	3
AS1103	1		B	B	2	-	2	2	B	B	B	2	B	2	2	B	B	2	2	B	2
AS2603	na	na	na	na	na	na	-	na	na	na	na	na	na	na	na	na	na	na	na	na	na
AS680	na	na	na	na	na	na	na	-	na	na	na	na	na	na	na	na	na	na	na	na	na
AS3269	B		B	B	2	B	B	B	-	B	B	B	B	2	2	2	B	B	2	B	1
AS3215	B		B	B	B	B	B	B	B	-	B	B	B	B	B	B	B	B	B	B	B
AS9121	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na	na	na	na	na	na
AS786	na	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na	na	na	na	na
AS8404	B	B	B	B	B	B	B	B	B	B	B	B	-	B	B	B	B	A	B	B	B
AS20485	na	na	na	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na	na	na
AS9035	na	na	na	na	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na	na
AS1267	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na	na
AS3352	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	-	na	na	na	na
AS6830	2	2	B	B	2	2	B	2	B	B	B	2	A	2	2	B	B	-	2	B	2
AS3216	2	2	B	B	2	2	B	B	B	B	B	2	B	2	2	B	B	2	-	2	3
AS5568	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	na	-	na
AS3356	2	2	B	B	2	B	2	2	B	B	2	2	B	2	2	B	2	2	2	B	-

Table 6.4: Longest chain from an originating AS to other ASs (Part 2): A=arc, B=bridge, number=chain height, na=error

6.3 Discussion

This final section discusses the benefits and drawbacks that Chain Routing possesses as a routing framework (subsections 6.3.1 and 6.3.2), and how it compares against the functionality provided by BGP (subsection 6.3.3).

6.3.1 Advantages of using Chain Routing

The main advantage of Chain Routing is that failure of a link or an AS does not require the protocol to reconverge to a new stable state because each AS knows, beforehand, the other alternative paths available to reach a destination. This property is provided by the two timescales, immediate and long-term, introduced in section 6.1.5.

A second advantage of Chain Routing is the possibility of implementing traffic engineering to avoid congestion when the traffic between a pair of adjacent ASs reaches a critical point, or to balance the traffic load between different routes. This is possible because, once a chain has been established, it is trivial to identify stable alternative paths and to use them concurrently. It must be noticed that load balancing between ASs has an impact on how traffic is routed within the AS. This is specially true when an AS spans a large geographical area, which causes the network administrators to limit the amount and type of traffic the network would carry for other ASs. Therefore, the implementation and full implications of traffic engineering is a topic that lies outside the scope of this thesis, but that still requires consideration for future research.

Finally, because partial orders are acyclic digraphs, it is possible to guarantee that failure of a link will not cause transient loops in the network. Also, because the chain structure must be coordinated by the ASs that form it, it is easier to determine the set of ASs thay could suffer from PRO and to detect these oscillatory behaviours in the network. This means that Chain Routing has the potential to become a very stable

routing protocol.

6.3.2 The cost of implementing Chain Routing

A very important factor that needs to be considered about Chain Routing is that all the ASs that are included in the vertex set of a chain, must agree to be used as intermediate nodes to reach a destination. Since vertices that are closer to the destination in the complete order will enjoy fewer benefits than those closer to the source, there will be disputes on the order that ASs will follow when defining a chain. This means that some human intervention and negotiation may be required in some cases to form chains. It also means that the traditional customer-provider model might need to be reconsidered and perhaps superseeded by another economic model which accommodates for chains.

In order to discover alternative paths to reach destinations, Chain Routing propagates its routing information through announcement digraphs (section 4.1), which possess more arcs than the currently used BGP digraph. The difference between these two digraphs represents the number of extra-messages needed to establish chains, and it depends on the policy restrictions that each AS applies to its neighbours. Hence, it is not possible to accurately quantify the additional messages needed to define chains in a network. Still, the numerical analysis described in this chapter suggests that many more destination messages would be travelling in the network if the announcement digraph is used, without employing the PPR.

Besides the additional messages produced by applying the announcement digraph, it is necessary to consider that coordination messages will be transmitted between the ASs that form chains in order to validate these structures. The details of the messages needed to establish such chains have not been finalised yet, but a tentative solution will need:

1. A message from the source to every intermediate node requesting to establish a chain using $n-2$ intermediate nodes.

2. A message from every intermediate node to the source accepting or rejecting to be part of such chain.

3. A message from the source to all the intermediate nodes that have accepted to be part of the chain confirming that the chain has been validated.

This means that every chain of length n will need up to $3(n-2)$ extra-messages to establish each chain. This number assumes that a source AS has been identified and that intermediate nodes agree to the chain proposed by the designated source. In practice, there may be conflicts when deciding who would be the source and the order of the intermediate vertices, and it is possible that such conflicts could only be solved by financial agreements or even human intervention.

Finally, it is also important to consider the size of the routing table for Chain Routing. As it was described in section 6.2.1, the CRDS contains a list of chains to different destinations. In order to avoid conflicts and cyclic behaviours when a chain is selected, the chain's intermediate nodes will also receive information using this same structure, even if there are better options to send packets to these nodes. In other words, a selected chain is a common solution for all the involved nodes, which may need to sacrifice some connectivity in order to guarantee network stability. This means that many destinations in the network will be contained in a chain or an abstracted structure of a chain, hence Chain Routing will need to use less chains than the total number of destinations. On the other hand, because Chain Routing would allow more paths to be announced between ASs, each router's CRDS will also store more topological information.

Although it is impossible to predict by how much the CRDS will increase the size of the routing table, the example in Tables 6.1 and 6.2 demonstrate how chains with nested

structures tend to need larger data structures, while chains which are higher may use memory more efficiently because they contain more intermediate nodes. In any case, each chain will have to record $n(n-1)/2$ segments (Theorem 2), without considering any nested structures which may exist.

6.3.3 Chain Routing vs. BGP

How does the potential functionality of Chain Routing compares to BGP's functionality? Once a chain has been defined, Chain Routing is a more stable solution than BGP, but it also requires more coordination between ASs. This means that Chain Routing is a more complex solution than BGP. On the other hand, because Chain Routing is based on partial orders, when a failure occurs, the chain is guaranteed to converge to a unique solution, and this is not the case for the current implementation of BGP.

Many network administrators have tried to configure BGP to perform traffic engineering by artificially creating two destinations that travel through two different paths and finally reach the same AS. Although this rudimentarily solution works, it demonstrates that BGP was not designed to support traffic engineering and it actually goes against the principles of route aggregation and CIDR prefixes. In contrast to BGP's lack of support for traffic engineering, the Chain Routing data structure can easily be used to distribute traffic amongst its different paths, as it was proposed by the maximal traffic distribution approach introduced in section 6.1.5.

A problem that derives from the increased complexity of Chain Routing is that route aggregation may be difficult once chains to destinations have been established. This happens because a new set of IP addresses would require further network coordination which may result in an intermediate AS rejecting the modification and impacting a previously defined chain. The solution to this problem is outside the scope of this

research, but it needs to be considered in any future research.

To better inform the judgement of whether the benefits of using Chain Routing surpass those of using BGP, first it is necessary to analyse the effects that this routing framework will produce in the Internet. This is further explored in the following chapter.

Chapter 7

The Implications of Chain Routing for the Internet

This chapter focuses on how the routing mechanism proposed in this thesis will interact with the rest of the Internet. Section 7.1 demonstrates how the four most important Internet instabilities could be either fixed or ameliorated by Chain Routing and provides the reasons behind these assertions. Then, section 7.2 extends this analysis by demonstrating the effect that temporal and topological orders have on the stability of communication networks. Sections 7.3 and 7.4 explore the two implementation options in which the new proposal could be adopted: (1) add Chain Routing's functionality to BGP; or (2) create a new routing protocol purely based in Chain Routing. Finally, section 7.5 discusses some of the factors that need to be considered before Chain Routing becomes a viable implementation in either form.

7.1 Internet instabilities vs. Chain Routing

The following subsections show how Chain Routing's increased resilience could help to either eliminate or reduce the impact of the four most important network instabilities of the Internet described in Chapters 2 and 3.

7.1.1 Chain Routing vs. persistent route oscillations

As it was asserted in section 6.3.1, Chain Routing may allow to find which ASs contribute to the development of PRO because the ASs that form a chain need to maintain a certain level of coordination. Unfortunately, the PRO problems described in Chapter 2 also have a temporal aspect, because of the path selection process which is continously executing in BGP. This means that in order to avoid developing cyclic behaviours within the chain, it is necessary to consider the dynamics of this system.

This subsection introduces a pair of mechanisms which dynamically avoid developing PROs in a chain. These mechanisms have been converted into rules which could easily be implemented. To demonstrate how these cyclic behaviours will be stopped from forming, examples have been provided after each rule's definition.

The first rule to avoid PRO in a chain is:

Rule 1. *Before accepting to become part of a chain, every AS needs to verify that the proposed chain does not create a cycle.*

This is similar to BGP's current functionality where ASs are constantly monitoring that cycles do not develop in their paths.

To demonstrate how Rule 1 stops PROs from developing, the topology described in Figure 7.1 will be used. This is the cannonical PRO example first introduced by Varadhan et al. [8], in which ASs a, b and c have different options to reach AS d, but

all of them prefer to use their longer path through the next neighbour over their shorter direct path. This preference is shown in Table 7.1. If AS a uses arcs $ab + bd$ to reach d, AS c prefers to use its direct path cd to reach d, but then AS b preferrs to use $bc + cd$ over its direct path which causes AS a to revert to its direct path ad, then AS c takes $ca + ad$, which in turn causes AS b to use again its direct path bd and AS a to adopt $ab + bd$, at which point it is clear that the sequence will repeat endlessly.

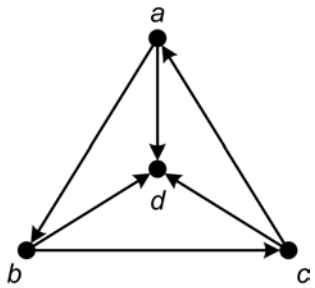


Figure 7.1: The cannonical PRO network by Varadhan et al.

AS	Avail. path	Pref.
a	a-b-d	1
	a-d	2
b	b-c-d	1
	b-d	2
c	c-a-d	1
	c-d	2

Table 7.1: Path preference for the cannonical PRO network

The cyclic behaviour decribed above will not develop if Chain Routing and Rule 1 are used. Since Chain Routing's main objective is to create chains, it may be possible that the first chain is defined from AS a as $C_a\,(a, b, d)$; this is a perfectly valid chain irrespective of which route a picks to reach d (either the direct ad route or the one throug b). Now AS c may try to use a to reach d, but because there is only one path from c to b ($ca + ab$), it needs to create a new chain $C_c\,(c, a, d)$ where segment ad is actually C_a. Finally, AS b will try to use c to reach d, but when b requests to create

chain $C_b\,(b, c, d)$, c will use Rule 1 and realise that b is already part of C_a (and C_c) and will reject the request to create C_b. Thus the PRO has been avoided.

The second rule to avoid PRO in a chain is:

Rule 2. *When a segment in a chain becomes unavailable and an alternative path needs to be used, it is safer to select paths that, because of the chain's topology, cannot route information through the unavailable segment. Once it has been confirmed that paths, which are adjacent to the failed segment, can still reach the destination, they may be reinstated as safe paths.*

This type of behaviour could help the network to reach a stable state faster because alternative paths that could be affected by the failure are not used.

How to use this rule is illustrated by the example provided by Griffin et al. [10] and reproduced in Figure 7.2. The route preference for this example is described in Table 7.2, where the X represents the fact that the originating AS does not really care which path the information takes as long as it goes through the counter clockwise neighbour and finishes in AS d. It is easy to demonstrate that when the link bd fails, BGP will develop a PRO similar to the one described in the previous subsection: as it is shown in Table 7.2, initially ASs c, e and f can easily send information to d through AS b, but when link bd fails, ASs c, e and f prefer to use their counter clockwise neighbour instead of the most direct route through AS a. This is the same cyclic behaviour described in Figure 7.1.

If Chain Routing is applied to this topology, the following three chains will be defined:

1. $C_1\,(c, f, b, d)$ with Varcs $ca + ad$ and $fa + ad$.

2. $C_2\,(e, c, b, d)$ with Varcs $ea + ad$ and $ca + ad$.

3. $C_3\,(f, e, b, d)$ with Varcs $fa + ad$ and $ea + ad$.

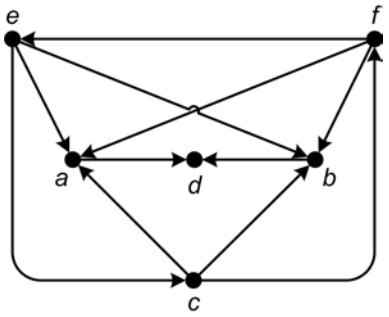


Figure 7.2: Another PRO case by Griffin et al.

AS	Avail. path	Pref.
c	c-b-d	1
	c-f-X-d	2
	c-a-d	3
e	e-b-d	1
	e-c-X-d	2
	e-a-d	3
f	f-b-d	1
	f-e-X-d	2
	f-a-d	3

Table 7.2: Path preference for the PRO network by Griffin et al.

So when link bd fails, ASs c, e and f will comply with Rule 2 by selecting paths that, because of the chain's topology, cannot route information through the faulty bd segment. In this example, the direct path of each chain would be the only route that agrees with Rule 2: AS c selects segment cd (Varc $ca + ad$) which cannot route information through segment bd.

The two previous rules mantain the acyclic topological characteristic in which Chain Routing is based. By assuring that no cycles develop when the topology of the network changes with the implementation of these rules, Chain Routing should provide enough robustness to avoid PROs in a network.

7.1.2 Chain Routing vs. delayed BGP convergence

The delayed BGP convergence problem, which was reviewed in section 3.1.2 could enjoy some improvements if the Chain Routing functionality is implemented. This problem is usually originated when a link or AS fails and other distant ASs believe they still have a valid path to reach the destination because the error messages have not reached them. While the error messages travel through the network and until it converges to a new stable state, transient loops may develop and these could, in turn, cause packets to be dropped.

Since Chain Routing is based on complete orders which are acyclic, it is possible to guarantee that transient loops will not develop. On the other hand, because Chain Routing does not control the dynamics of the system, it cannot assure that the network will reach faster convergence times, nor that information packets will not get lost while the network is in its transient state.

Fortunately, the proposed FESN solution by Chandrashekar et al. [42] (also reviewed in section 3.1.2) may be the ideal solution to this problem. Closer inspection of this proposal reveals that the FESN mechanism actually adds time labels to the messages, which is the same as creating a complete order in time. As will be illustrated in section 7.2, by combining complete orders in time and topology it is possible to provide a best compromise effort to eliminate transient instabilities in the Internet.

7.1.3 Chain Routing vs. congestion

Congestion, which is usually caused by excessive traffic over a single path or link, is usually avoided by implementing **traffic engineering**: this is the operation of distributing the data flows that a single path would normaly carry over other available paths.

As has already been discussed in sections 6.1.5, 6.3.1 and 6.3.3, Chain Routing pro-

vides a great opportunity to perform traffic engineering between all its different paths. In general, by applying Theorem 4, a chain of n vertices provides $n - 1$ arc-disjoint (and internally disjoint) paths that could be used to balance traffic between the source and the destination. Therefore, although Chain Routing cannot directly eliminate congestion in a network, it allows to implement better traffic administration mechanisms to avoid this problem.

7.1.4 Chain Routing vs. unnecessary route withdrawals

Unnecessary route withdrawals usually originate when congestion causes a path to be unavailable for a short period of time. This then causes the BGP mechanisms that prevent path oscillations to withdraw congested paths for a considerable period of time which, in turn, causes more congestion. This problem is amplified by the fact that ASs that are far from the failure receive multiple messages from ASs that are closer to the original congested path, which could cause the distant ASs to increase the amount of time that the failed path is unavailable. A full description of this pathology was first provided by Labovitz et al. [4] (section 2.2.1).

It has been discussed how Chain Routing indirectly helps to avoid congestion. However, if congestion cannot be avoided in a link, the chain structure could easily allow to temporarily set a path as unavailable while excessive traffic is diverted to alternative paths. Once the congestion has cleared, the path may be reinstated to carry traffic again. There are many traffic managment techniques that could be implemented to obtain the desired results, but these are outside the scope of this thesis. The importance of Chain Routing is that it allows more alternatives for traffic distribution than BGP.

The FESN modification by Chandrashekar et al. [42] may help to correctly diagnose and solve the unnecessary route withdrawal problem because it can determine when and

where the failure ocurred. Then, the Chain Routing framework could help to efficiently divert the information by exploiting the diversity of the chain structure.

In conclusion, although Chain Routing cannot solve all the instabilities observed in the Internet, it could become a powerful tool to develop a topological framework which is stable and exploits the path diversity that the Internet offers. A further analysis of how Chain Routing could contribute to achieve this stability is offered in the following section.

7.2 Orders and their effect on network stability

This section describes the interaction between events that follow a temporal order or a topological order. By analysing these interactions it is possible to demonstrate that, when either of these orders fails to exist, instabilities may arise in the network.

Figure 7.3 shows a small network in which there is an order in topology, but not in time. In this example, AS A suffers a failure at $t = 0$, but recovers from this failure at $t = 2$. AS B has a fast link to A and D, so it takes him a very short time to report A's failure and recovery. On the other hand, AS C is experiencing high volumes of traffic and cannot pass routing information to D as fast as B does. This causes a delay in the messages which corrupts the order at which they arrive in D. Therefore D sees that A failed at $t = 1$, recovered at $t = 3$, failed again at $t = 4$ and recovered at $t = 5$. This may cause other problems in larger networks as it was described by Mao et al. [15] in section 2.2.3.

In order to avoid this route flapping, it is necessary to implement an order in time. A simple way to achieve this is by not only reporting the occurrence of an event, but also the time at which the event happened. This is the exact same function of the FESN parameter proposed by Chandrashekar et al. [42]: If the messages in Figure 7.3 include

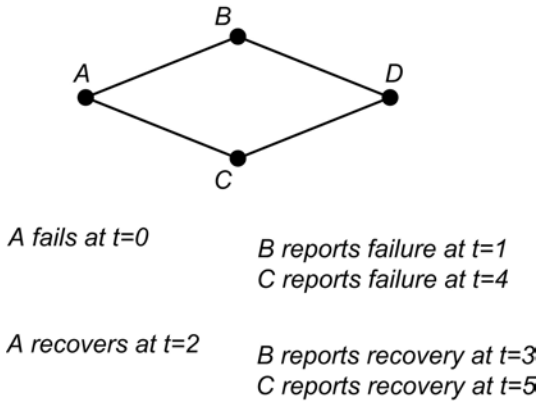


Figure 7.3: Network with no termporal order

the time at which the failure ($t = 0$) and the recovery ($t = 2$) happened, when AS D receives the delayed message from C at $t = 4$, it will be able to determine that C is reporting an event that happened before the recovery message received from B at $t = 3$. This example shows the importance of maintaining an order in time, even in a small network.

Conversely, Figure 7.4 shows a different network in which there is no clear topological order. In this example, AS A fails at $t = 0$, and ASs B and E acknowledge this failure at $t = 1$, but AS C takes more time to realise about the failure, which causes ASs B and D to believe that they could still send data to A through C. It is not until $t = 4$ that C reports that it cannot reach A, and that B and D will stop trying to send data through C.

The previous instability could be easily avoided if an order in topology was established before the failure occurred. For example, if a chain $C(A, B, C, D)$ was defined, when A fails and B reports the failure at $t = 1$, because C has a higher level than B, the Chain Routing system will forbid B to use C to reach A, thus the problem is avoided. On the other hand, if the chain $C(A, C, B, D)$ is defined, Rule 2 (section 7.1.1) will force AS D to use its direct link $Varc(ED + DA)$ to reach A, but because D also reported the failure at $t = 1$, D will be aware that A has become unavailable.

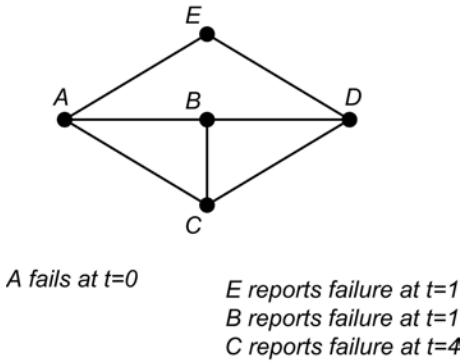


Figure 7.4: Network with no topological order

A final example of how orders in time and topology could be used to achieve greater network stability is shown in Figure 7.5. This figure shows a network that has been ordered using the chain $C_1(A, B, C, D)$. Now assume that AS C becomes unavailable at $t = 0$ and, because of congestion at B and E, AS D is the first one to realise that C is down. Then, without breaking any order, D can announce to A that C has become unavailable at $t = 0$ and therefore A could stop sending data to C, through E, before E announces the failure to A at $t > 0$. This example shows how orders are a powerful tool to maintain network stability.

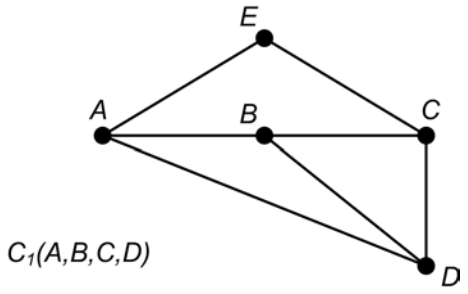


Figure 7.5: Network example

In conclusion, by combining orders in time and topology it is possible to maintain network stability, but when any of these two orders is broken, it is possible that instablity due to inaccuarte topological information will occur in the network. The following two sections address the options that may be selected for implementing Chain Routing

in the Internet: either as an enhancement to BGP, or as its replacement.

7.3 Chain Routing as an enhancement to BGP

This section analyses the most important features that must be considered before Chain Routing could be included as an enhancement to BGP. The feature which may come under the heaviest criticism is the proposal to eliminate BGP's PPR (Definition 2 in section 2.1). This BGP functionality does not allow transmitting alternative paths between ASs, as was discussed in section 2.1, which is one of the major improvements that Chain Routing offers. Although it may be argued that the PPR helps BGP to obtain stability, it is also possible to demonstrate that BGP develops the PRO problem because it needs to pick a preferred path which, due to policy conflicts, is sometimes simply not possible. It is also important to remember that the PPR hinders the natural implementation of traffic engineering in BGP. Taking these issues into account, it would be plausible to eliminate the PPR functionality from BGP to allow the best use of the Internet's path diversity.

Another important consideration is how to modify BGP's announcements to accomodate for Chain Routing's functionality. There are many posibilities to address this issue, but a basic implementation should at least include:

1. Besides BGP paths (AS_PATH), it will also be necessary to announce chains and their data structure.

2. Coordination messages that request forming a chain, and that either accept or reject such chain are necessary.

3. Coordination messages for announcing the failure of a segment in a chain.

4. Timers to define Chain Routing's long-term timescale (section 6.1.5), and the messages needed to establish new chains, when it has been determined that the network has suffered a permanent topological modification.

Some rules will also be needed for the translation of current BGP functionality to the new BGP version and vice versa. In summary, a new BGP version will be radically different to the current implementation, and its development and deployment would take considerable effort and time, but the gain would be a more resilient and stable routing protocol.

7.4 Chain Routing as a stand-alone routing protocol

Because the Chain Routing mechanism is not based on any previous routing protocol, the new set of messages that this protocol will implement needs careful design in order to efficiently transmit as much structural information about chains as possible with minimum delay and effort to the network. Since policies are an important feature of any exterior gateway protocol, it is necessary that topological information propagates through the network without the support of a central coordination node, and Chain Routing conforms to this requirement because it is a **decentralised routing algorithm**.

The possibilities of adopting Chain Routing as a new routing protocol are numerous. This section only intends to provide a list of the most notable features that should be considered if this path of development were to be taken.

Faster convergence time: As it was discussed in section 7.2, it is important to support complete orders in time as well as in topology.

Policies: Although the same set of policies that BGP supports could still be used for Chain Routing, it would be beneficial to reconsider them in order to implement

operations more suitable to the functionality provided by Chain Routing. For example, it could be desirable to design policies that allow ASs to negotiate their precedence in a particular chain.

Traffic engineering: This is an important feature that should be included in any implementation of Chain Routing. One important factor that also needs analysis is whether Chain Routing should support negotiating cost between ASs or routers in order to have access to more routing options. Similar features have already been proposed and explored for BGP by other authors, like the MIRO routing protocol by Xu and Rexford [45].

How this type of negotiation will be implemented and the amount of functionality that could be delegated to the algorithm, rather than the network administrator is a topic that guarantees to become very interesting.

Security: There have been calls to increase BGP's support of security as a routing protocol, so it seems reasonable to expect that future implementations will increase the support of this feature. Since the topic of network security is outside the scope of this research, the interested reader must refer to other sources, such as Pei et al. [76], for futher information in this area.

7.5 Final considerations

This section reviews some of the Chain Routing's factors which have not been covered before because they equally apply to the two implementation options discussed in the previous two sections.

First, it is important to consider that although the experiment described in section 6.2 offers and tests a basic implementation of a Chain Routing program, this imple-

mentation is neither complete nor it claims to be the best possible solution: as it was described in section 6.2.4, there is a possibility to build longer chains which would require to develop post-processing to recombine the chains discovered by the modified BFS algorithm. Therefore, other options should be considered before deciding to either refine the effort described in section 6.2 or to develop a new computer architecture which might be better at searching and building chains in a digraph, maybe through the use of artificial intelligence techniques or other specialised algorithms.

Another factor that has not been addressed by this research is the potential interactions between interior gateway protocols and Chain Routing. As it has been previously demonstrated by Labovitz et al. [7], such interactions have proved to be problematic for BGP. Also the PROs originated by conflicts with the MED parameter and described by Walton [29] could be considered as an interaction between the needs of BGP and the internal network's parameter setup. Therefore, it makes sense to learn from previous experiences and propose a better solution that allows network administrators to have safe and stable interactions between different types of routing protocols.

Finally, it is important to consider that, just like BGP becomes unstable when it cannot find a suitable set of paths to reach a destination, it is also possible that Chain Routing cannot find a suitable set of chains in a network. This may produce oscillations between competing alternative chains to a destination. Fortunately, this is a different kind of problem in which it is possible to stop or kill the oscillations without affecting the traffic, because every chain is already a robust solution. Still, it is necessary to develop mechanisms that eliminate the possibility of Chain Routing becoming unstable because it cannot find a definitive chain to reach a destination.

The following final chapter concludes this thesis by providing a summary of the contributions offered by this study and by suggesting further lines of research that could be explored in the future.

Chapter 8

Conclusions

This final chapter concludes this thesis by analysing the research activities performed, and some topics which were not covered but that may need to be considered in the future (section 8.1). Then, section 8.2 states the main results produced by this study. Finally, section 8.3 describes the steps that could be taken to continue the work started by this research.

8.1 Summary

This study started with the premise that the main cause of instability in the Internet is the imperfect topological information that the routers possess about the current state of this network. For that reason, the main objective of this research was to find if it is possible to design a routing protocol which is resilient to imperfect topological information.

Then, a numerical experiment which analysed the connectivity of a section of the Internet was performed. Such analysis considered the policies that had been applied between the ASs, but it was not restricted by the single best path rule which prevails

in BGP. The results of this experiment demonstrated that the Internet possesses unused path diversity which could be used to increase its resilience.

By analysing the structural topology of the Internet after policies have been applied, a series of alternatives were considered, and it was determined that a routing protocol based on acyclic digraphs could provide the best approach to obtain stability in this network. The proposed solution, termed Chain Routing, offers two advantages over the current BGP algorithm: it avoids the occurrence of transient loops in the network, and it allows easy implementation of traffic managment.

Still, the Chain Routing framework proposed in this thesis is a theoretical solution that would need further empirical development and testing. This research has only laid down the foundations of a new routing scheme, and its final implementation was never part of the scope of this study. There are also many characteristics that were not sufficiently addressed while developing the Chain Routing framework. Therefore each one of the following topics would need to be considered while elaborating its final implementation:

- Influence and implementation of policies in Chain Routing.
- Fast growth and shrinkage of chains.
- Influence of Chain Routing in the economic model of the network.
- Mechanisms that could allow route aggregation and scalability of the network.

On the other hand, it is also possible that the application of the Chain Routing framework could be extended to other systems which can be modelled as a digraph that needs to maintain its connectivity and could apply the mechanisms developed for Chain Routing. Examples of this type of system are: overlay networks which neet to maintain certain degree of logical connectivity; or distributing traffic in a city in order

to avoid congestion or if an accident has caused a street to be closed; or how to reroute trains that need to reach a destination when the original track has become unavailable.

A summary of the contributions of this research is provided in the following section.

8.2 Contributions of this study

The first research finding derived from the analysis that the connectivity of the ASs forming the Internet is greater than what was previously thought [17, 20]. Our finding converges with other studies which were specifically studying the Internet topology and did more comprehensive experiments and analyses [21].

Another contribution is the proposal that networks which announce their destinations according to a defined set of policies, such as the Internet, can be described by using digraphs, and that an efficient and organised way to represent how destinations propagate through different paths could be achieved by using the mathematical concept of a complete order.

A numerical experiment demonstrated that it is possible to establish complete orders in Internet-like networks, and although the limits to which this functionality can be taken are still not known, it was established that the rudimentary software used was efficient at exploiting the network's path diversity. Still, there is a need to understand and characterise the chain structure of real networks much more exhaustively, before and after policies have been applied. However, this is a topic which could not be further explored because of time constraints.

Finally, the conditions under which a network maintains its stability were explored, and it was determined that by applying complete orders in two realms, temporal and topological, it could be possible to obtain a highly stable routing protocol which is more resilient to failures than the current BGP implementation. This validates the assertion

that the topological complete order presented in this thesis, Chain Routing, has the potential to become a stable solution for routing in the Internet.

8.3 Moving forward

This final section outlines the steps required to move forward the concepts and contributions started by this study in further research projects.

The first step would be to explore alternative algorithms in order to determine which one is the most appropriate for implementing Chain Routing because, as it was stated in sections 6.2.4 and 7.5, the prototype implementation done for this research has not been determined to be necessarily the best possible approach. Then, in order to explore different options, it would be desirable to have at least two alternatives on how to implement such algorithm in a routing protocol. These options should be detailed enough so they could be implemented in a network simulator, like Opnet, ns-2 or SSFNet, and tested in large realistic Internet topologies. It would be very helpful if the results of these simulations could be compared against the functionality provided by the current BGP implementation, paying particular attention to the following characteristics:

- Algorithm convergence times.
- Traffic engineering (reliability under intense traffic).
- Resilience to network failures.
- Controlled transient behaviour.

Because is impossible to anticipate all the effects a new routing protocol could produce in a real environment, if a considerable enhancement over BGP's functionality can be achieved through the simulations, it would be necessary to implement the best

solution in a real network for further testing, and to gain empirical evidence of its usefulness.

It is expected that standards could be developed and proposed simultaneously with the research and test activities described above, in case it is decided that deployment of Chain Routing over the Internet is a worthwhile option.

References

[1] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)." RFC 4271, Jan. 2006.

[2] V. Paxson, "End-to-end routing behavior in the internet," in *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, vol. 26 of *Computer Communication Review*, pp. 25–38, ACM Press, Aug. 1996.

[3] R. Govindan and A. Reddy, "An analysis of internet inter-domain topology and route stability," in *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 2, pp. 850–857, Apr. 1997.

[4] C. Labovitz, G. R. Malan, and F. Jahanian, "Internet routing instability," in *SIGCOMM '97: Proceedings of the ACM SIGCOMM '97 conference on Applications, technologies, architectures, and protocols for computer communication*, no. 5, pp. 115–126, ACM Press, Sep. 1997.

[5] G. R. Malan and F. Jahanian, "An extensible probe architecture for network protocol performance measurement," in *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, no. 4, pp. 215–227, ACM Press, Sep. 1998.

[6] C. Labovitz, G. R. Malan, and F. Jahanian, "Origins of internet routing instability," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, pp. 218–226, Mar. 1999.

[7] C. Labovitz, A. Ahuja, and F. Jahanian, "Experimental study of internet stability and backbone failures," in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pp. 278–285, Jun. 1999.

[8] K. Varadhan, R. Govindan, and D. Estrin, "Persistent route oscillations in inter-domain routing," tech. rep., Department of Computer Science, University of Southern California, Feb. 1996.

[9] R. Govindan, C. Alaettinoglu, G. Eddy, D. Kessens, S. Kumar, and W.-S. Lee, "An architecture for stable, analyzable internet routing," *IEEE Network*, vol. 13, pp. 29–35, Jan. 1999.

[10] T. G. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 277–288, ACM Press, Sep. 1999.

[11] L. Gao and J. Rexford, "Stable internet routing without global coordination," in *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, no. 6, pp. 307–317, ACM Press, Jun. 2000.

[12] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed internet routing convergence," in *SIGCOMM '00: Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, no. 3, pp. 175 – 187, ACM Press, Sep. 2000.

[13] C. Labovitz, A. Ahuja, R. Wattenhofer, and S. Venkatachary, "The impact of internet policy and topology on delayed routing convergence," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1, pp. 537–546, Apr. 2001.

[14] T. G. Griffin and B. J. Premore, "An experimental analysis of BGP convergence time," in *Ninth International Conference on Network Protocols (ICNP'01)*, pp. 53–61, Nov. 2001.

[15] Z. M. Mao, R. Govindan, G. Varghese, and R. H. Katz, "Route flap damping exacerbates internet routing convergence," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 221–233, ACM Press, Aug. 2002.

[16] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang, "An analysis of BGP multiple origin AS (MOAS) conflicts," in *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 31–35, ACM, Nov. 2001.

[17] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 251–262, ACM Press, Sep. 1999.

[18] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of Modern Physics*, vol. 74, pp. 47–97, Jan. 2002.

[19] H. Tangmunarunkit, R. Govindan, S. Jamin, S. Shenker, and W. Willinger, "Network topology generators: degree-based vs. structural," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 147–159, ACM, Aug. 2002.

[20] P. Mahadevan, D. Krioukov, M. Fomenkov, X. Dimitropoulos, k c claffy, and A. Vahdat, "The internet AS-level topology: Three data sources and one definitive metric," *SIGCOMM Comput. Commun. Rev.*, vol. 36, pp. 17–26, Jan. 2006.

[21] R. Oliveira, D. Pei, W. Willinger, B. Zhang, and L. Zhang, "In search of the elusive ground truth: The nternet's AS-level connectivity structure," in *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 217–228, ACM, Jun. 2008.

[22] A. Dhamdhere and C. Dovrolis, "Ten years in the evolution of the internet ecosystem," in *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pp. 183–196, ACM, Oct. 2008.

[23] D. McPherson and K. Patel, "Experience with the BGP-4 protocol." RFC 4277, Jan. 2006.

[24] Y. Rekhter and T. Li, "A Border Gateway Protocol 4 (BGP-4)." RFC 1771, Mar. 1995.

[25] R. Zhang and M. Bartell, *BGP design and implementation*. Cisco Press, Dec. 2003.

[26] B. J. Premore, *An Analysis of Convergence Properties of the Border Gateway Protocol Using Discrete Event Simulation*. PhD thesis, Dartmouth College, May 2003.

[27] C. Villamizar, R. Chandra, and R. Govindan, "BGP Route Flap Damping." RFC 2439, Nov. 1998.

[28] K.-W. Chin, "On the characteristics of BGP multiple origin AS conflicts," in *Telecommunication Networks and Applications Conference, 2007. ATNAC 2007. Australasian*, pp. 157–162, Dec. 2007.

[29] D. Walton, "BGP MED Oscillation," in *NANOG 21*, NANOG, Feb. 2001.

[30] D. McPherson, V. Gill, D. Walton, and A. Retana, "Border Gateway Protocol (BGP) Persistent Route Oscillation Condition." RFC 3345, Aug. 2002.

[31] T. G. Griffin and G. Wilfong, "Analysis of the MED oscillation problem in BGP," in *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on*, pp. 90–99, Nov. 2002.

[32] V. J. Bono, "7007 explanation and apology." `http://www.merit.edu/mail.archives/nanog/1997-04/msg00444.html`, Apr. 1997. NANOG mailing list.

[33] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "Policy disputes in path-vector protocols," in *Network Protocols, 1999. (ICNP '99) Proceedings. Seventh International Conference on*, pp. 21–30, Nov. 1999.

[34] T. G. Griffin and G. Wilfong, "A safe path vector protocol," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 490–499, Mar. 2000.

[35] T. G. Griffin, F. B. Shepherd, and G. Wilfong, "The stable paths problem and interdomain routing," *IEEE/ACM Transactions on Networking*, vol. 10, pp. 232–243, Apr. 2002.

[36] J. A. Cobb and R. Musunuri, "Enforcing convergence in inter-domain routing," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, vol. 3, pp. 1353–1358, Nov. 2004.

[37] C. T. Ee, B.-G. Chun, K. Lakshminarayanan, V. Ramachandran, and S. Shenker, "Resolving inter-domain policy disputes," in *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 157–168, ACM, Aug. 2007.

[38] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang, "Improving BGP convergence through consistency assertions," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 902–911, Jun. 2002.

[39] A. Bremler-Barr, Y. Afek, and S. Schwarz, "Improved BGP convergence via ghost flushing," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, vol. 2, pp. 927–937, Mar. 2003.

[40] B. Zhang, D. Massey, and L. Zhang, "Destination reachability and BGP convergence time," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, vol. 3, pp. 1383–1389, Dec. 2004.

[41] D. Pei, M. Azuma, D. Massey, and L. Zhang, "BGP-RCN: improving BGP convergence through root cause notification," *Computer Networks*, vol. 48, pp. 175–194, Jun. 2005.

[42] J. Chandrashekar, Z. Duan, Z.-L. Zhang, and J. Krasky, "Limiting path exploration in BGP," in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE*

Computer and Communications Societies. Proceedings IEEE, vol. 4, pp. 2337–2348, Mar. 2005.

[43] W. Lijun, W. Jianping, and X. Ke, "Modified flap damping mechanism to improve inter-domain routing convergence," *Computer Communications*, vol. 30, pp. 1588–1599, May 2007.

[44] L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica, "HLP: A next generation inter-domain routing protocol," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, Computer Communication Review, pp. 13–24, ACM Press, Aug. 2005.

[45] W. Xu and J. Rexford, "MIRO: Multi-path interdomain routing," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 171–182, ACM Press, Sep. 2006.

[46] A. D. Jaggard and V. Ramachandran, "Robust path-vector routing despite inconsistent route preferences," in *Network Protocols, 2006. ICNP '06. Proceedings of the 2006 14th IEEE International Conference on*, pp. 270–279, Nov. 2006.

[47] CAIDA, "Skitter." `http://www.caida.org/tools/measurement/skitter/`, Jan. 2008.

[48] University of Oregon, "Route Views Project." `http://www.routeviews.org/`, Jan. 2008.

[49] RIPE NCC, "RIPE Database." `http://www.ripe.net/db/index.html`, Oct. 2007.

[50] J. Bang-Jensen and G. Gutin, *Digraphs: Theory, Algorithms and Applications*. Springer, 2002.

[51] F. Harary, R. Z. Norman, and D. Cartwritght, *Structural models: An introduction to the theory of directed graphs*. John Wiley & Sons, Inc., 1965.

[52] "The Cooperative Association for Internet Data Analysis (CAIDA)." `http://www.caida.org/home/`, Jan. 2008.

[53] CAIDA, "Autonomous System (AS) ranking." `http://as-rank.caida.org/`, Jan. 2008.

[54] Robtex, "AS2856 BT UK AS BTnet UK Regional Network." `http://www.robtex.com/as/as2856.html`, Jan. 2008.

[55] T. Richardson, "BT's Infonet acquisition completed," *The Register*, Feb. 2005.

[56] BBC News, "BT secures $965m US telecoms deal." http://news.bbc.co.uk/1/hi/business/3991707.stm, Nov. 2004.

[57] "Open peering initiative." http://www.openpeering.nl/, Jan. 2008.

[58] "Real Time Network (RETN)." http://www.retn.net/en/, Jan. 2008.

[59] BBC News, "NTL seals $6bn Telewest takeover." http://news.bbc.co.uk/1/hi/business/4304008.stm, Oct. 2005.

[60] BBC News, "Virgin Mobile to be bought by NTL." http://news.bbc.co.uk/1/hi/business/4874694.stm, Apr. 2006.

[61] "RIPE Network Coordination Center (NCC)." http://www.ripe.net/, Oct. 2007.

[62] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra, "Routing Policy Specification Language (RPSL)." RFC 2622, June 1999.

[63] RIPE NCC, "Routing Information Service (RIS)." http://www.ripe.net/np/ris/, Jan. 2008.

[64] RIPE NCC, "The libbgpdump installation files." http://www.ris.ripe.net/source/, Jan. 2008.

[65] University of Oregon, "Advanced Network Technology Center (ANTC)." http://antc.uoregon.edu/, Jan. 2008.

[66] CAIDA, "The asadj2graph.pl Perl script." http://sk-aslinks.caida.org/tools/, Jan. 2008.

[67] R. Sedgewick, *Algorithms in Java, Part 5: Graph Algorithms*, vol. 2. Addison-Wesley, 3rd ed., 2004.

[68] R. Shankar, *Principles of quantum mechanics*. Plenum Press, 2nd edition ed., 1994.

[69] W. Schnyder, "Planar graphs and poset dimension," *Order*, vol. 5, pp. 323–343, Dec. 1989.

[70] P. C. Fishburn, *Interval orders and interval graphs: a study of partially ordered sets*. Interscience series in discrete mathematics, Wiley, 1985.

[71] J. F. Kurose and K. W. Ross, *Computer Networking: a top-down approach featuring the Internet*. Addison Wesley, 3rd ed., 2004.

[72] W. D. Grover and D. Stamatelakis, "Cycle-oriented distributed preconfiguration: ring-like speed with mesh-like capacity for self-planning network restoration," in *Communications, 1998. ICC 98. Conference Record.1998 IEEE International Conference on*, vol. 1, pp. 537–543, Jun. 1998.

[73] C. C. Constantinou, A. S. Stepanenko, T. N. Arvanitis, K. J. Baughan, and B. Liu, *Handbook of applied algorithms: solving scientific, engineering, and practical problems*, ch. Resilient recursive routing in communication netowks, pp. 485–507. Wiley-Interscience, 2008.

[74] V. D. Park and M. S. Corson, "A highly adaptive distributed routing algorithm for mobile wireless networks," in *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, vol. 3, pp. 1405–1413, Apr. 1997.

[75] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs, "R-BGP: Staying connected in a connected world," in *4th USENIX Symposium on Networked Systems Design & Implementation*, pp. 341–354, USENIX, Apr. 2007.

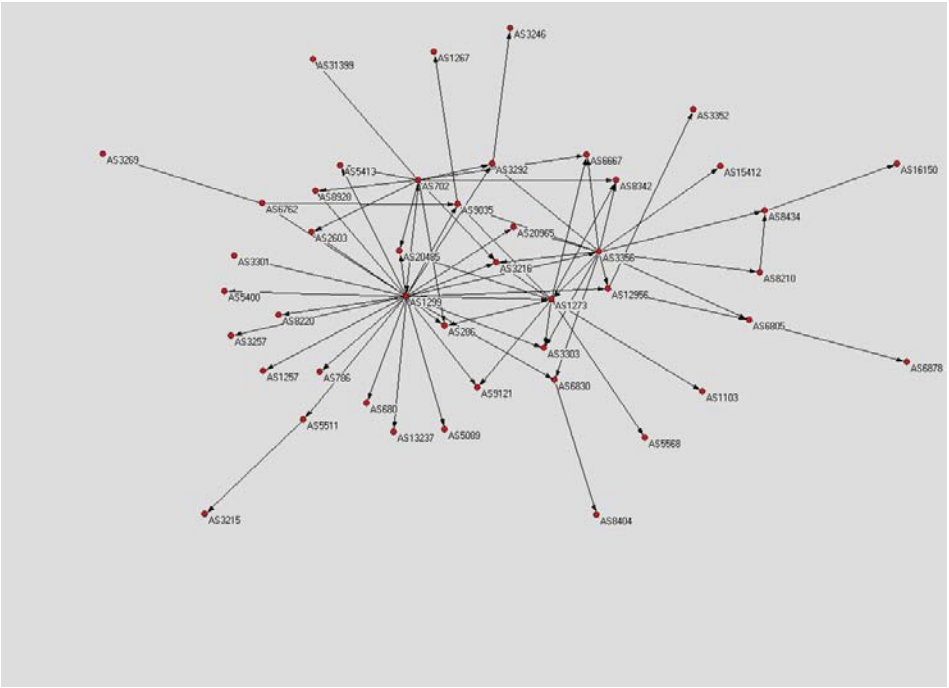
[76] D. Pei, L. Zhang, and D. Massey, "A framework for resilient internet routing protocols," *IEEE Network*, vol. 18, pp. 5–12, Mar. 2004.

Appendix A

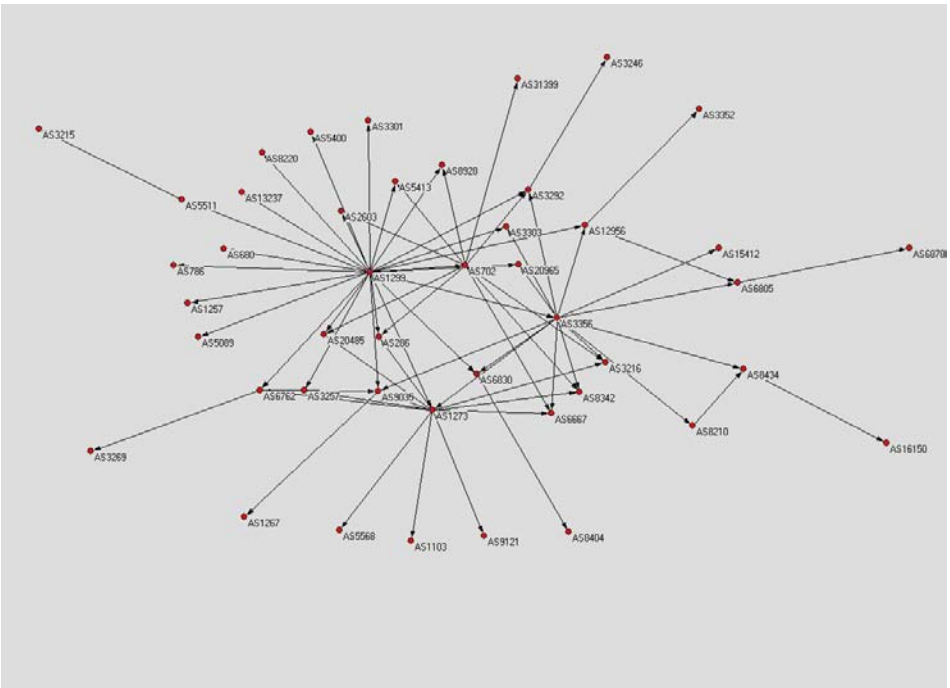
Announcement Digraphs for the Top-45 ASs

The following pages show the announcement digraphs, $D_{annc}(i)$, for the top-45 ASs in table 4.3. There are two graphs for each AS, one for each day at which routing data was collected:

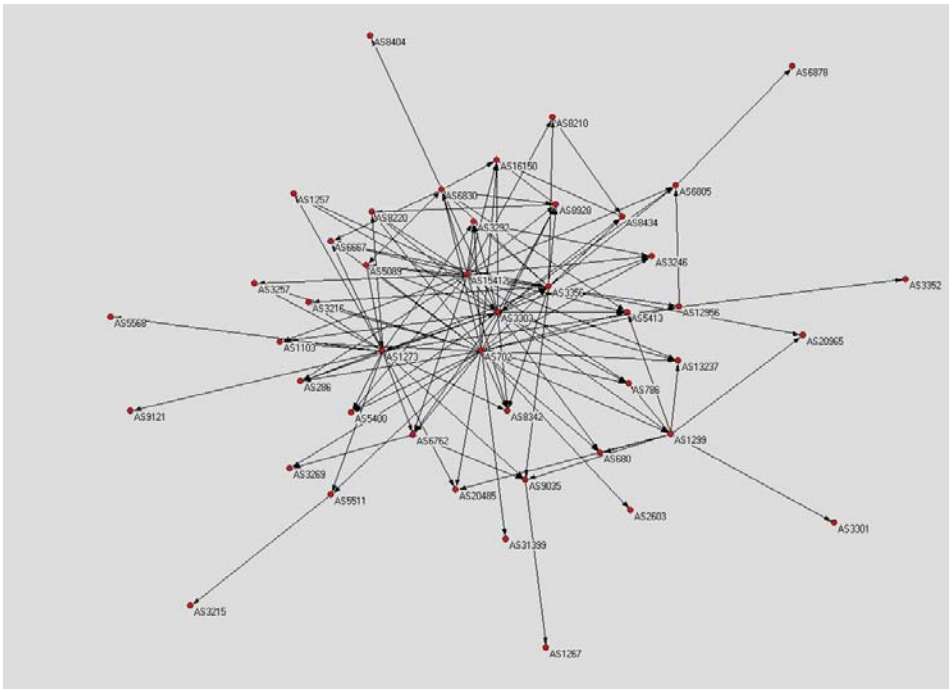
- November 29, 2007
- December 19, 2007



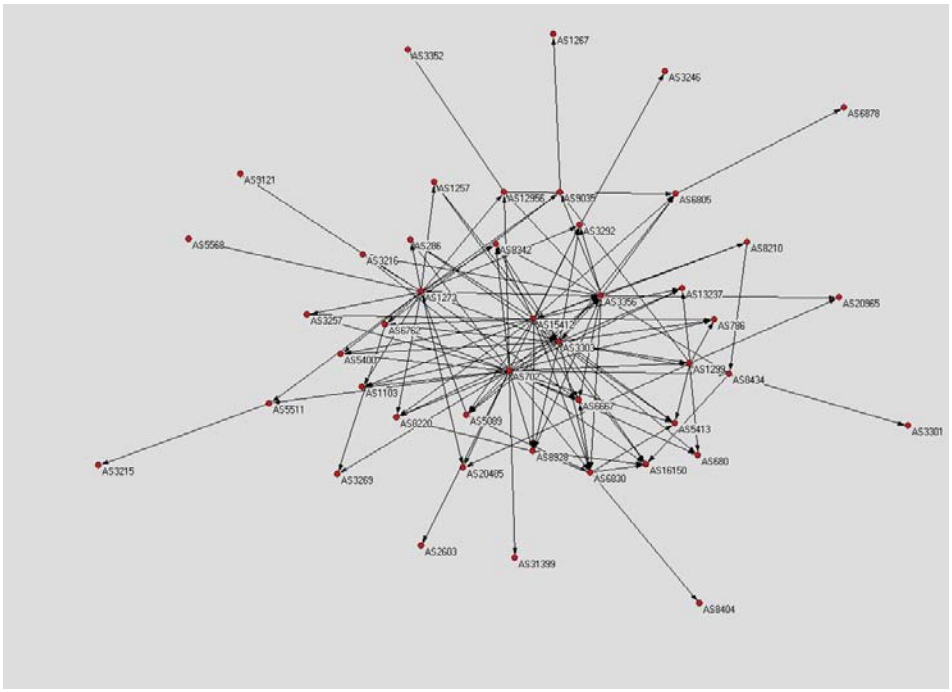
AS1299 - November 29, 2007



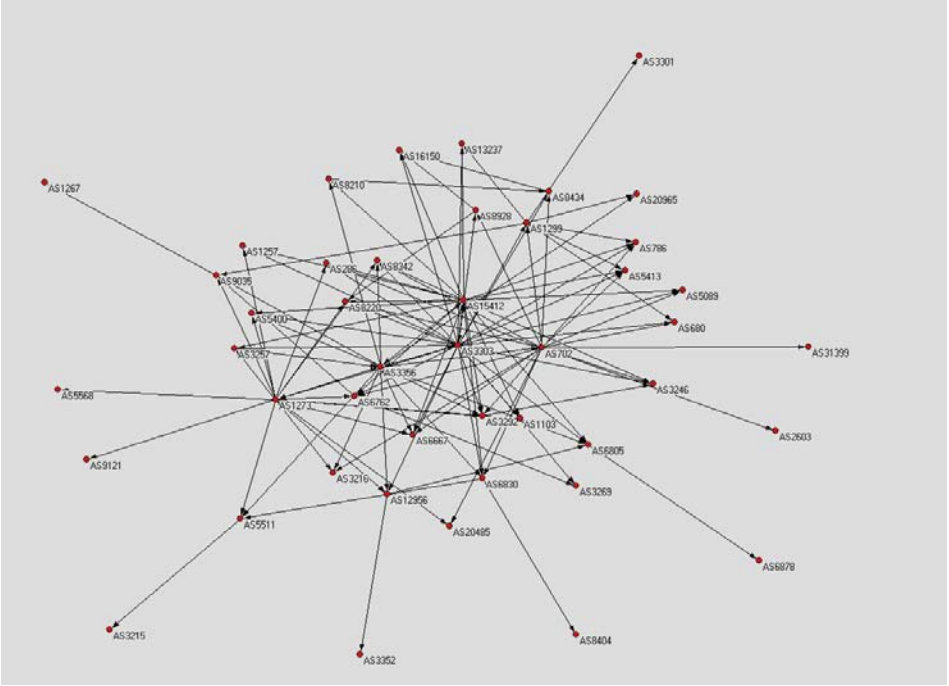
AS1299 - December 19, 2007



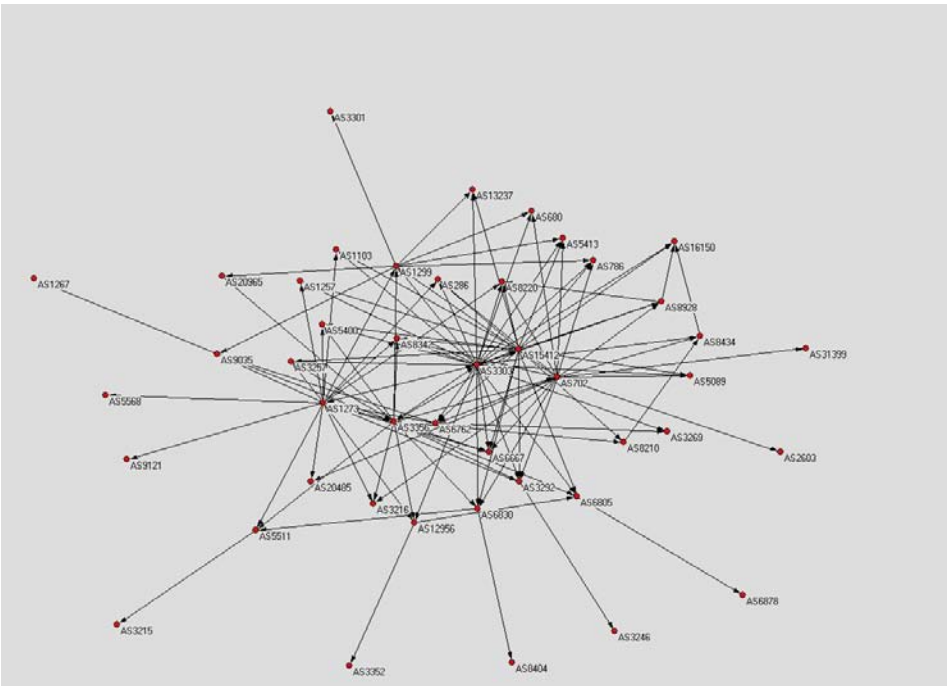
AS702 - November 29, 2007



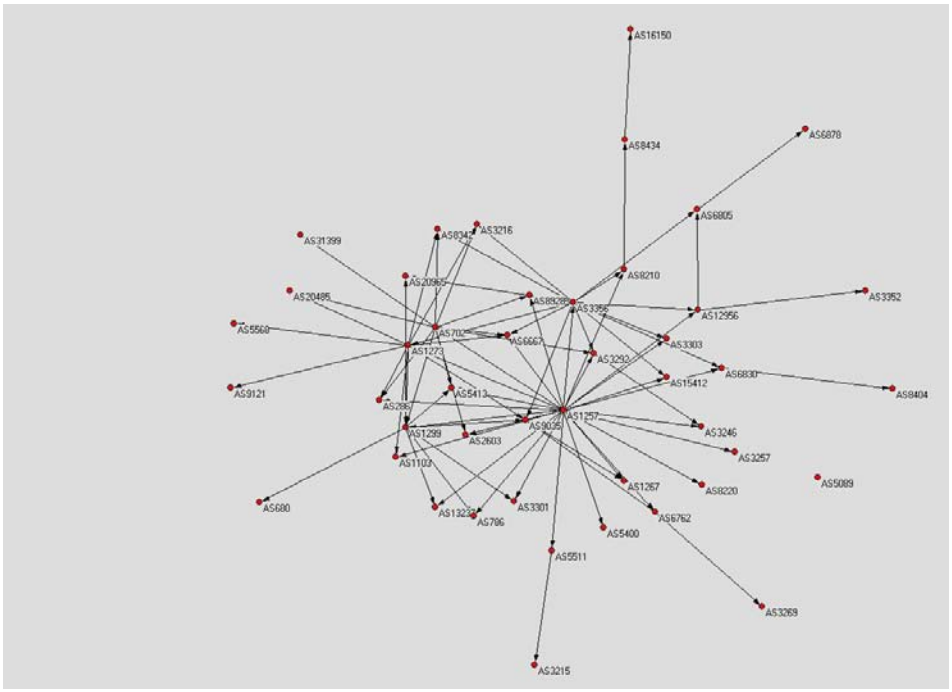
AS702 - December 19, 2007



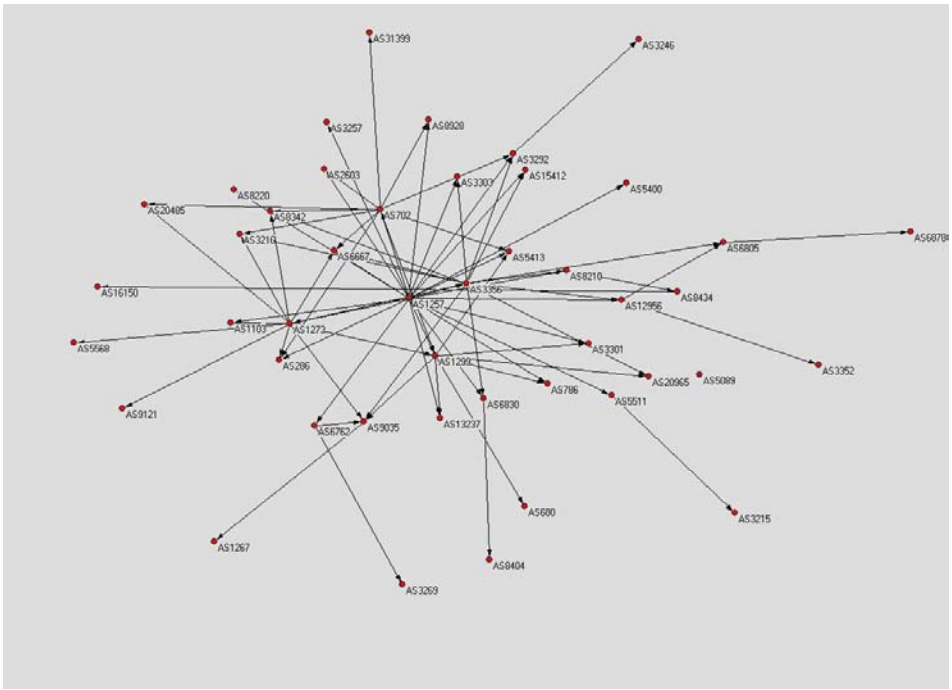
AS3303 - November 29, 2007



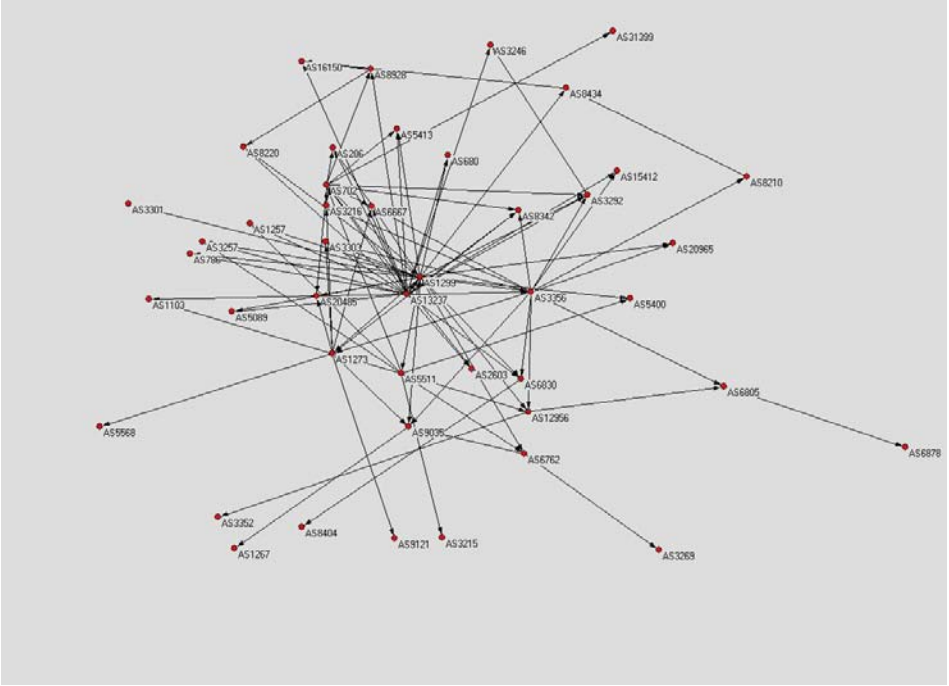
AS3303 - December 19, 2007



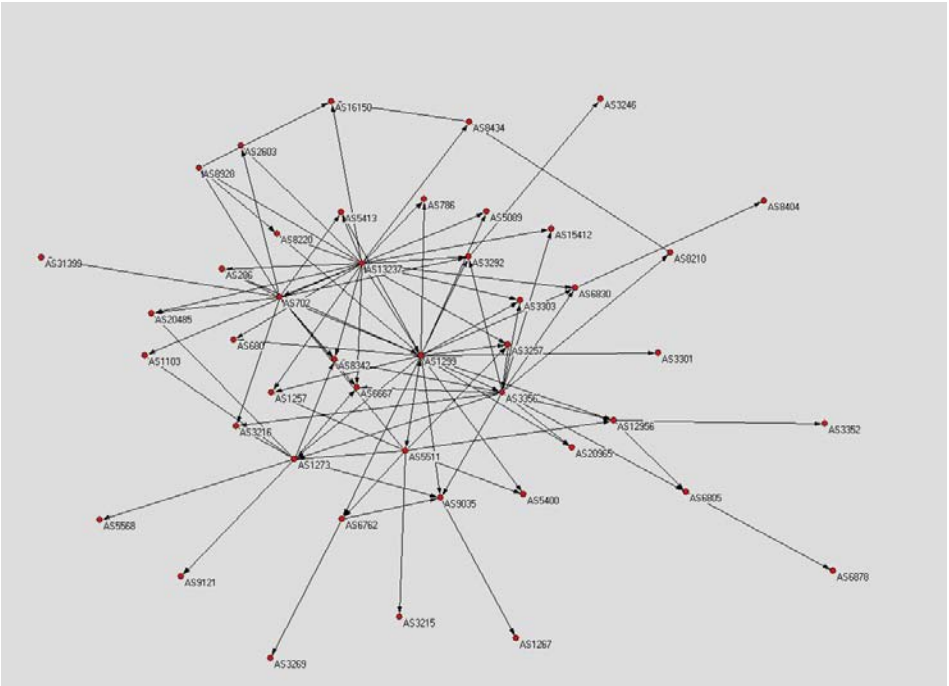
AS1257 - November 29, 2007



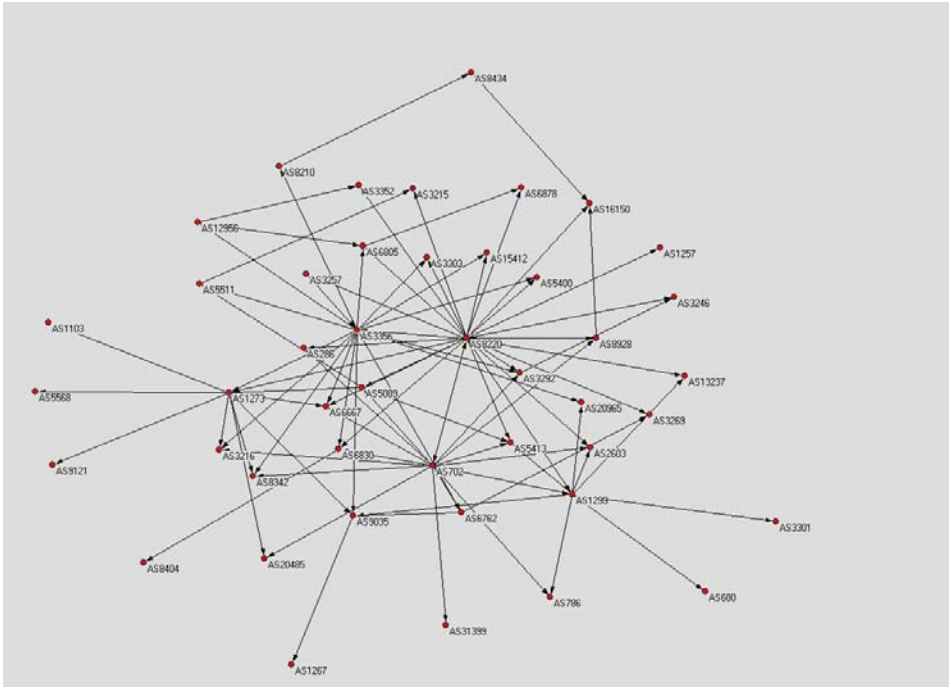
AS1257 - December 19, 2007



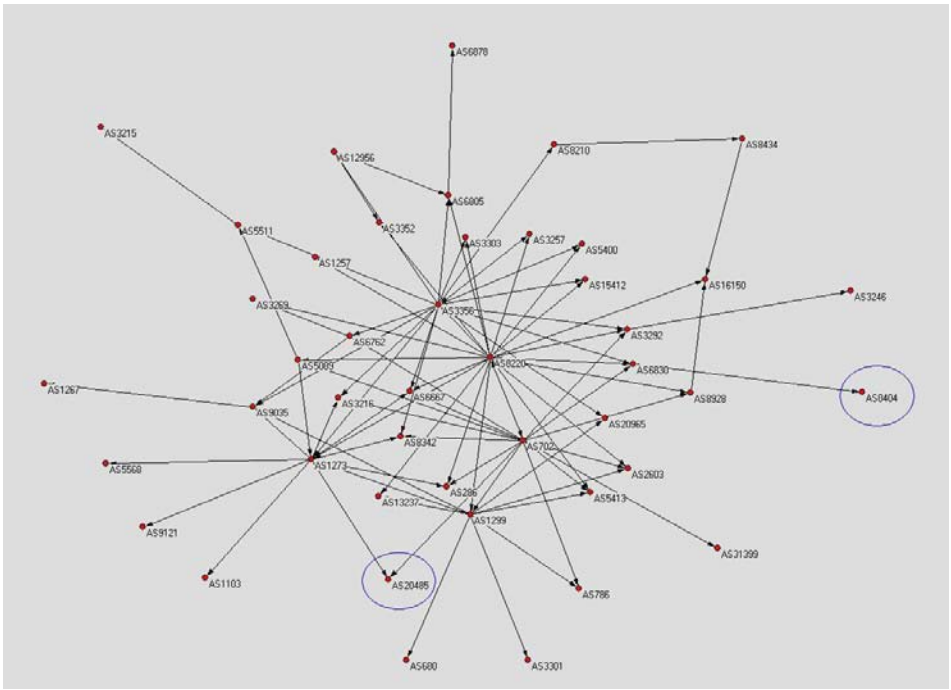
AS13237 - November 29, 2007



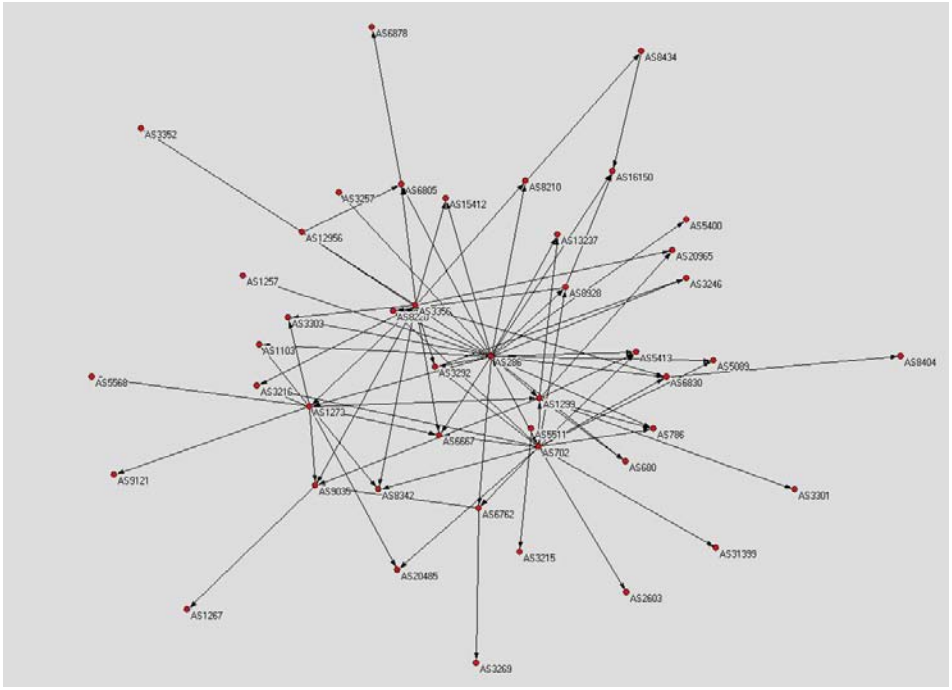
AS13237 - December 19, 2007



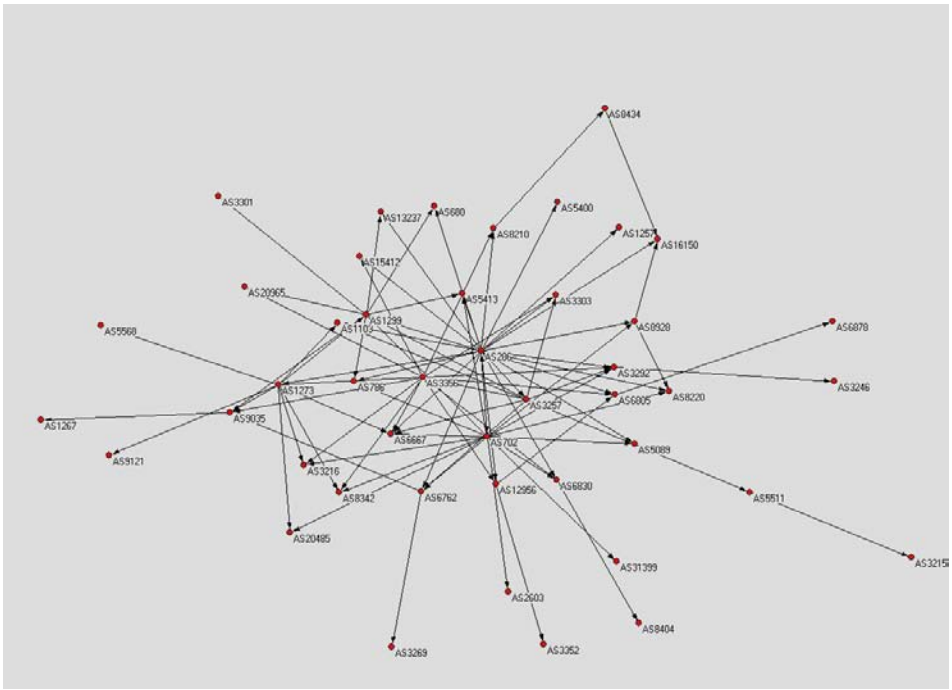
AS8220 - November 29, 2007



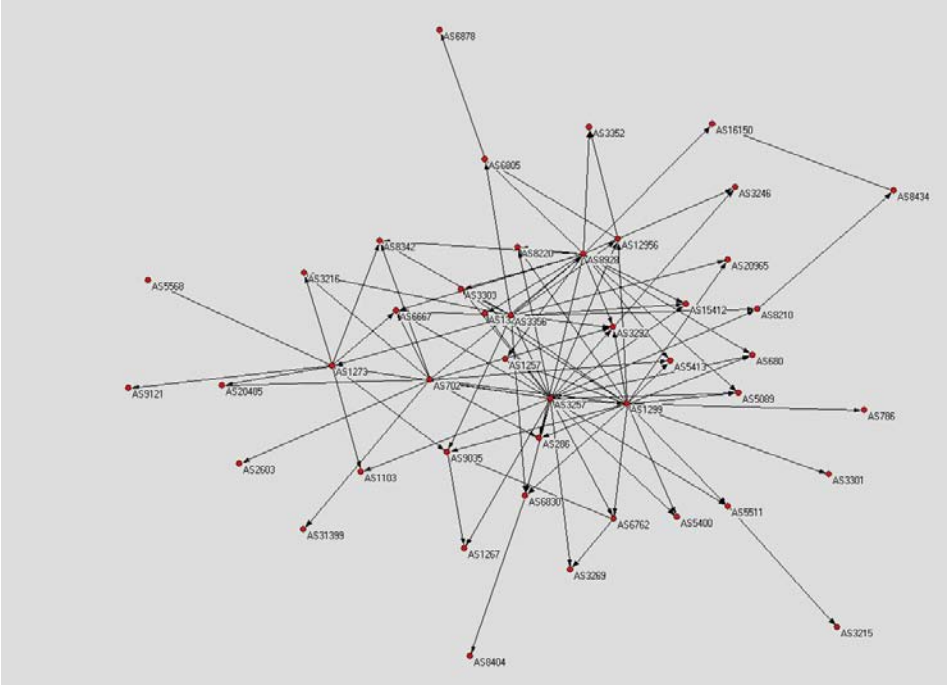
AS8220 - December 19, 2007



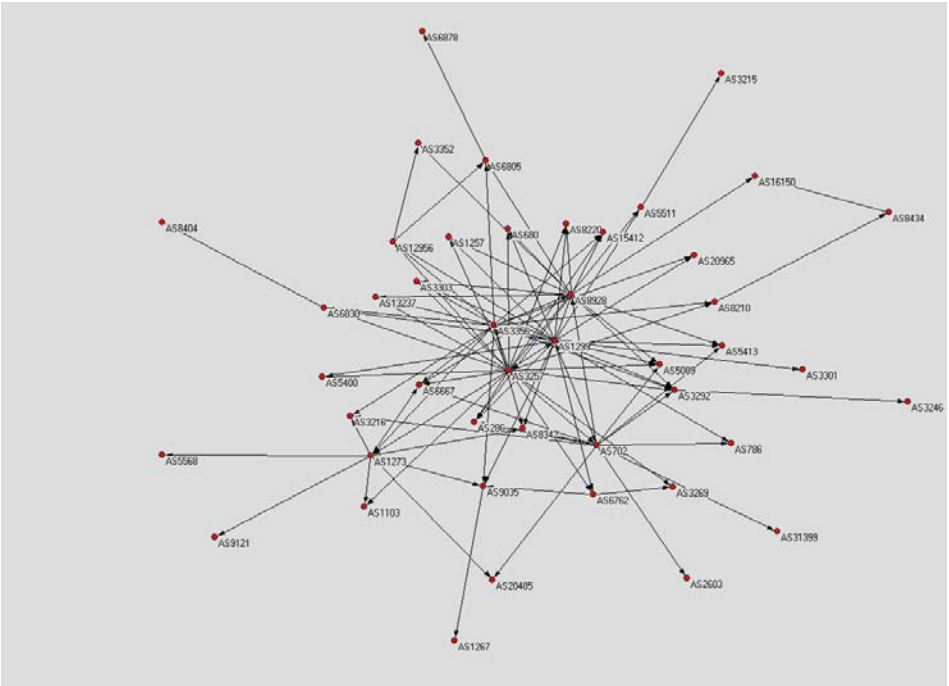
AS286 - November 29, 2007



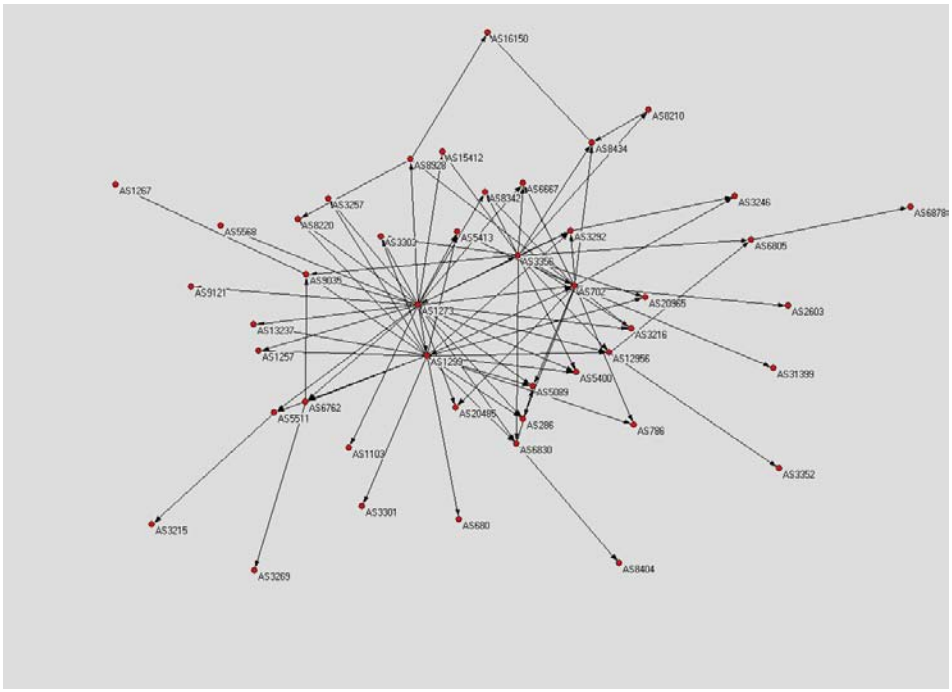
AS286 - December 19, 2007



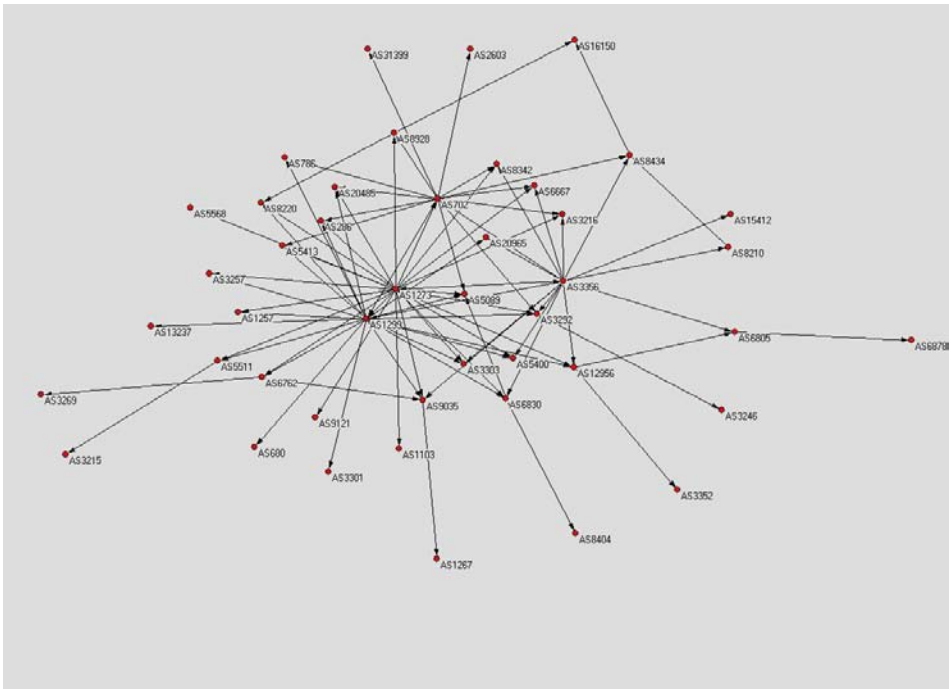
AS3257 - November 29, 2007



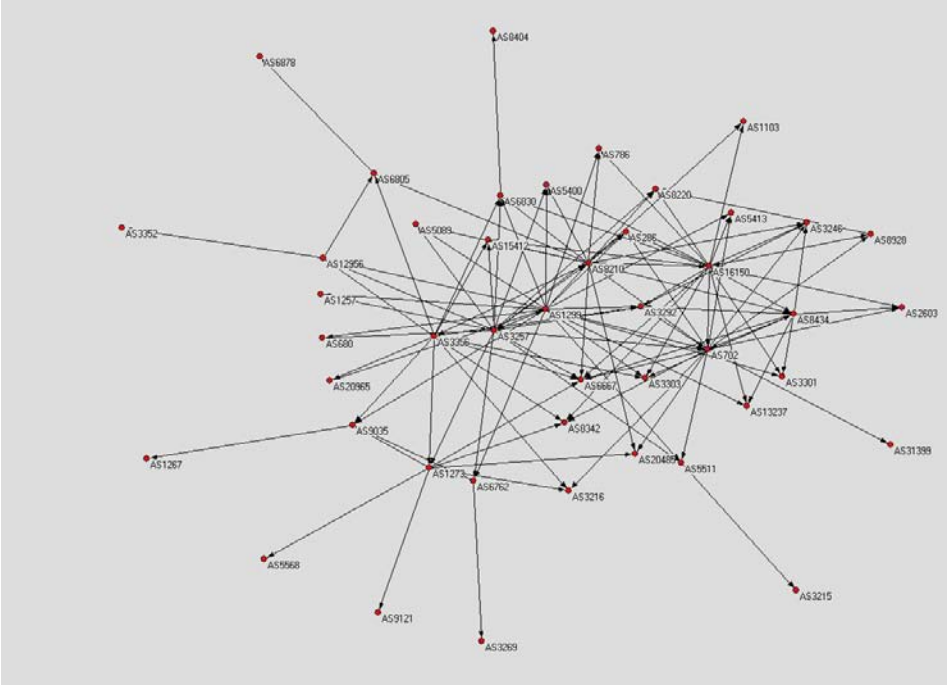
AS3257 - December 19, 2007



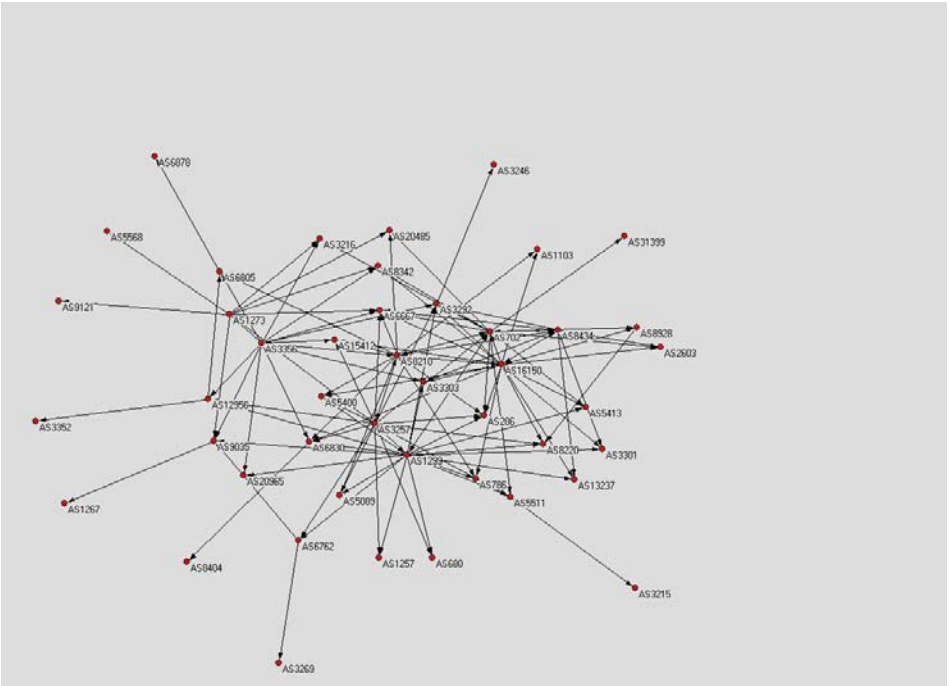
AS1273 - November 29, 2007



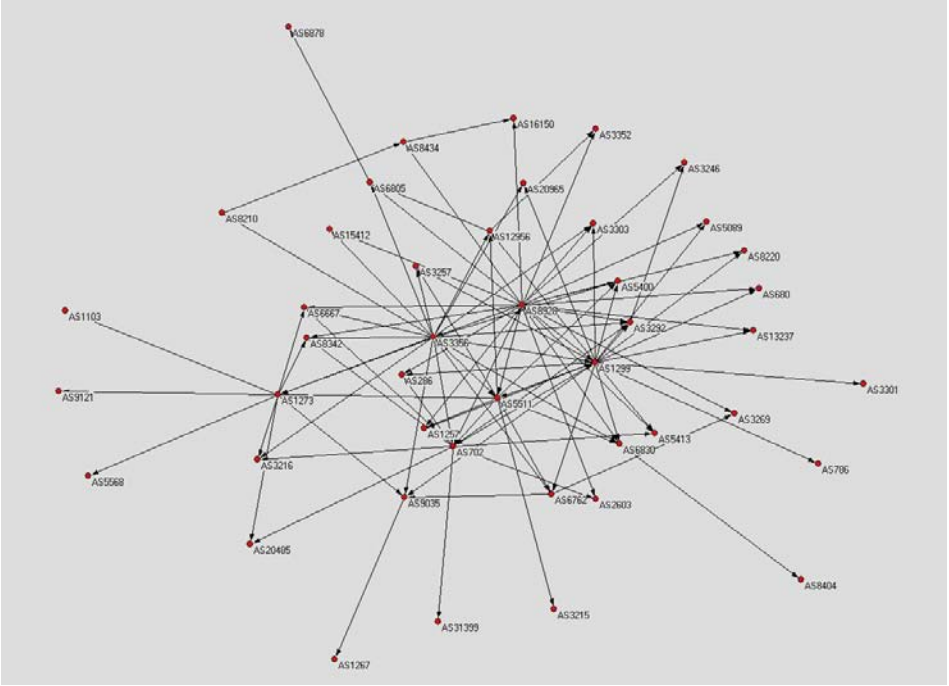
AS1273 - December 19, 2007



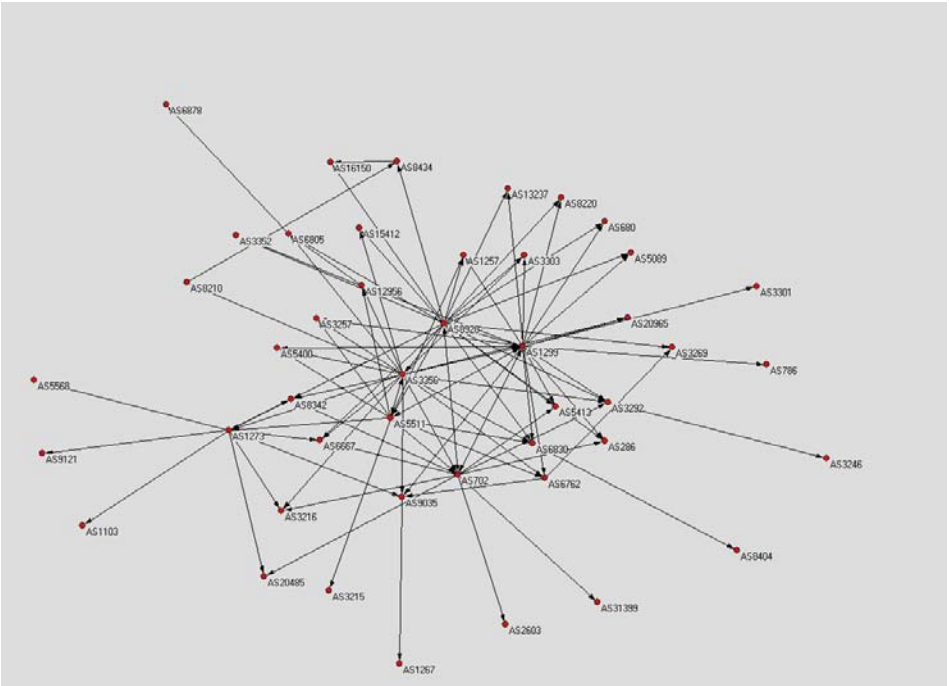
AS16150 - November 29, 2007



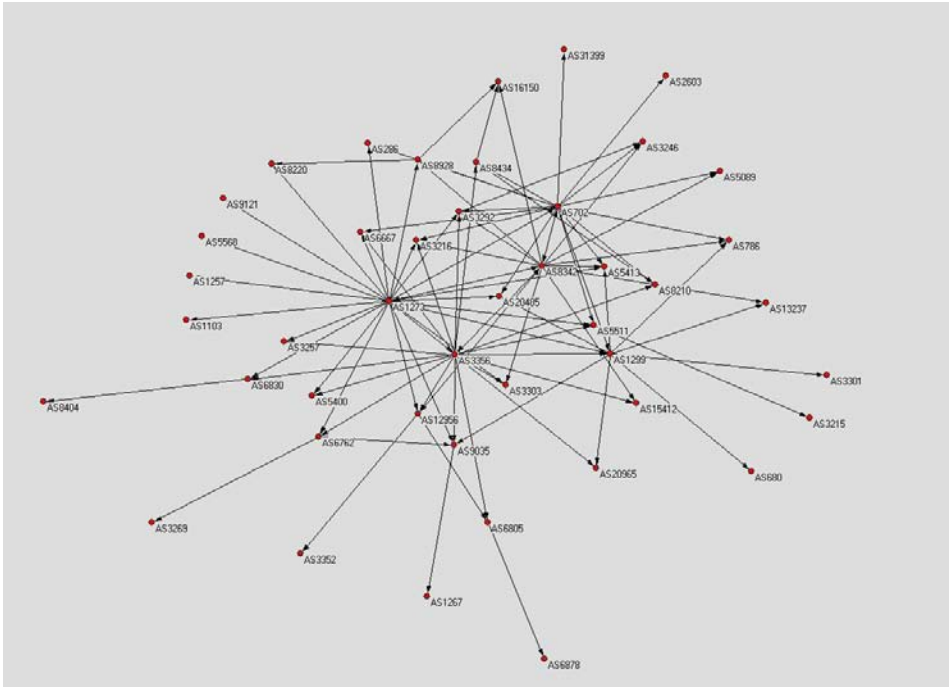
AS16150 - December 19, 2007



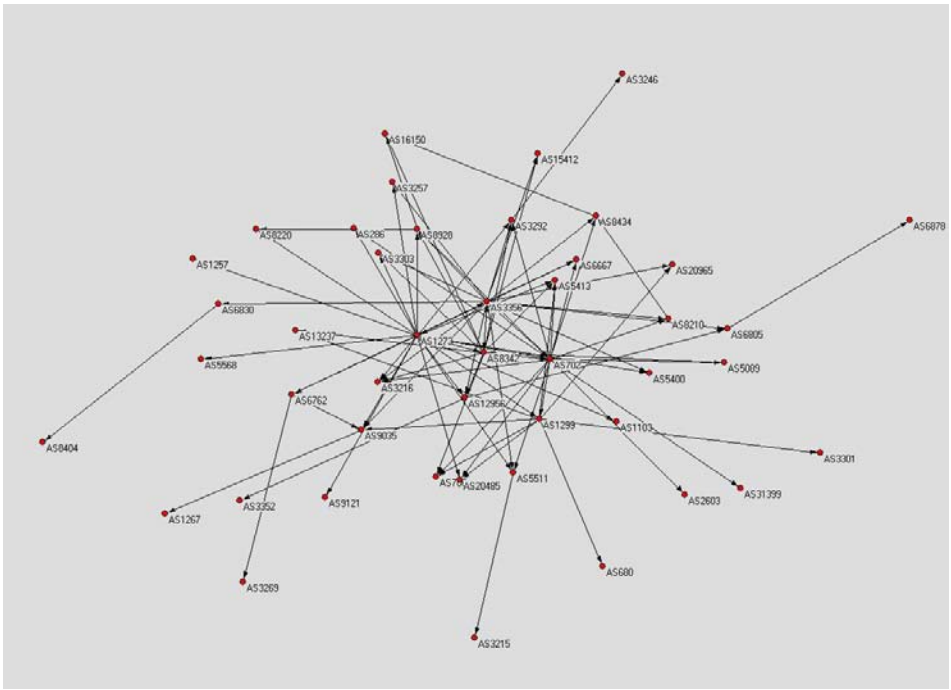
AS8928 - November 29, 2007



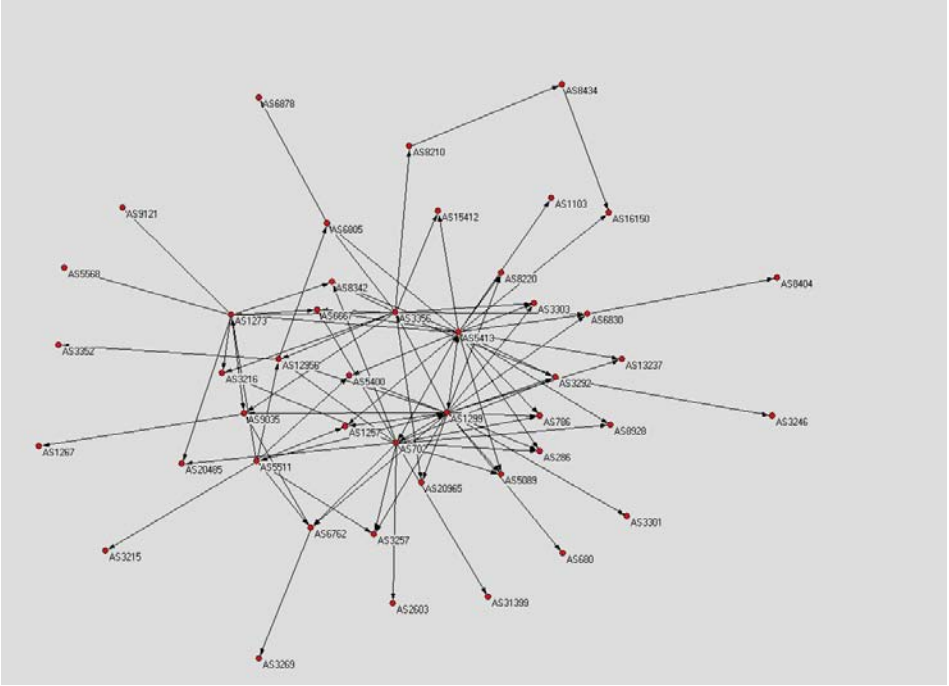
AS8928 - December 19, 2007



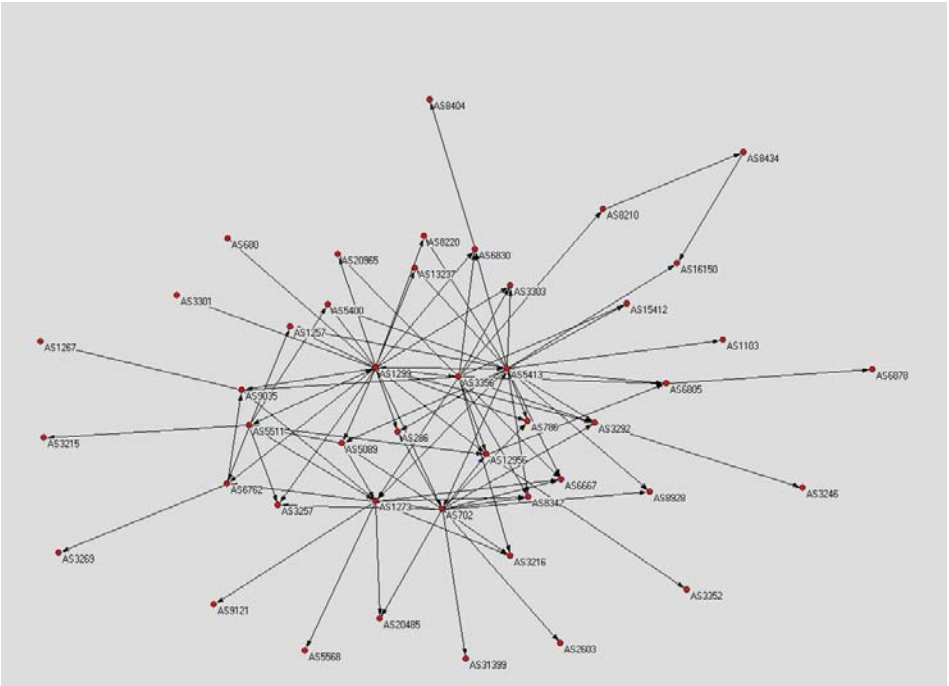
AS8342 - November 29, 2007



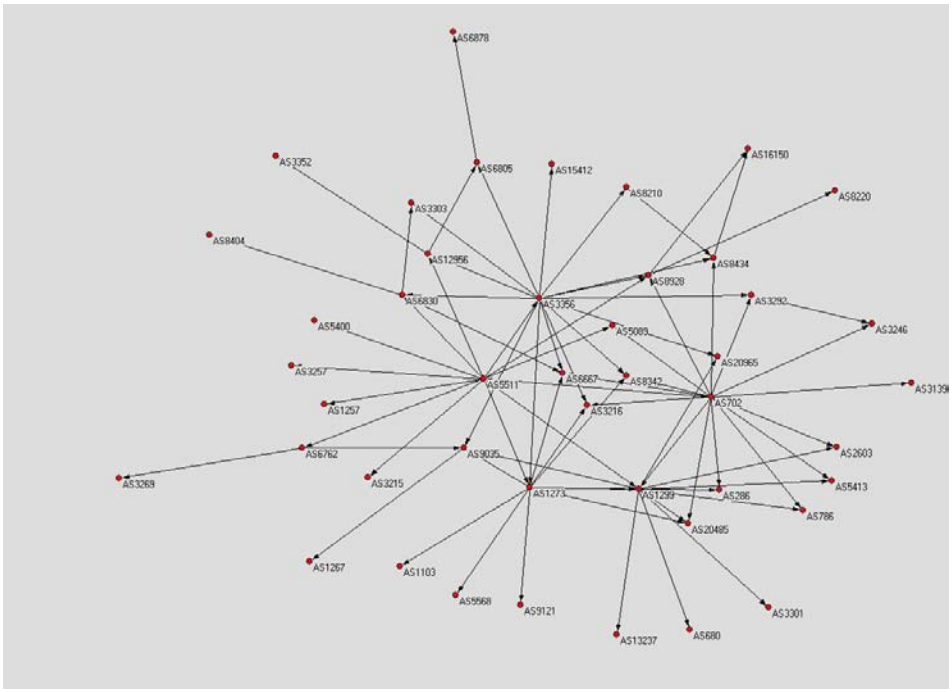
AS8342 - December 19, 2007



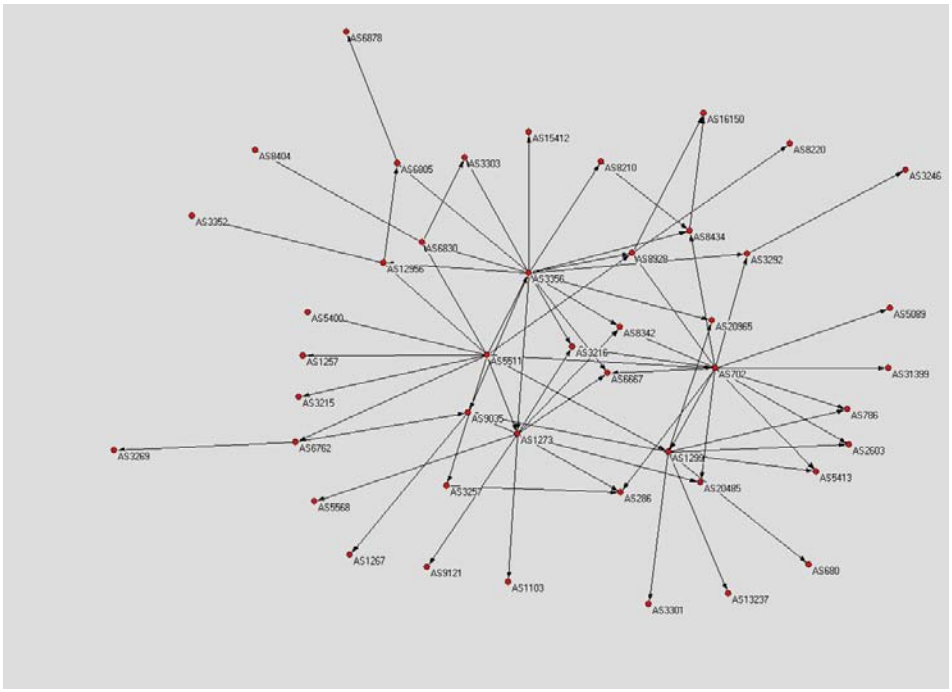
AS5413 - November 29, 2007



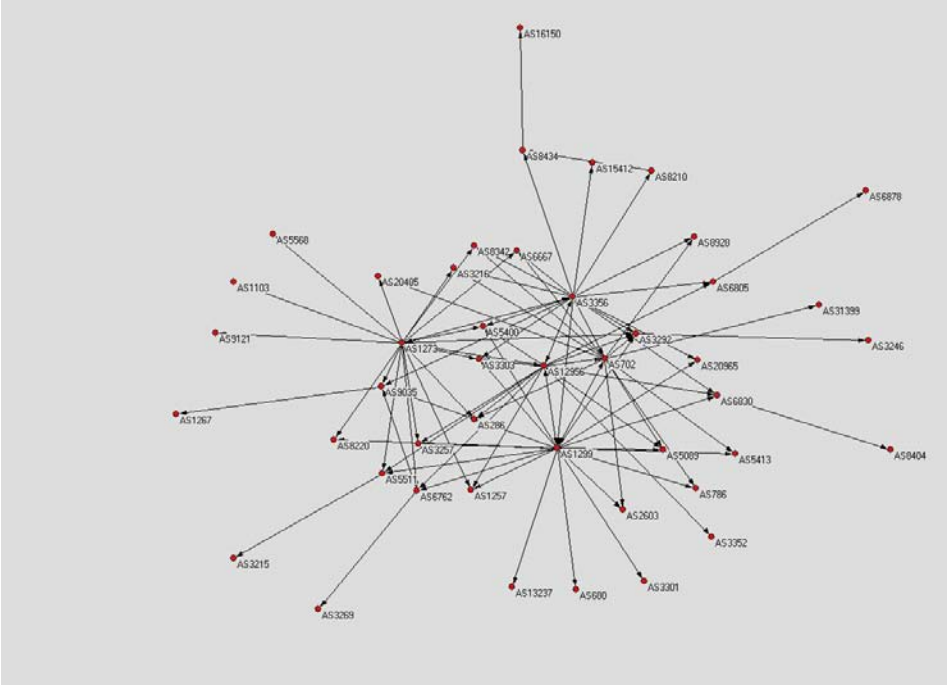
AS5413 - December 19, 2007



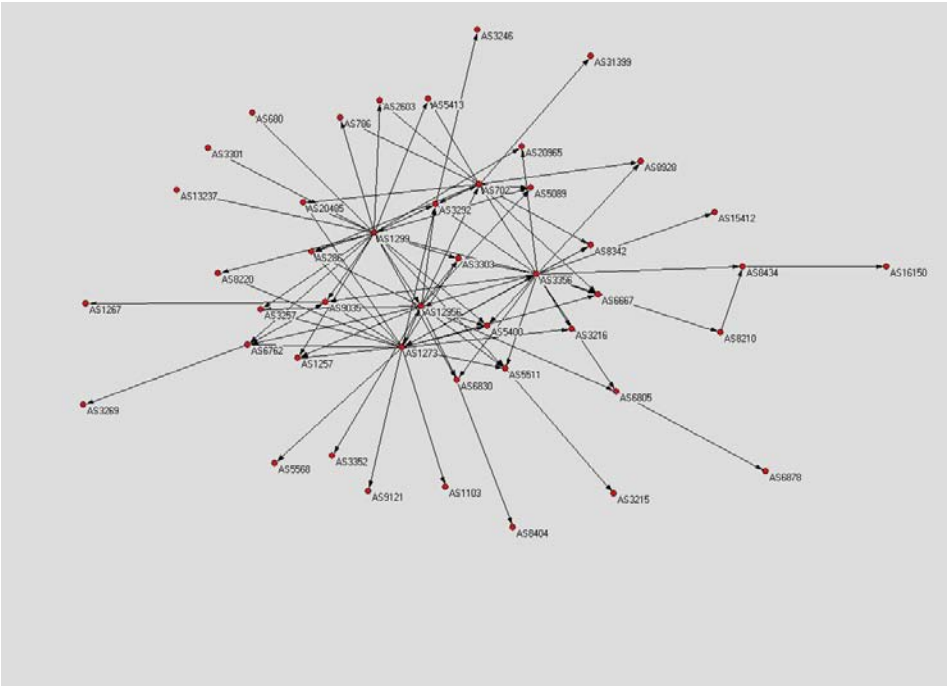
AS5511 - November 29, 2007



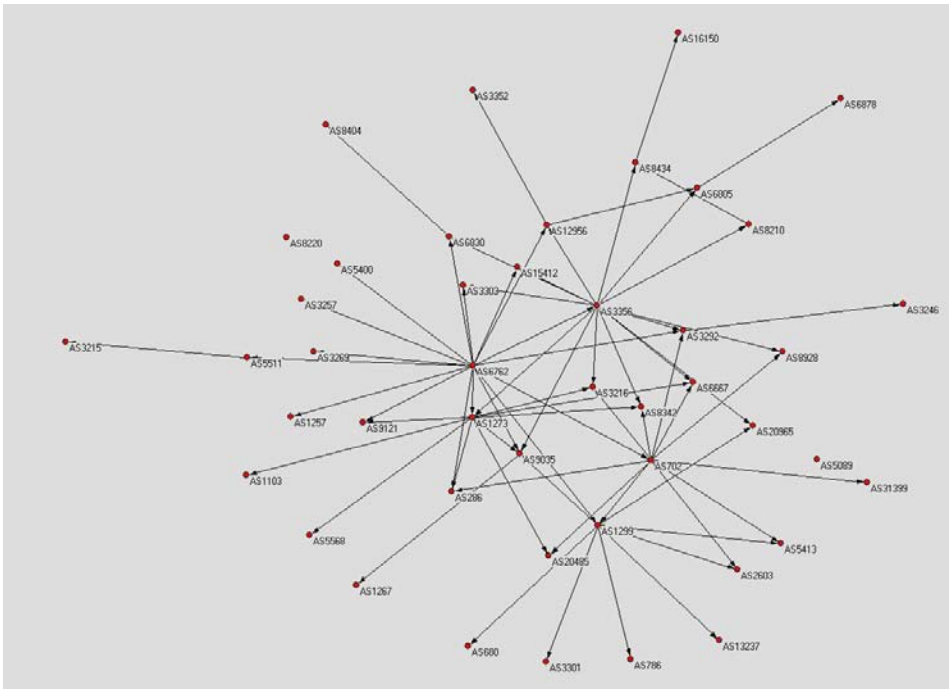
AS5511 - December 19, 2007



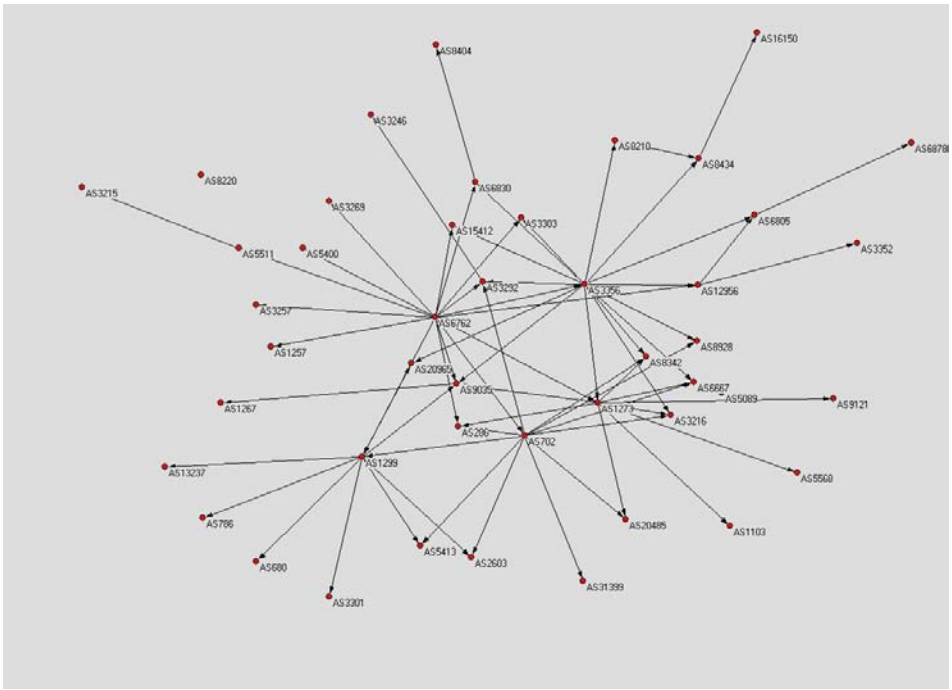
AS12956 - November 29, 2007



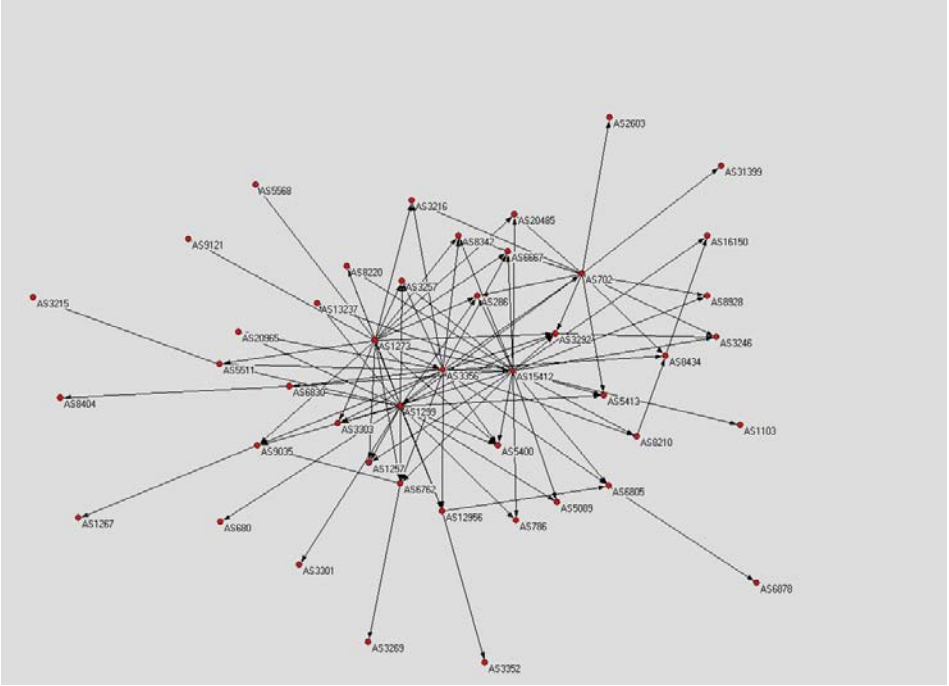
AS12956 - December 19, 2007



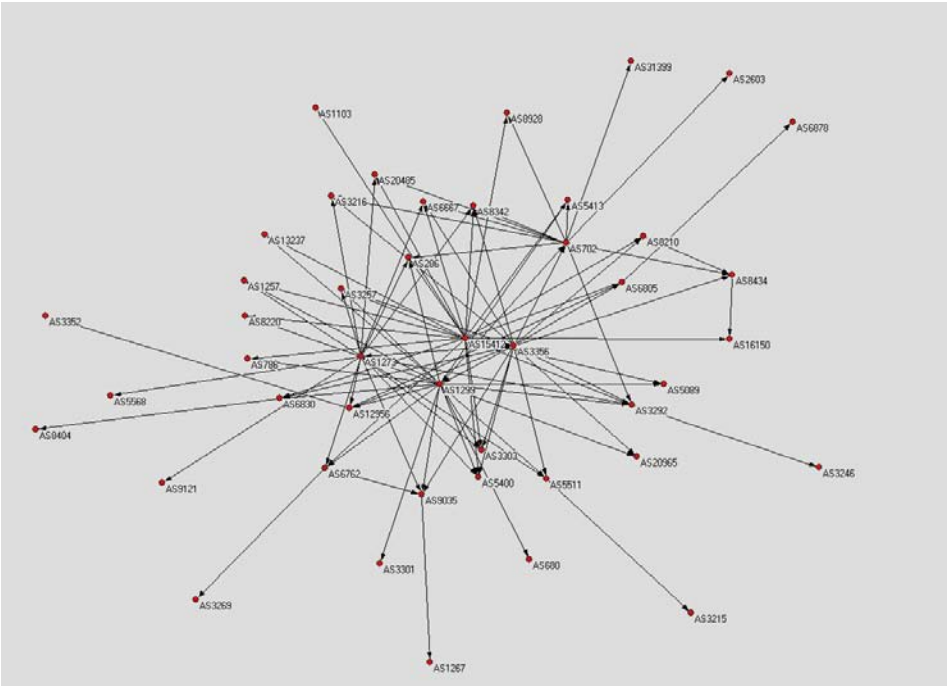
AS6762 - November 29, 2007



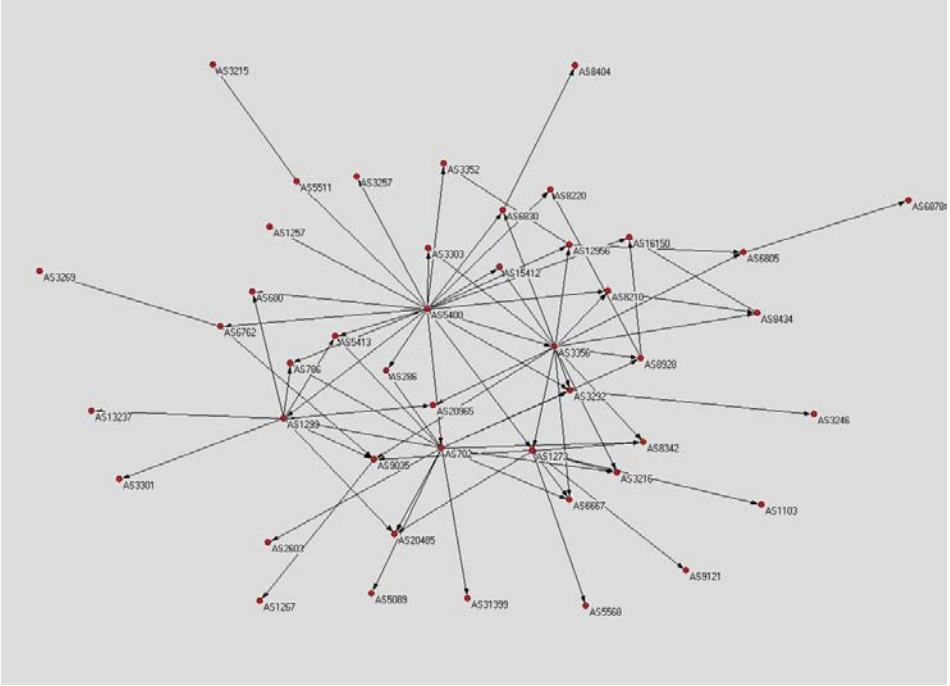
AS6762 - December 19, 2007



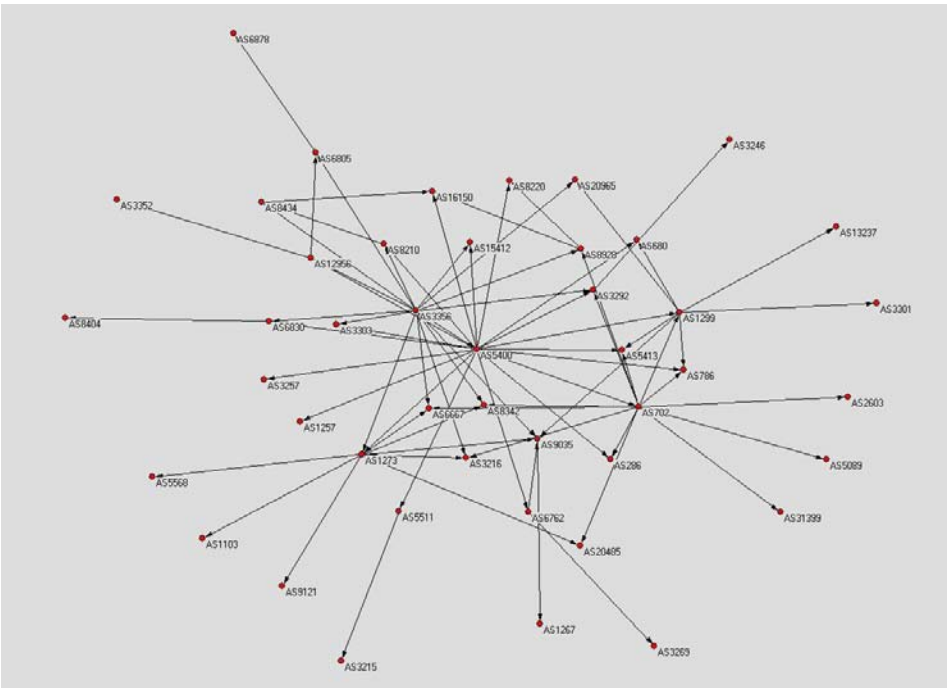
AS15412 - November 29, 2007



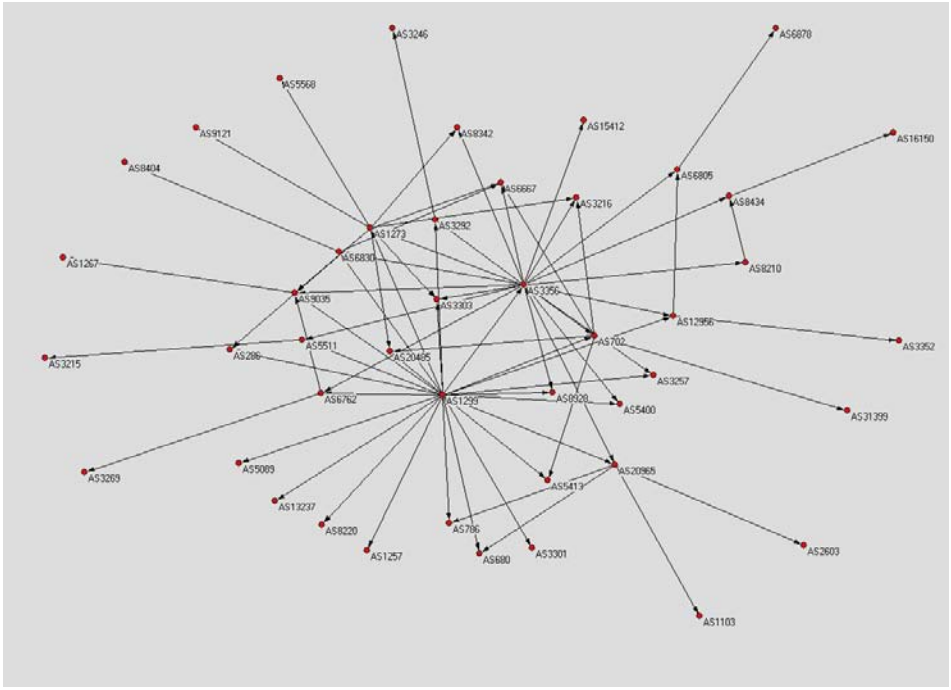
AS15412 - December 19, 2007



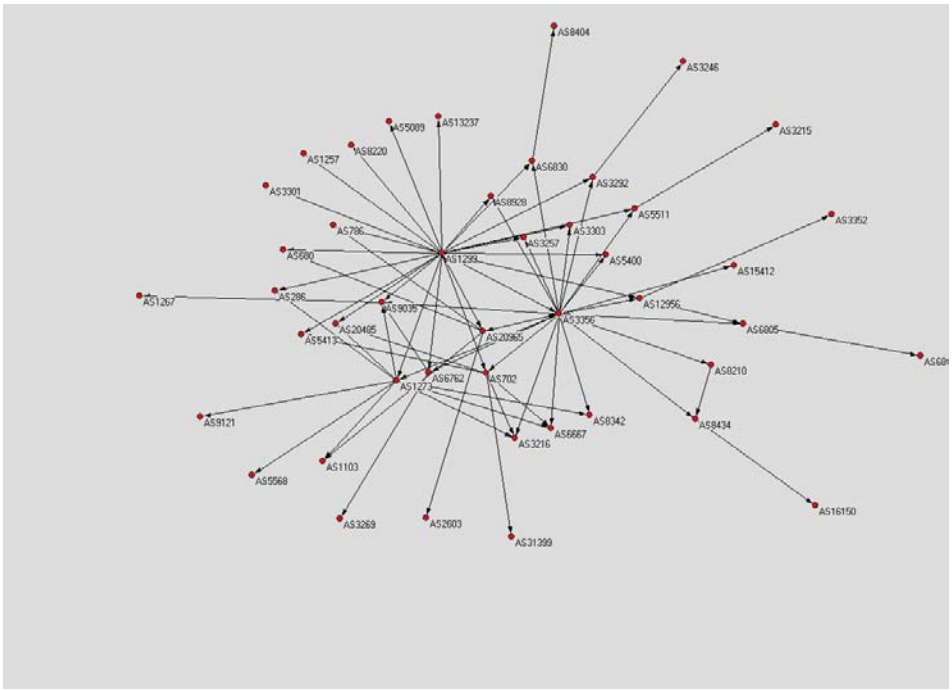
AS5400 - November 29, 2007



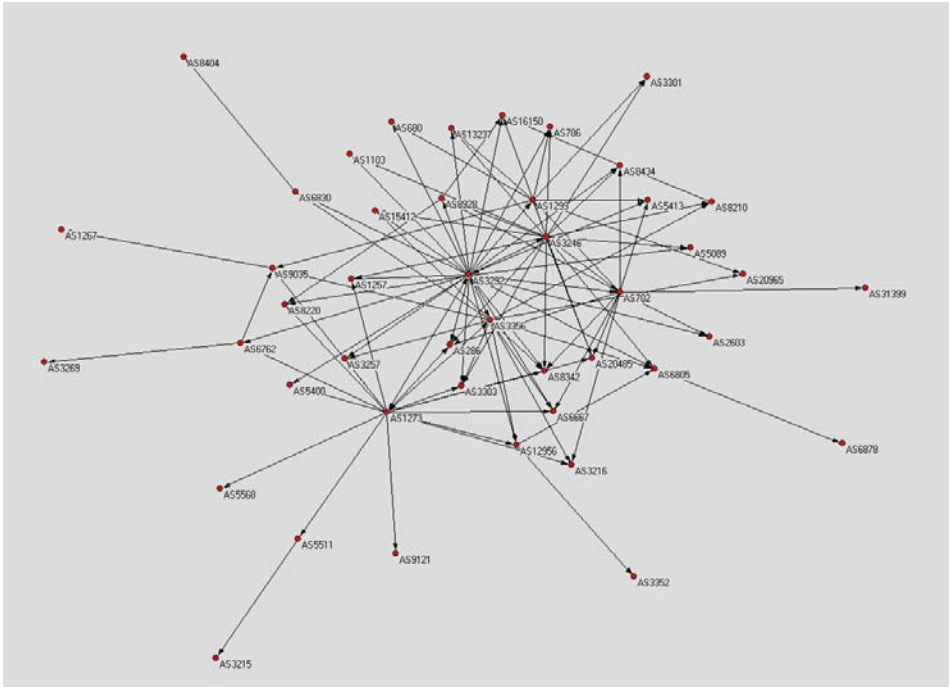
AS5400 - December 19, 2007



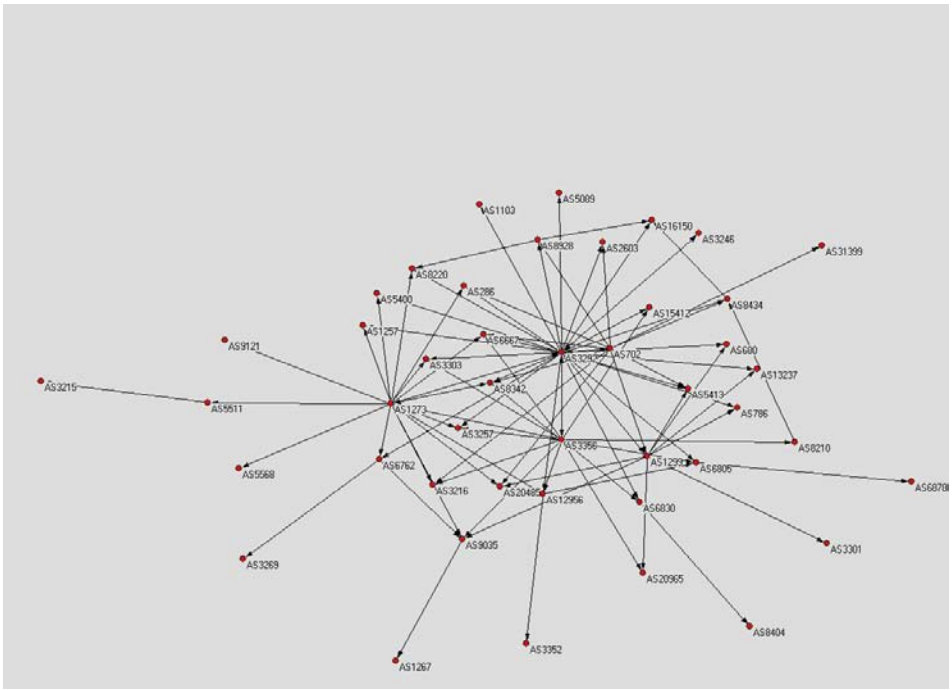
AS20965 - November 29, 2007



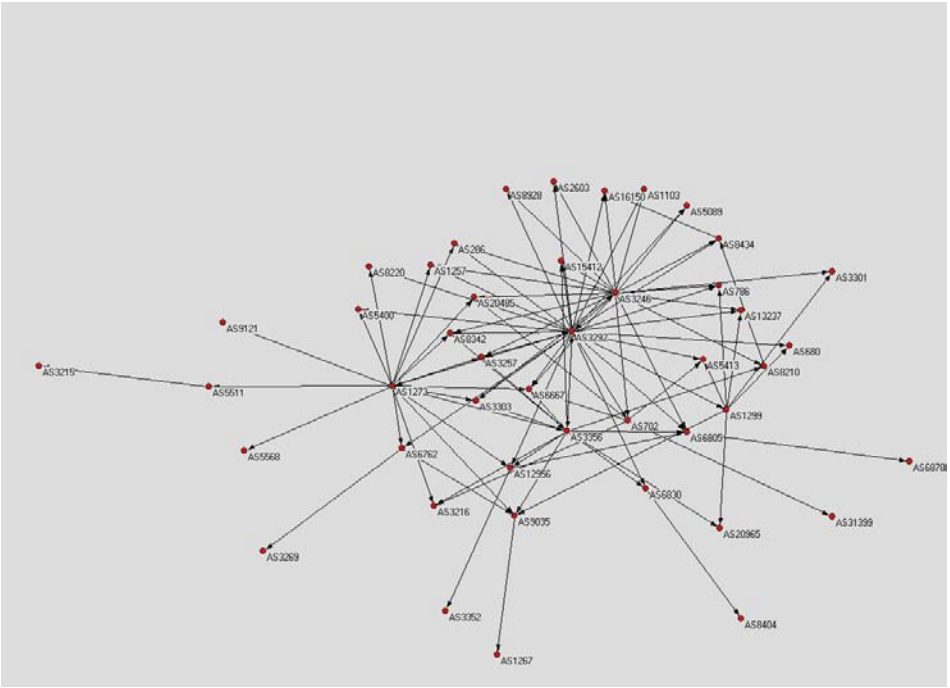
AS20965 - December 19, 2007



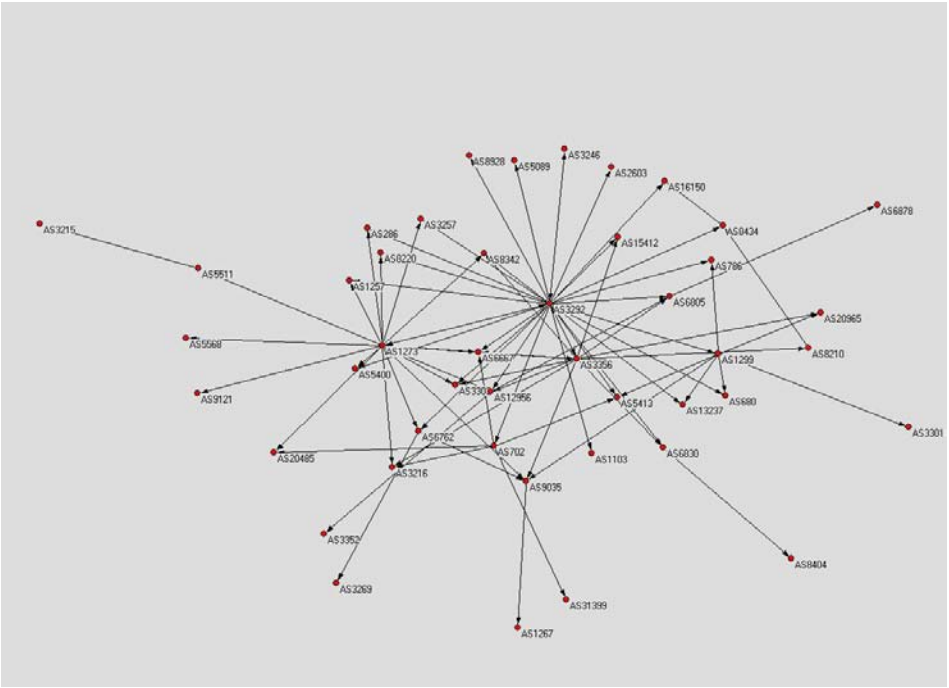
AS3292 - November 29, 2007



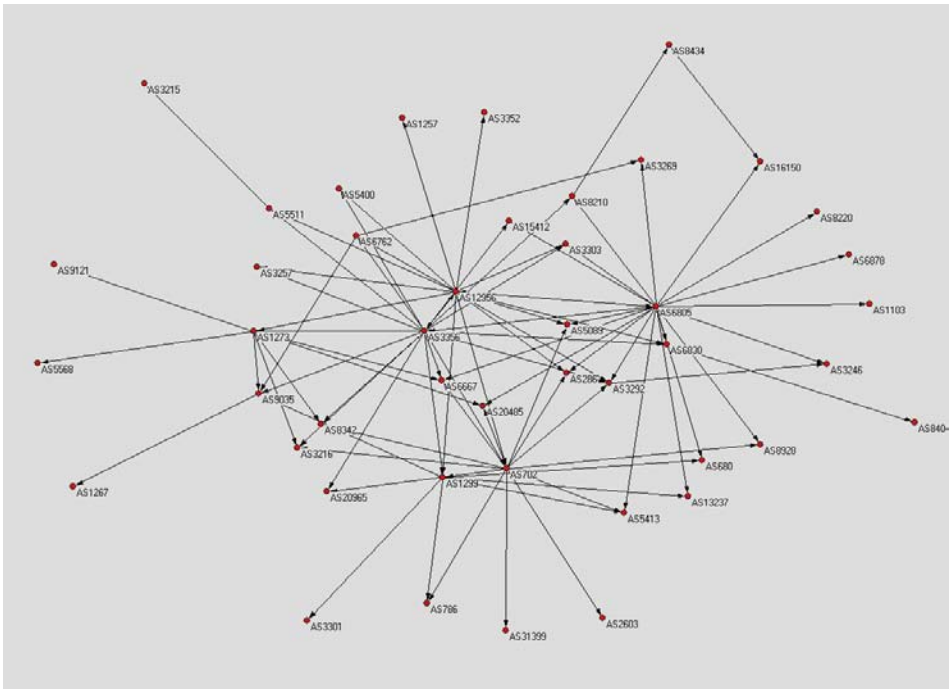
AS3292 - December 19, 2007



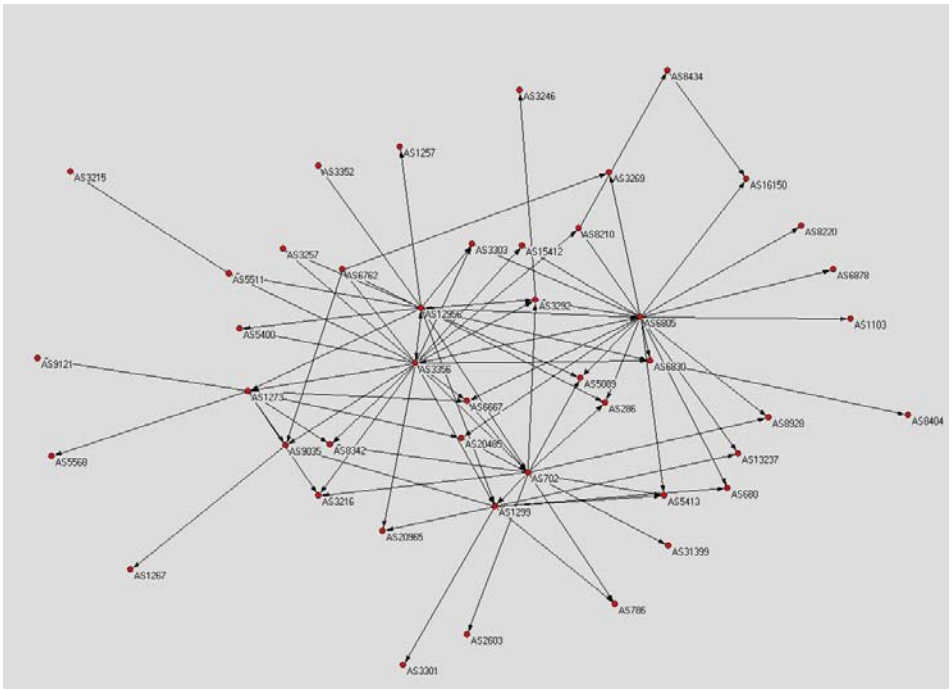
AS3246 - November 29, 2007



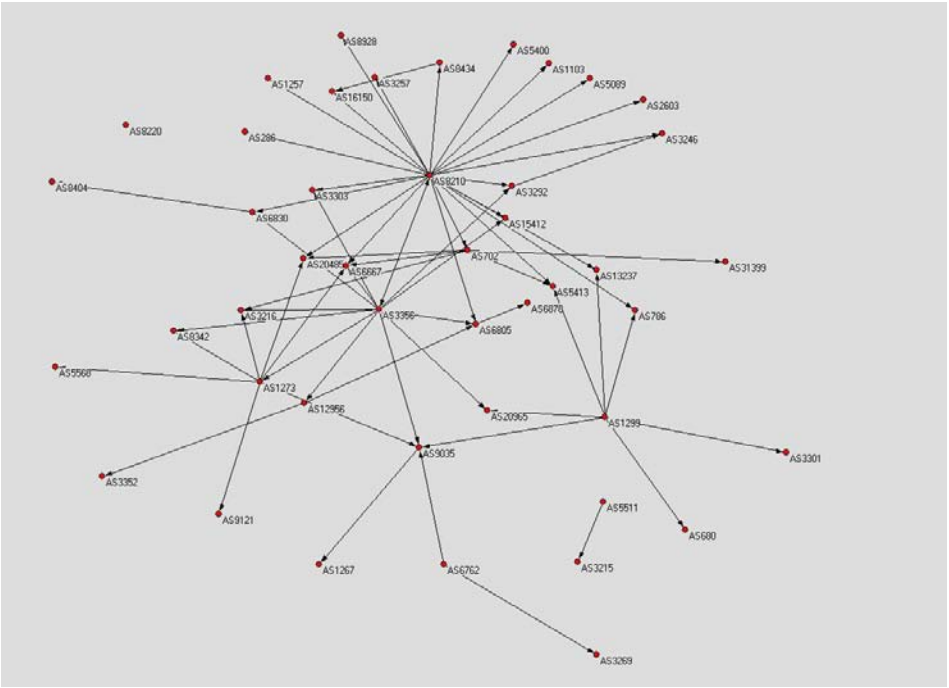
AS3246 - December 19, 2007



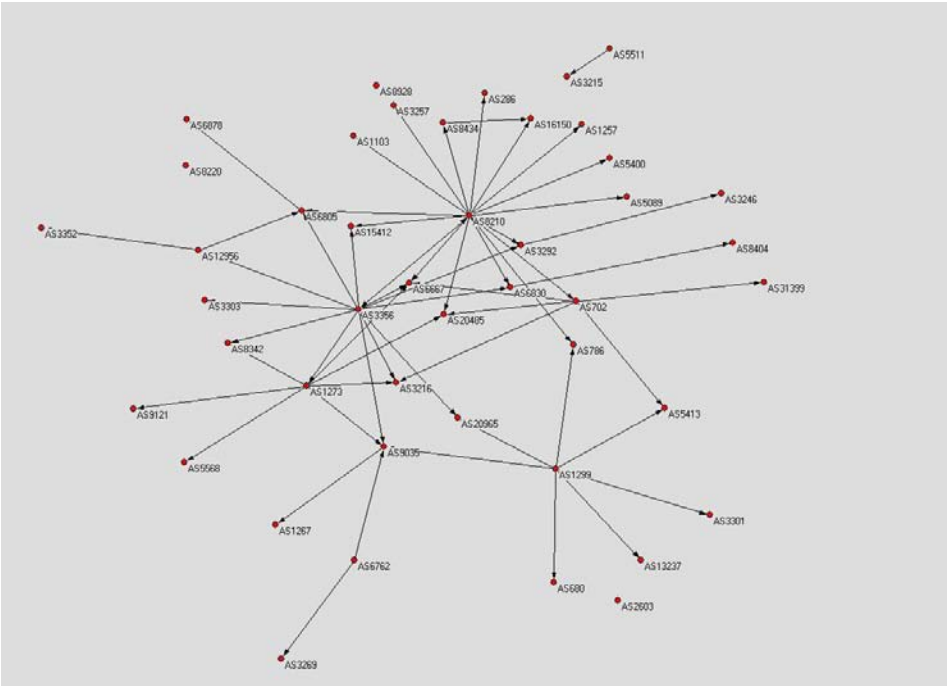
AS6805 - November 29, 2007



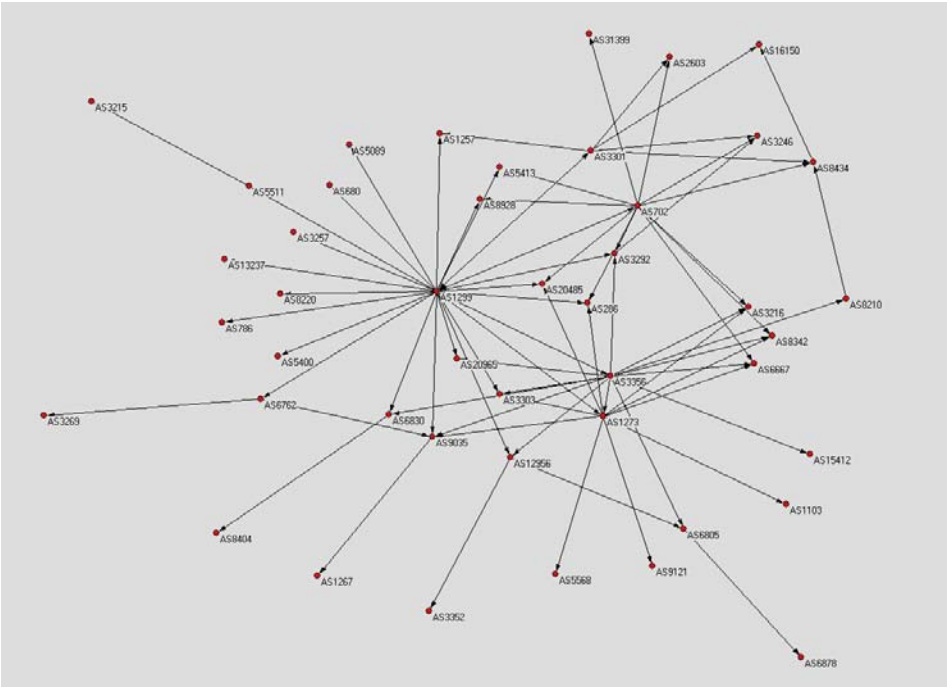
AS6805 - December 19, 2007



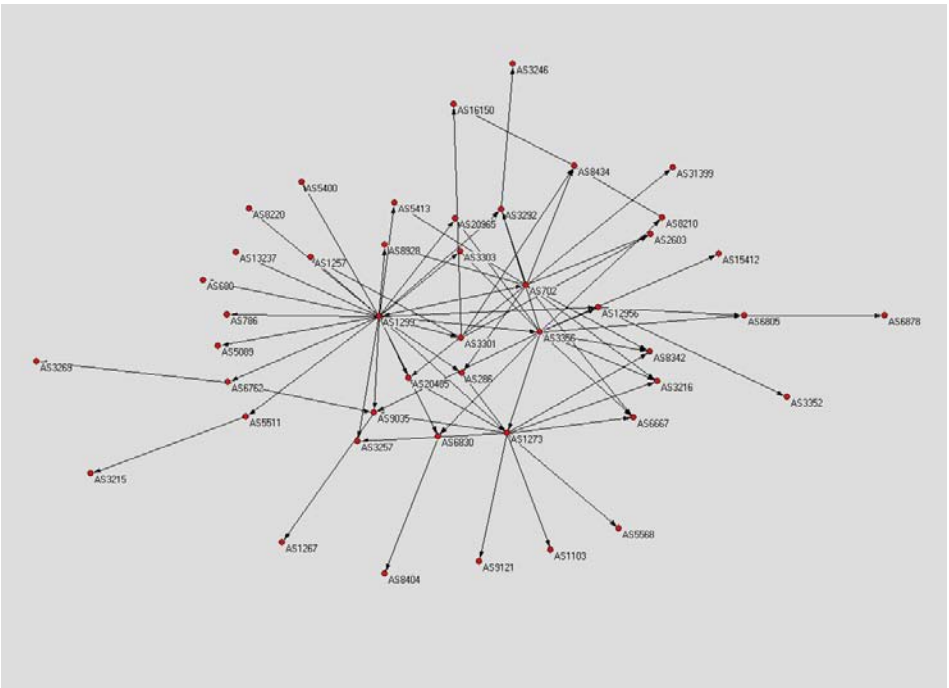
AS8210 - November 29, 2007



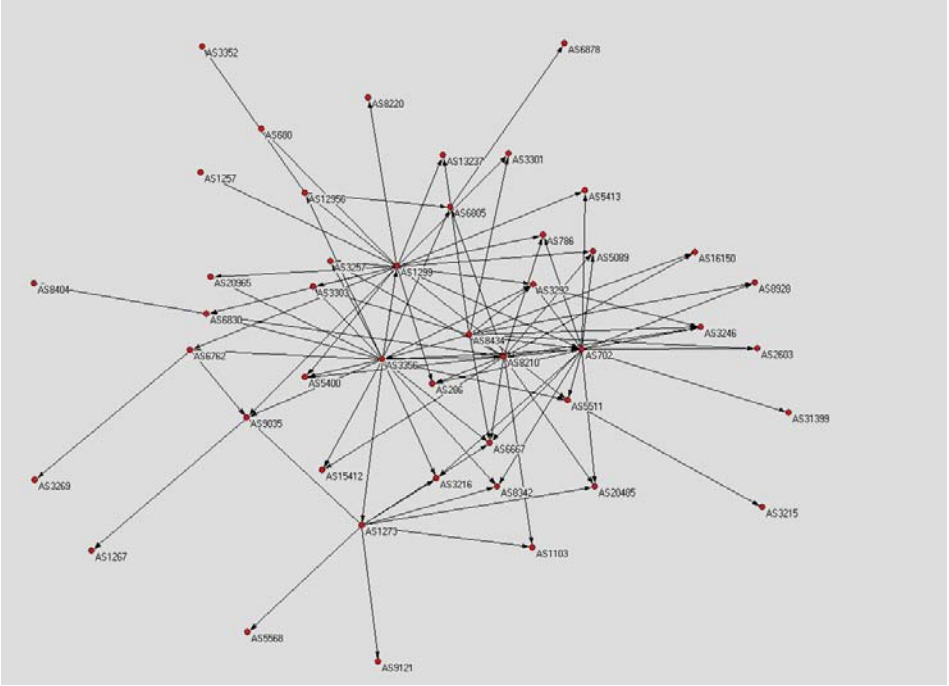
AS8210 - December 19, 2007



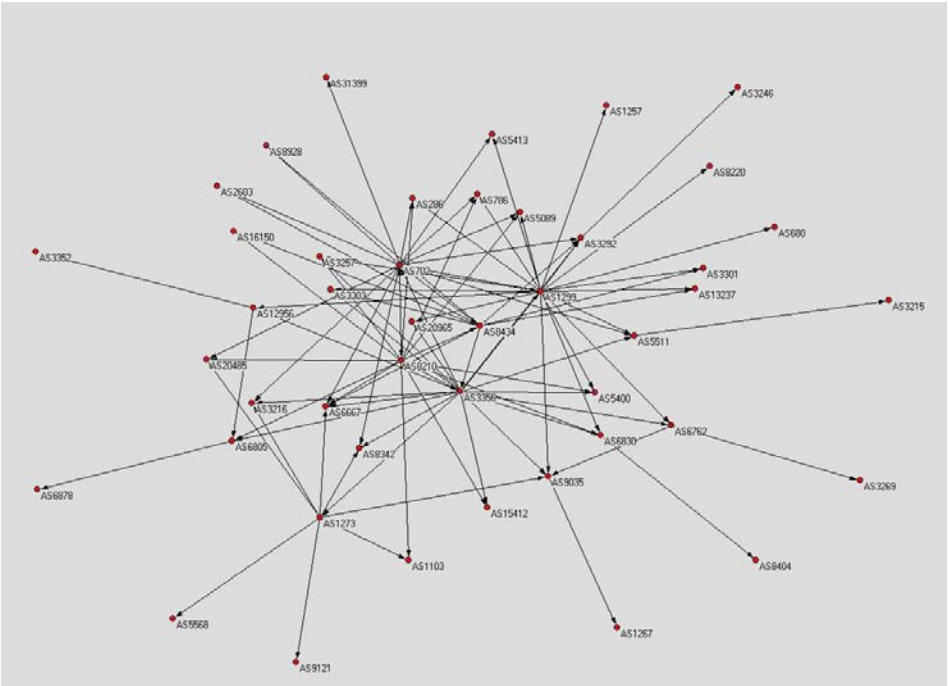
AS3301 - November 29, 2007



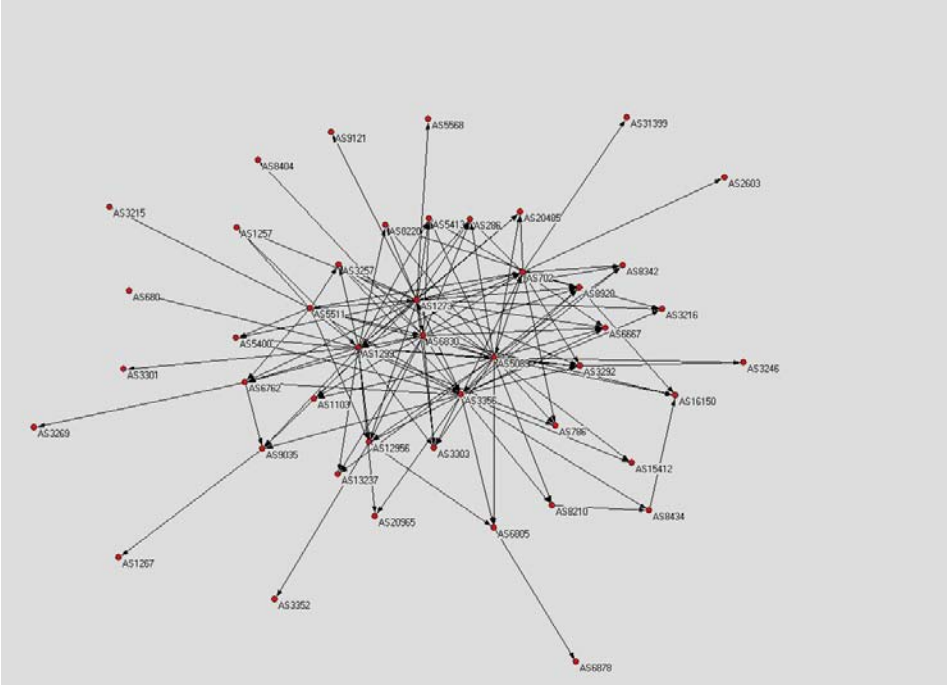
AS3301 - December 19, 2007



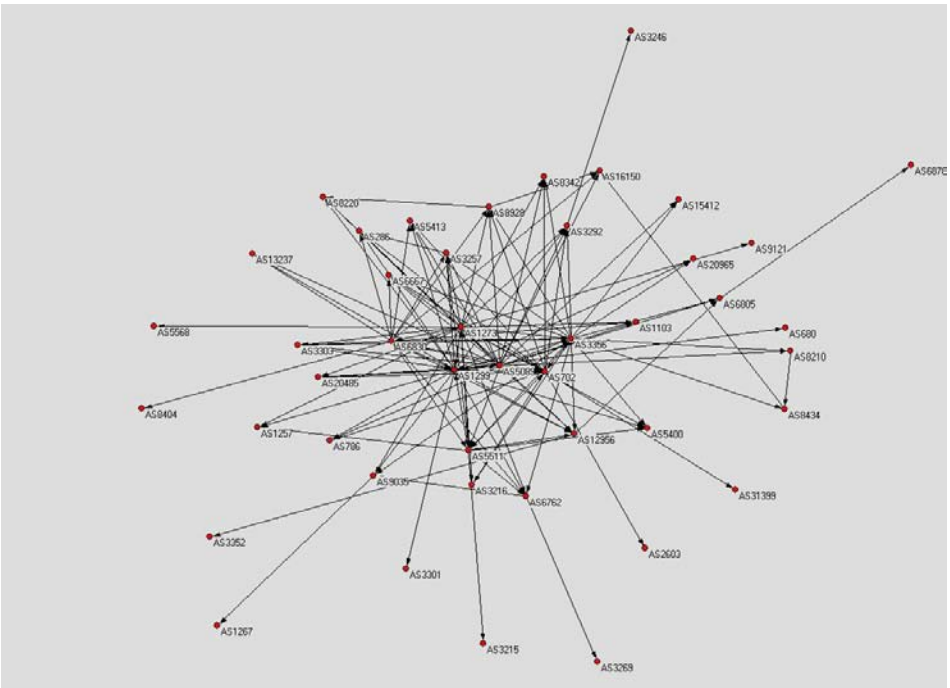
AS8434 - November 29, 2007



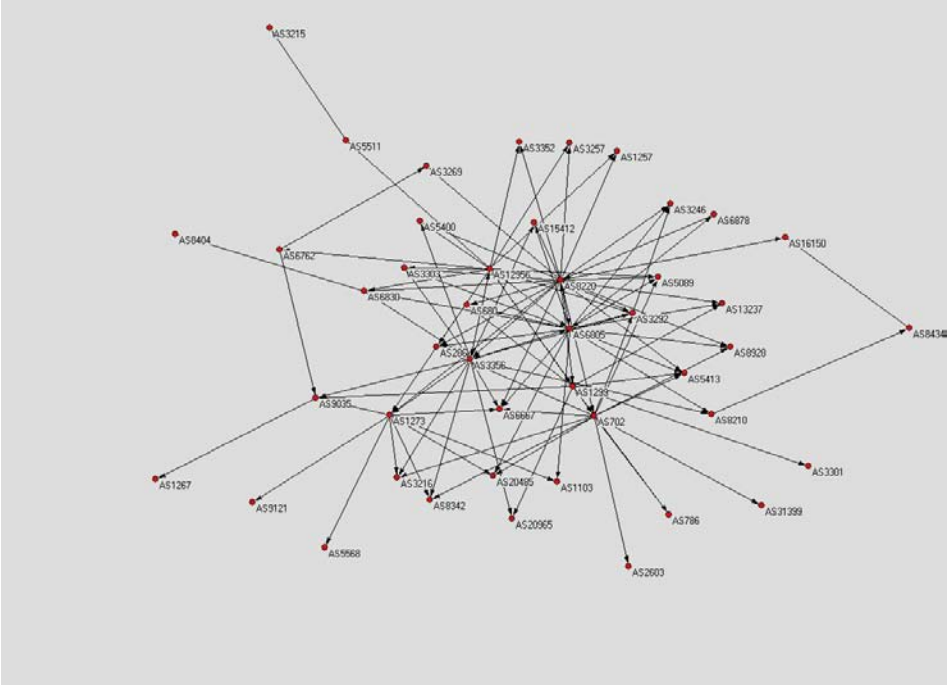
AS8434 - December 19, 2007



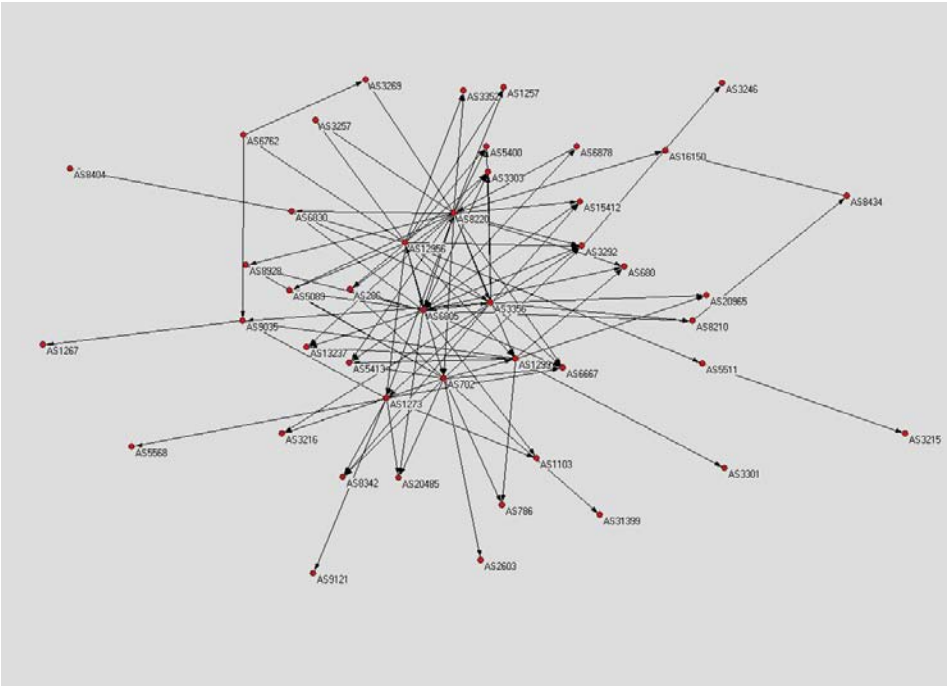
AS5089 - November 29, 2007



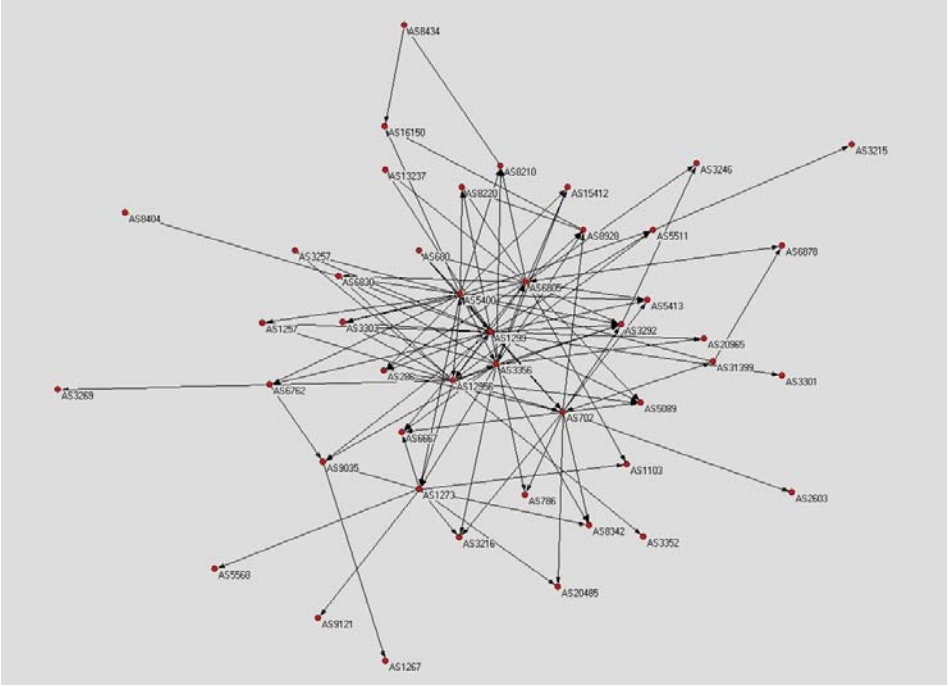
AS5089 - December 19, 2007



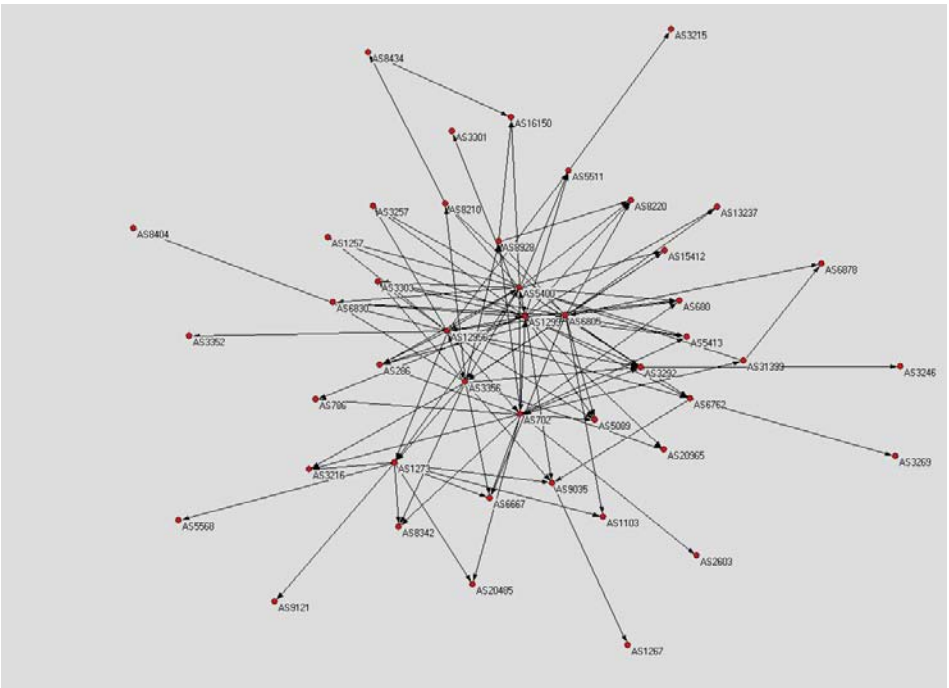
AS6878 - November 29, 2007



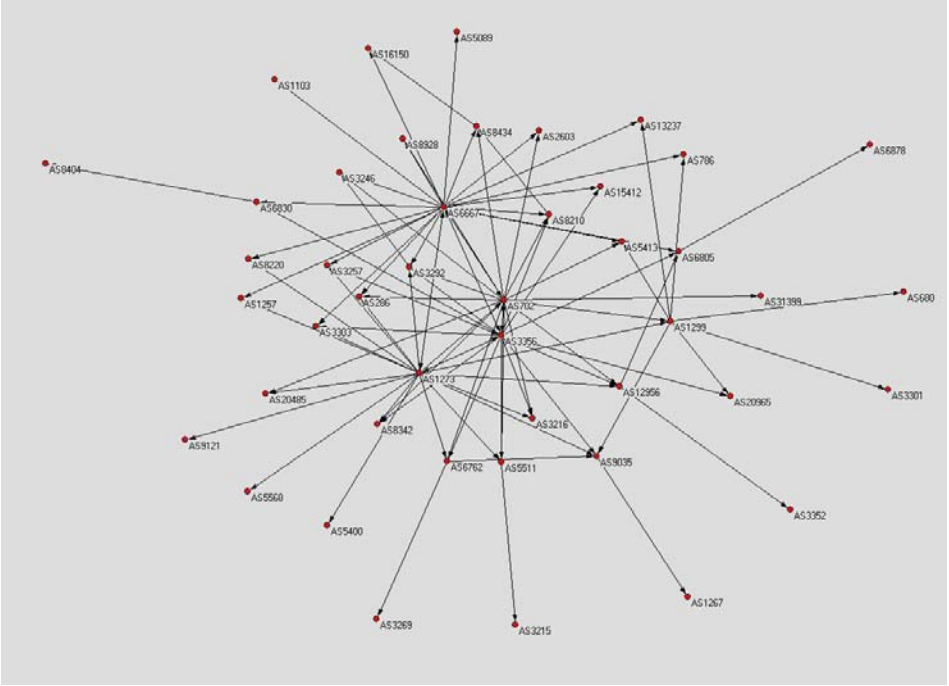
AS6878 - December 19, 2007



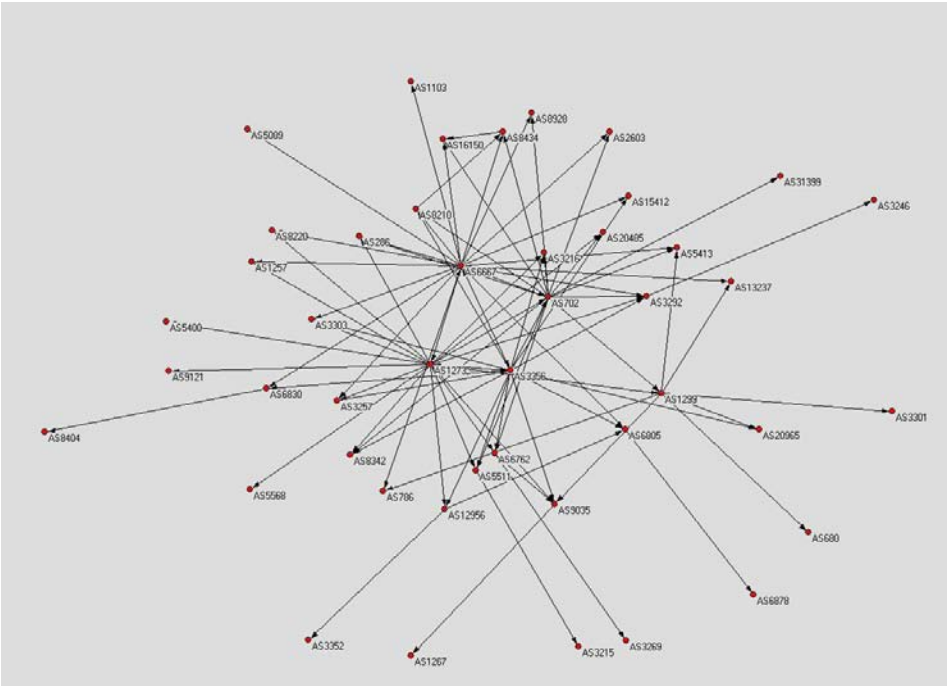
AS31399 - November 29, 2007



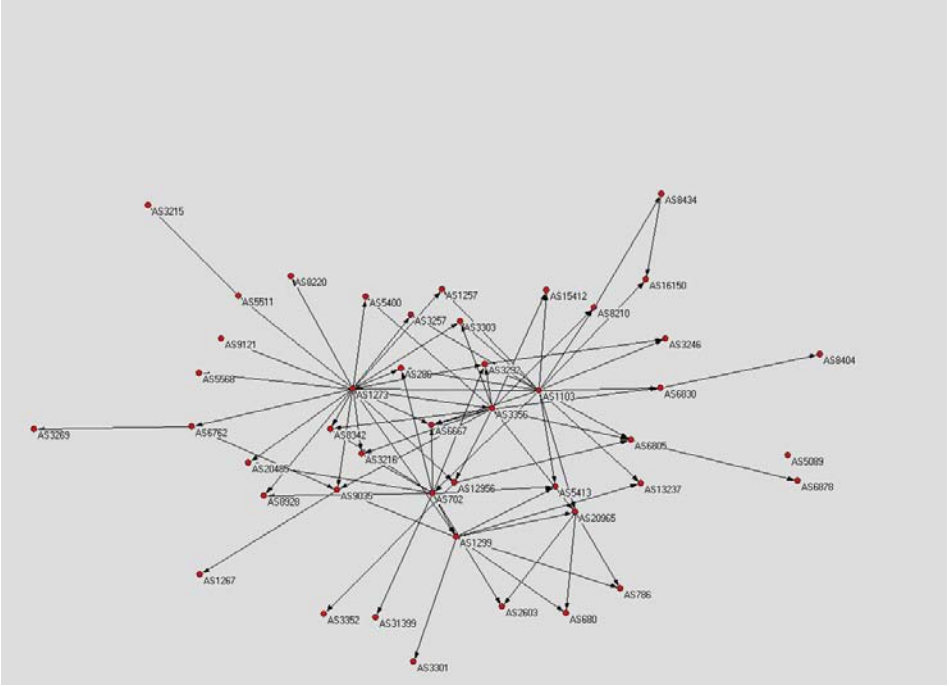
AS31399 - December 19, 2007



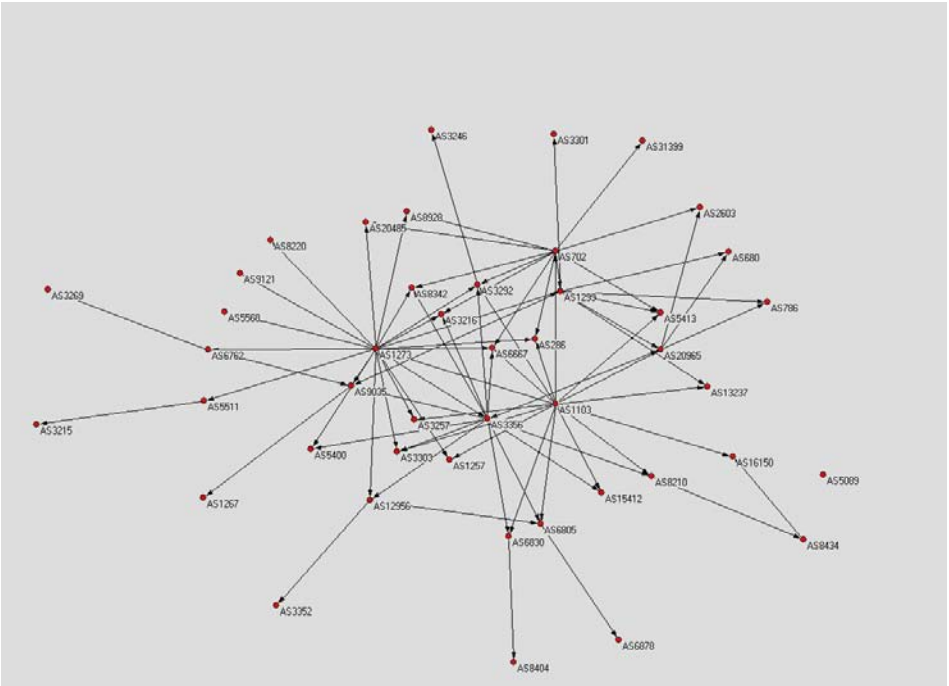
AS6667 - November 29, 2007



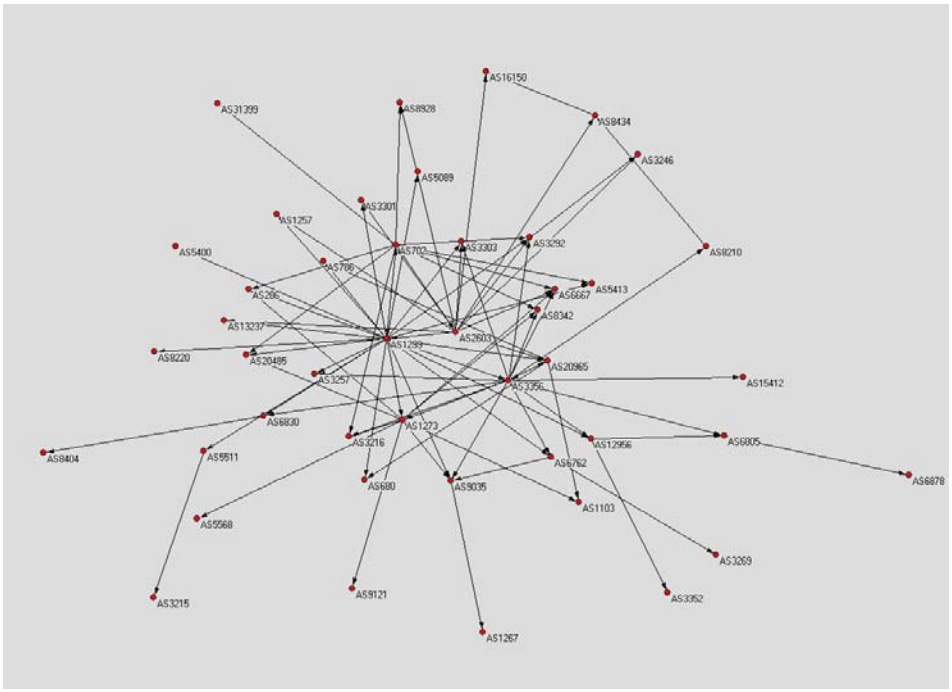
AS6667 - December 19, 2007



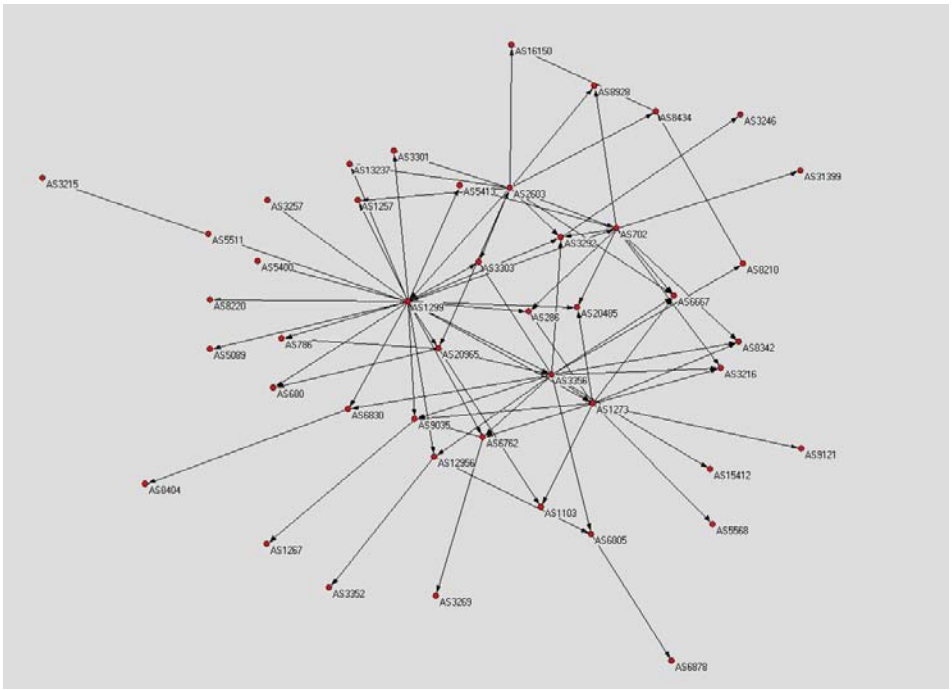
AS1103 - November 29, 2007



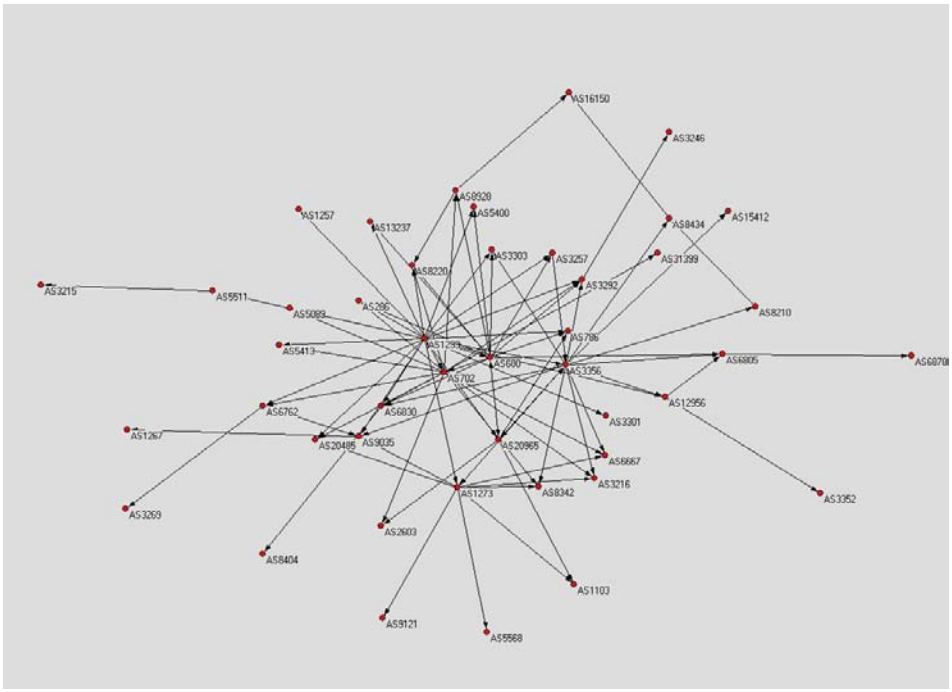
AS1103 - December 19, 2007



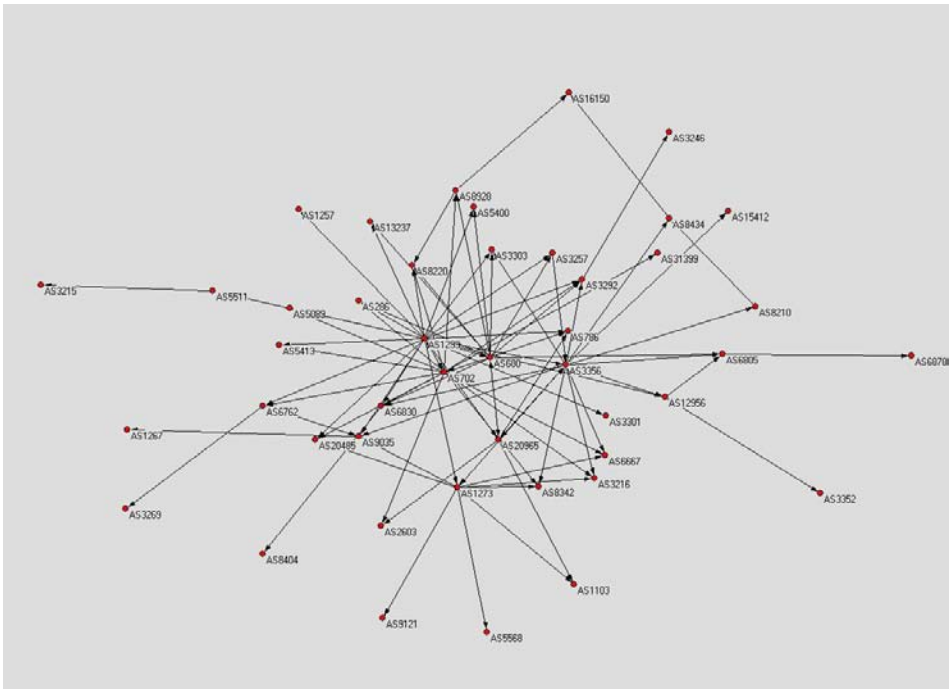
AS2603 - November 29, 2007



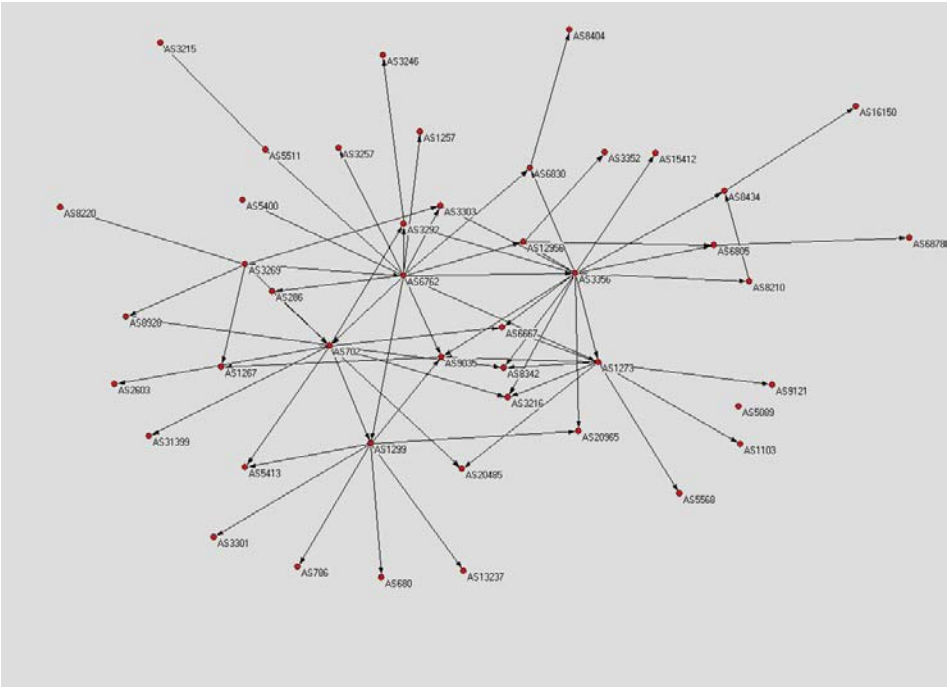
AS2603 - December 19, 2007



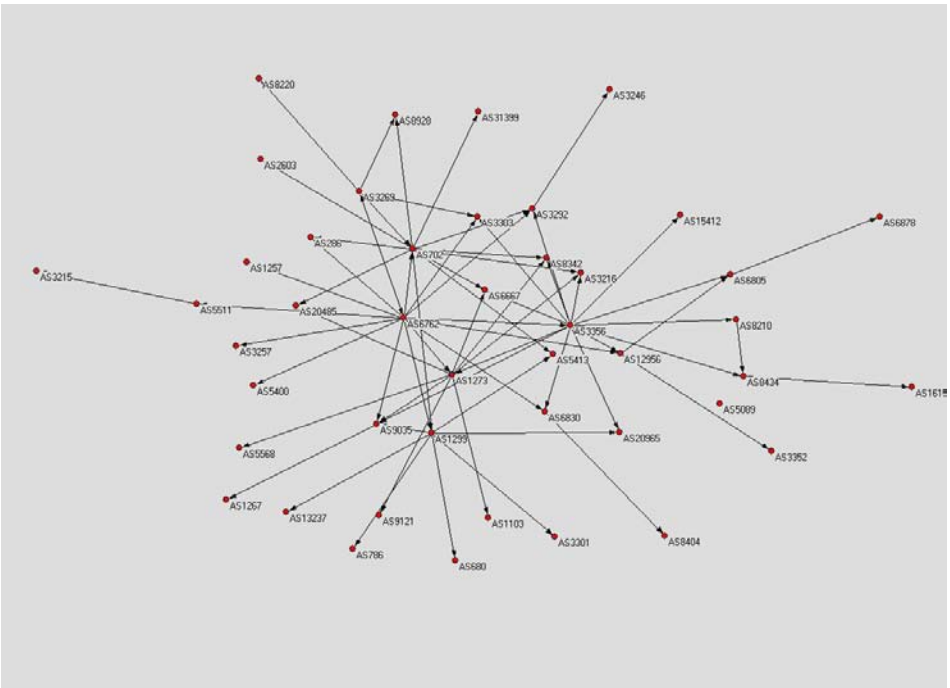
AS680 - November 29, 2007



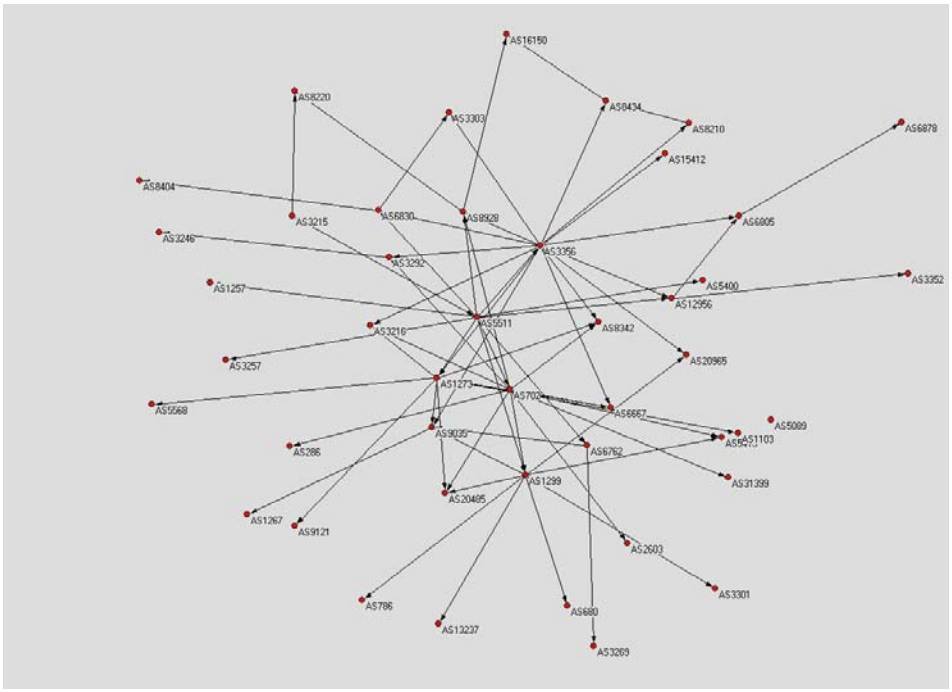
AS680 - December 19, 2007



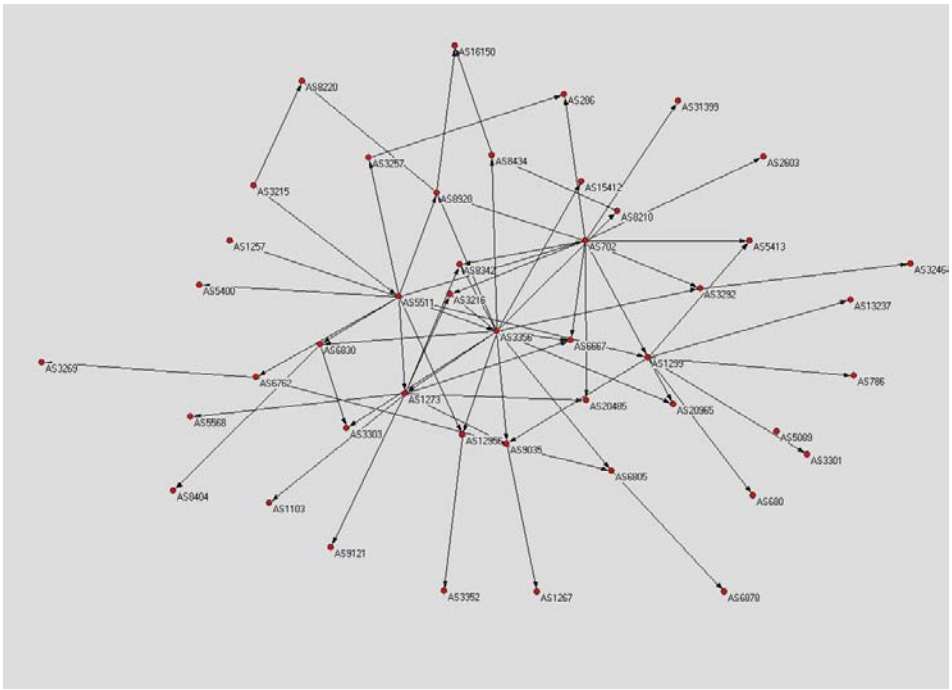
AS3269 - November 29, 2007



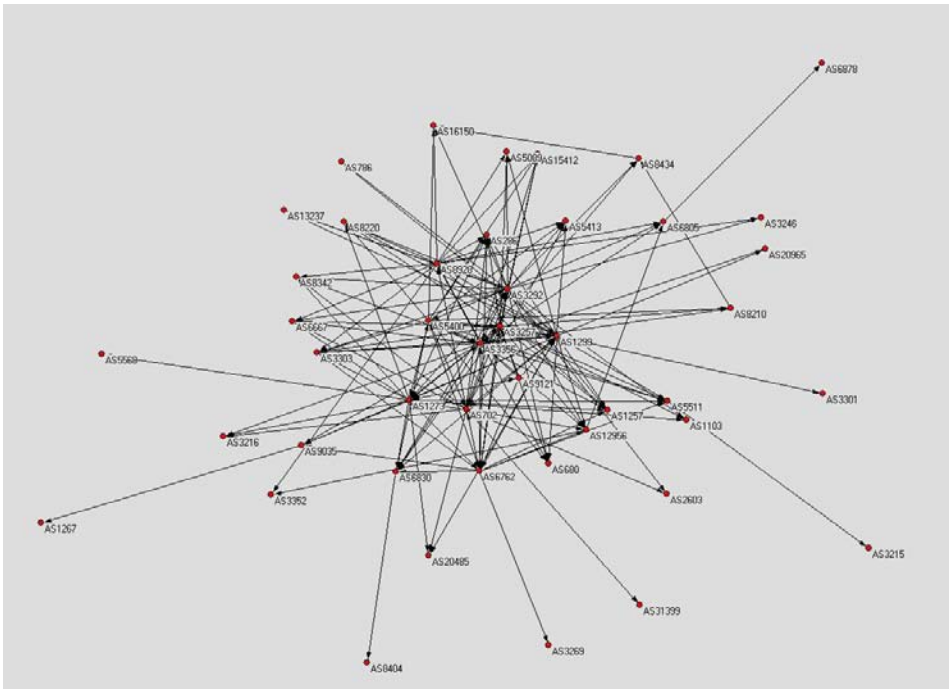
AS3269 - December 19, 2007



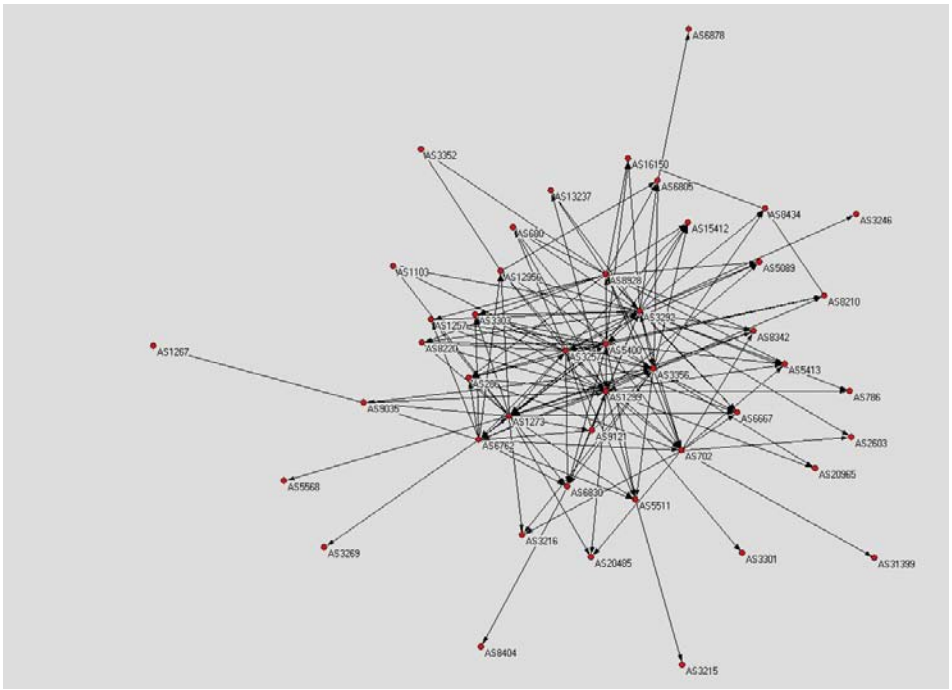
AS3215 - November 29, 2007



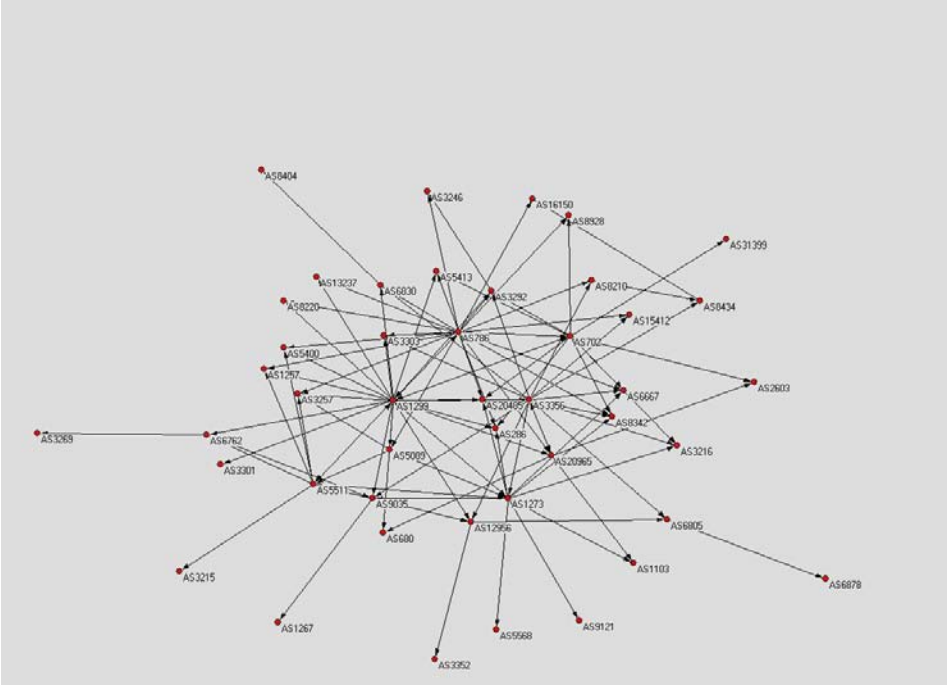
AS3215 - December 19, 2007



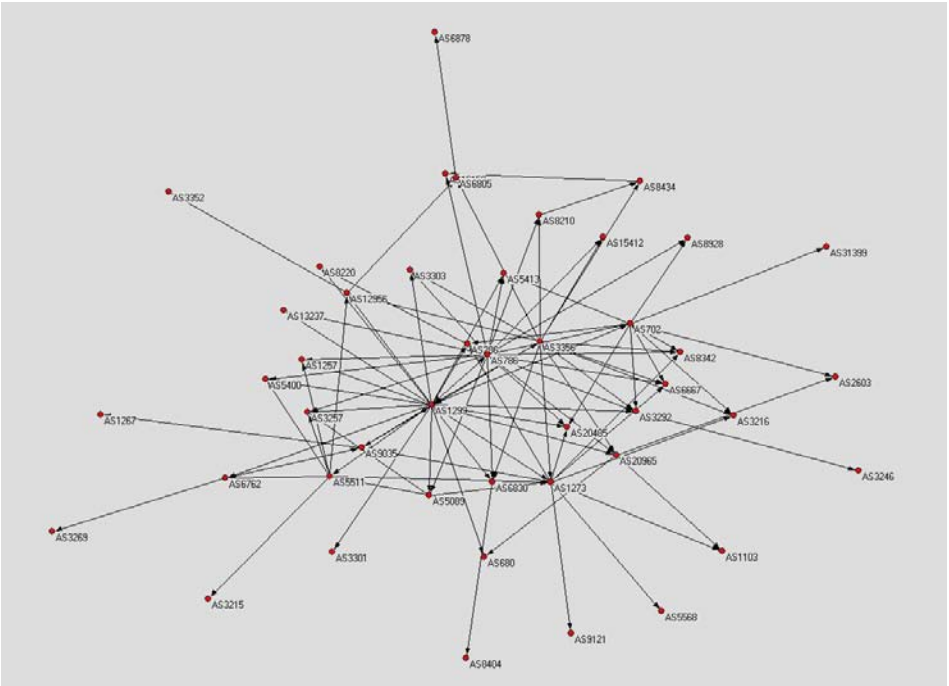
AS9121 - November 29, 2007



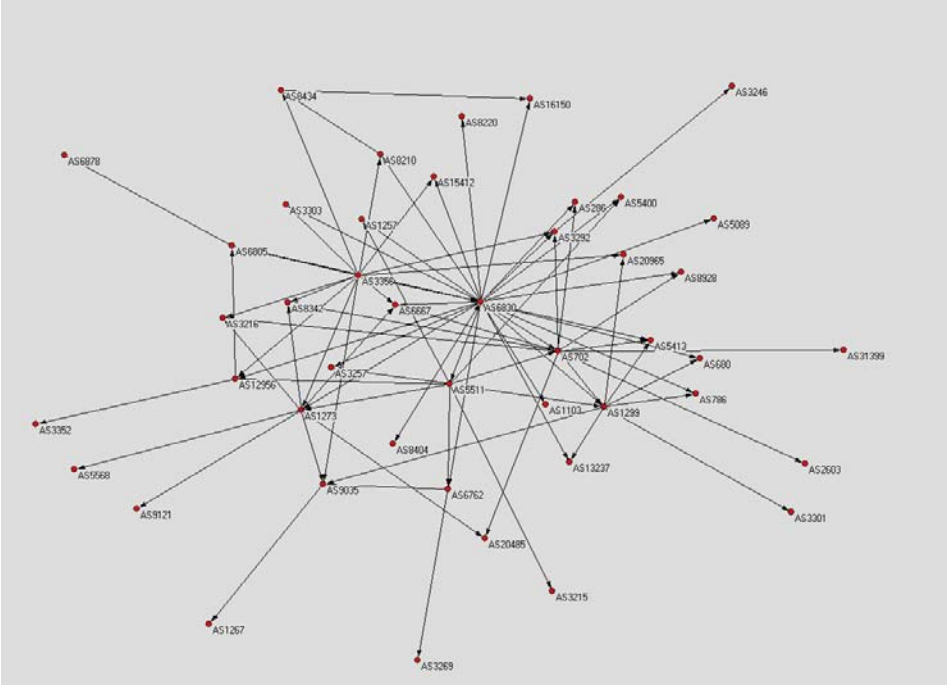
AS9121 - December 19, 2007



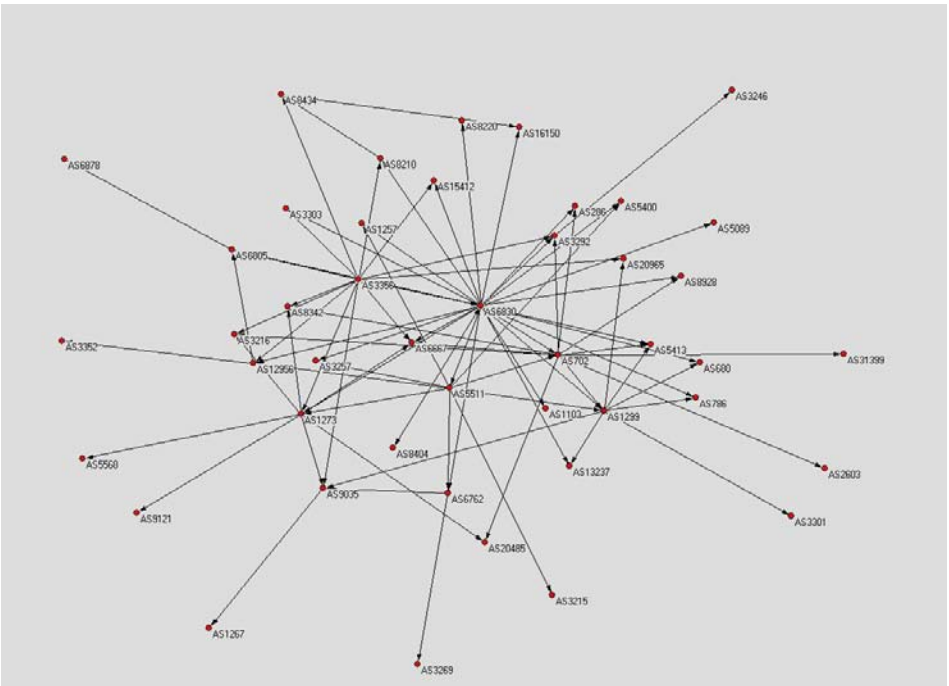
AS786 - November 29, 2007



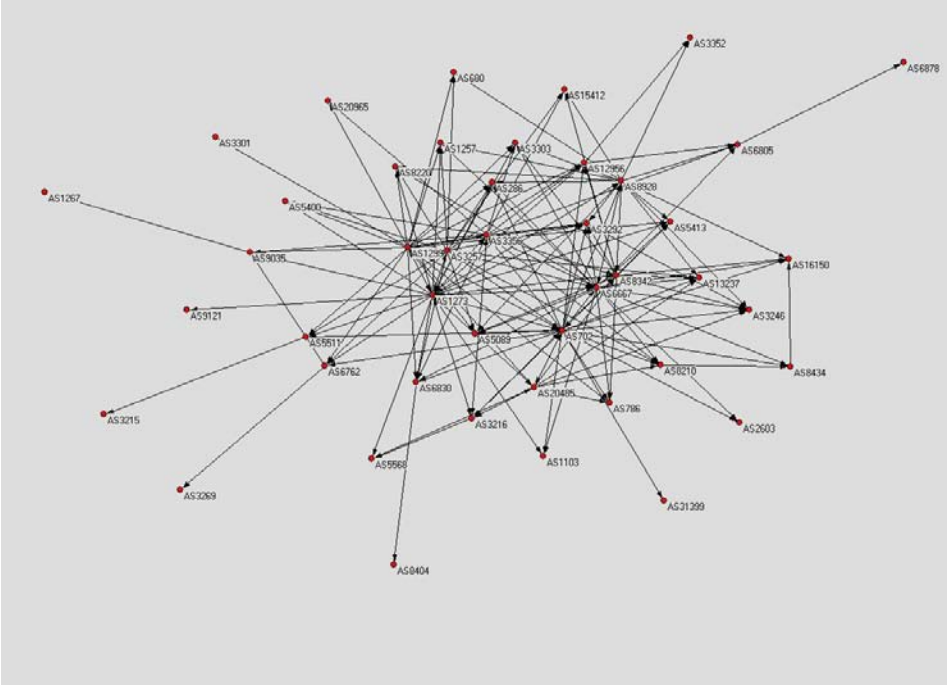
AS786 - December 19, 2007



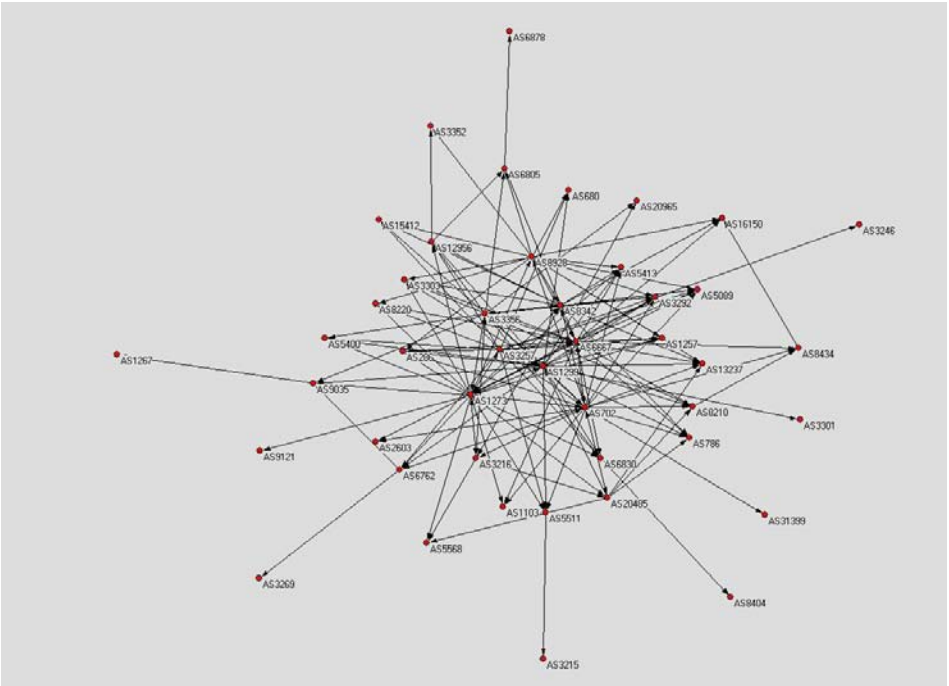
AS8404 - November 29, 2007



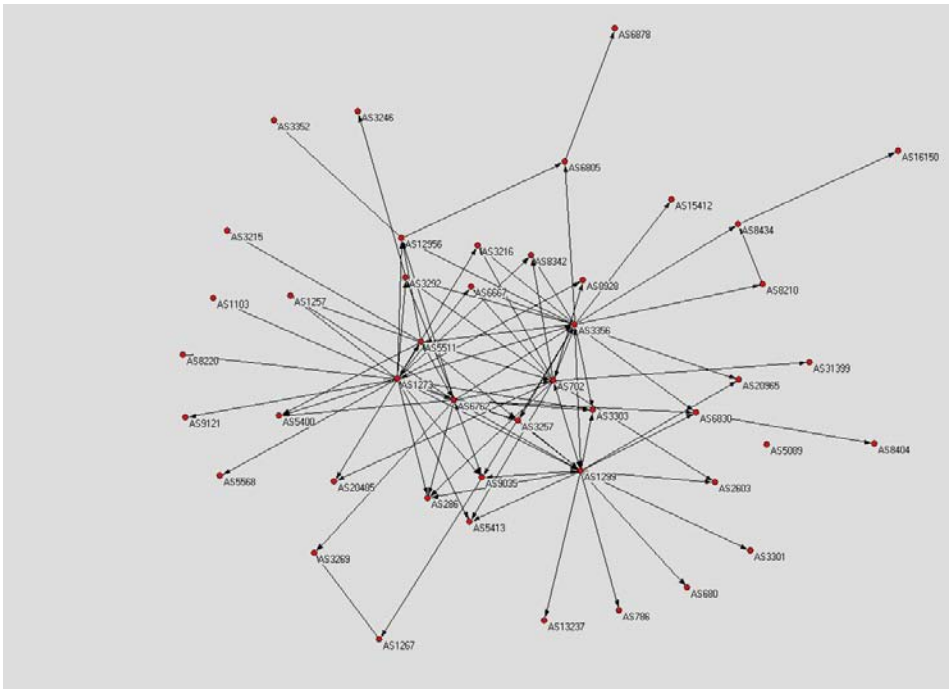
AS8404 - December 19, 2007



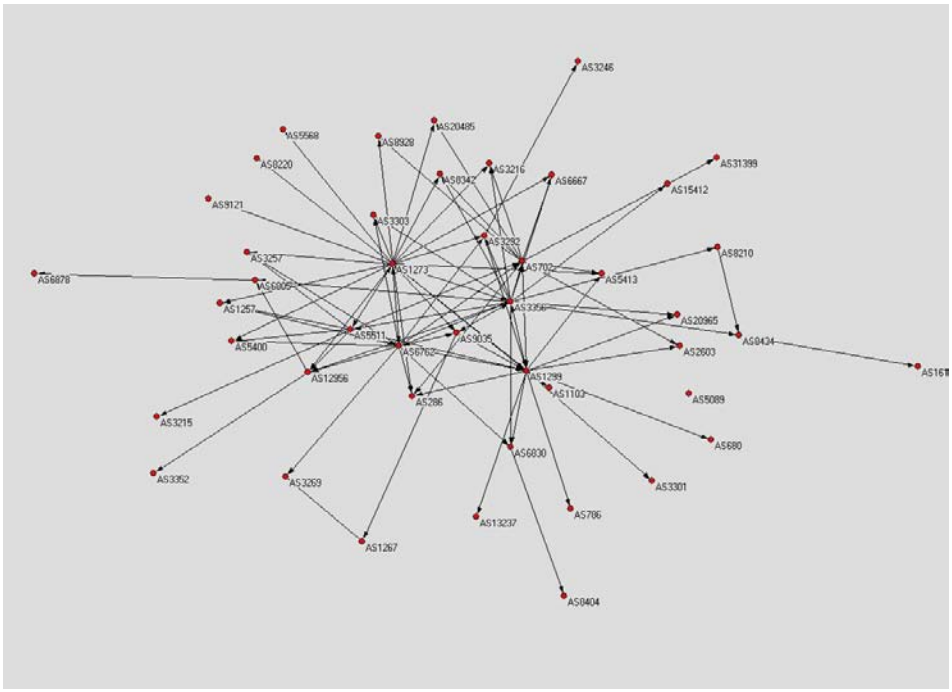
AS20485 - November 29, 2007



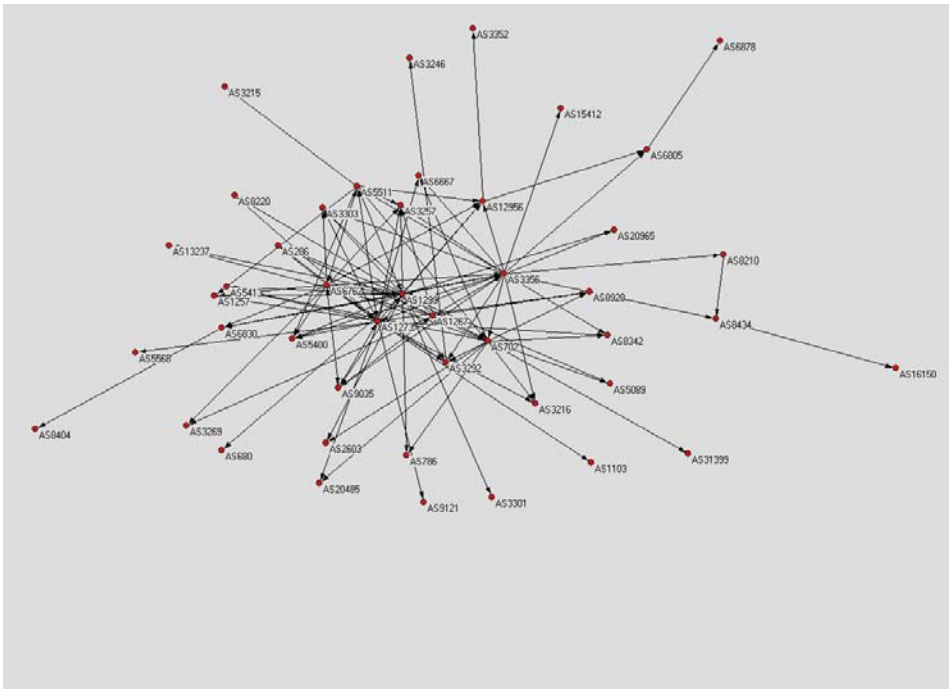
AS20485 - December 19, 2007



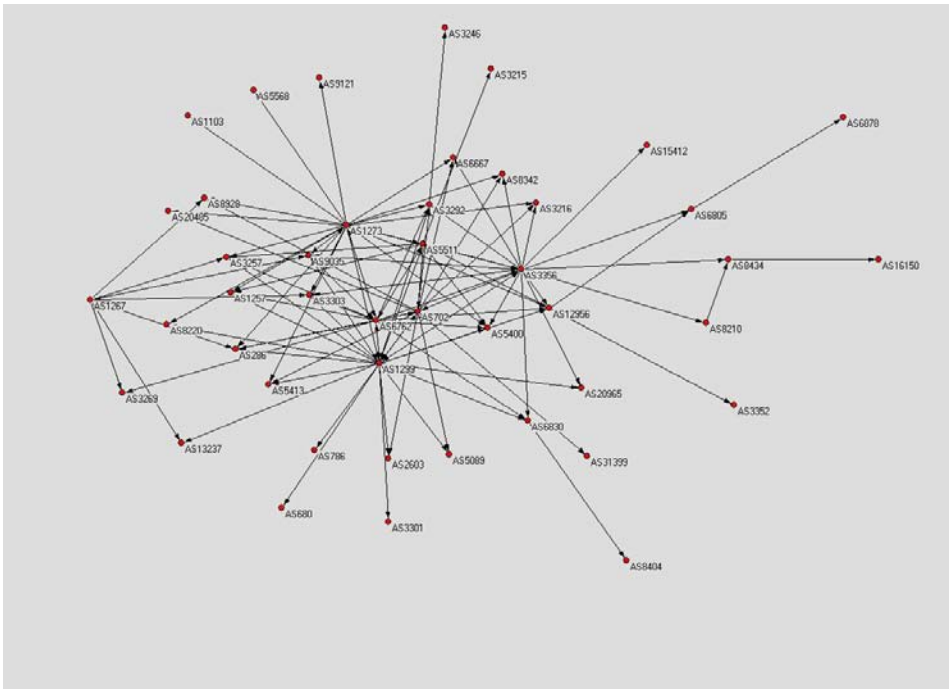
AS9035 - November 29, 2007



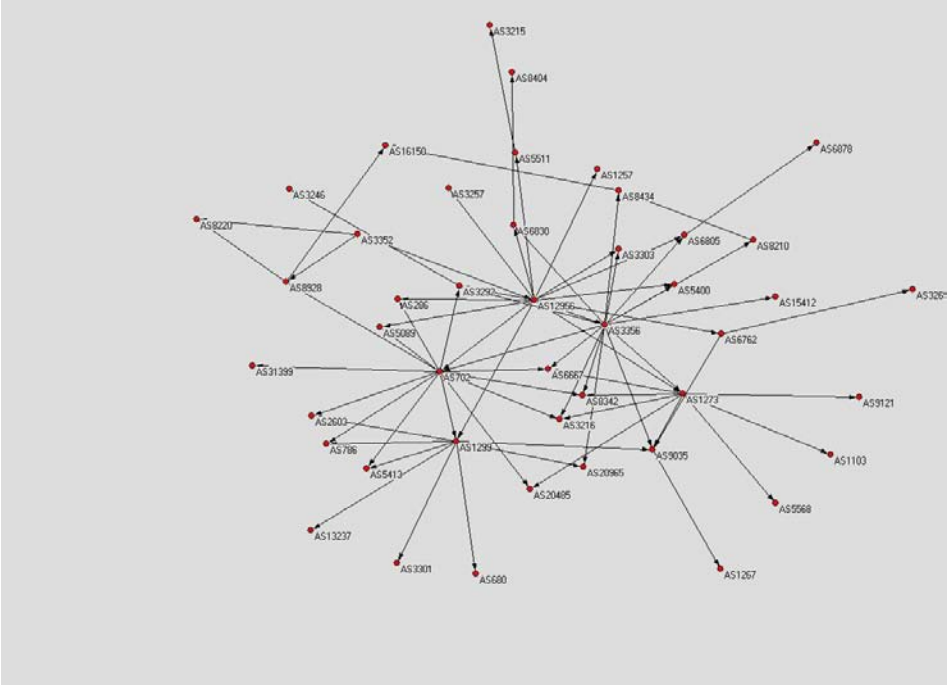
AS9035 - December 19, 2007



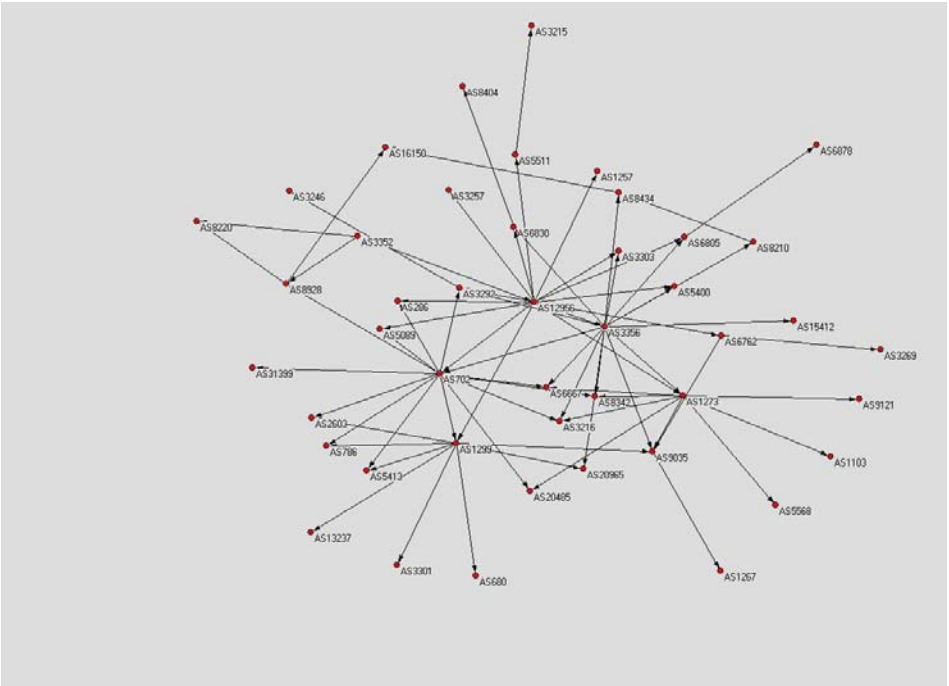
AS1267 - November 29, 2007



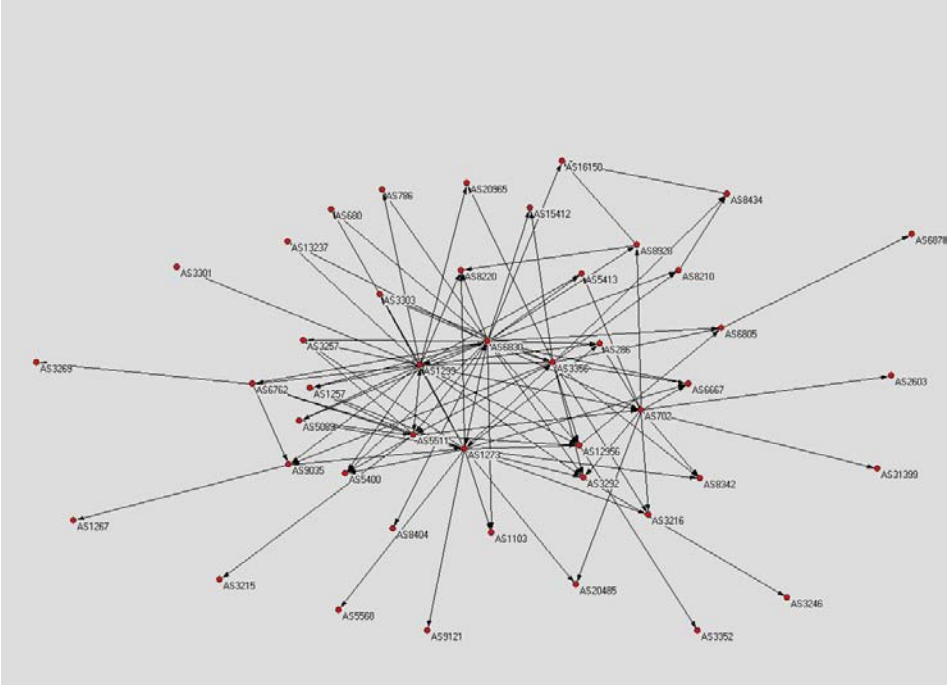
AS1267 - December 19, 2007



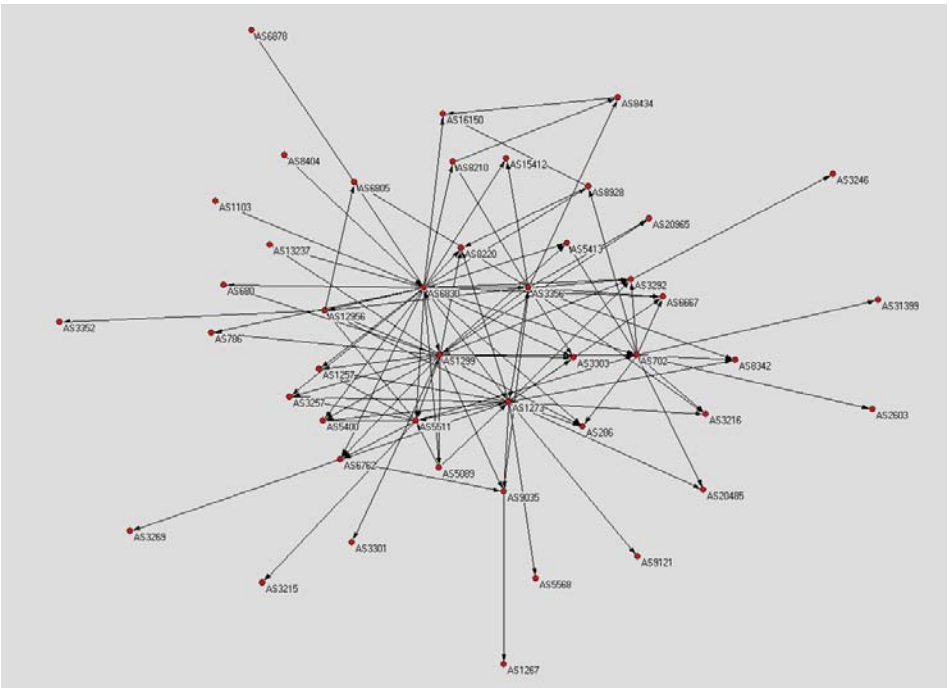
AS3352 - November 29, 2007



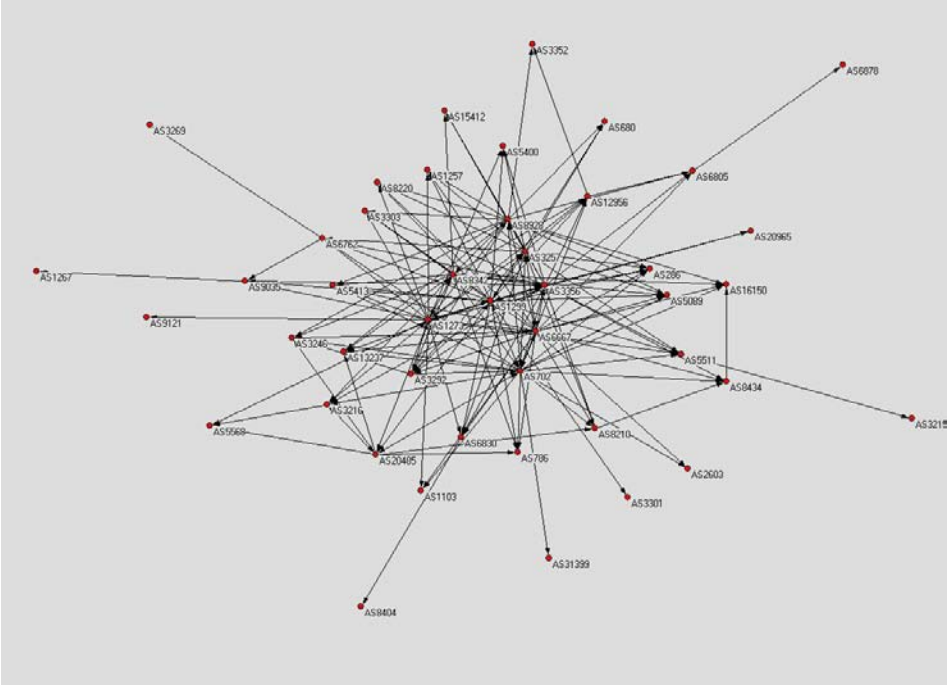
AS3352 - December 19, 2007



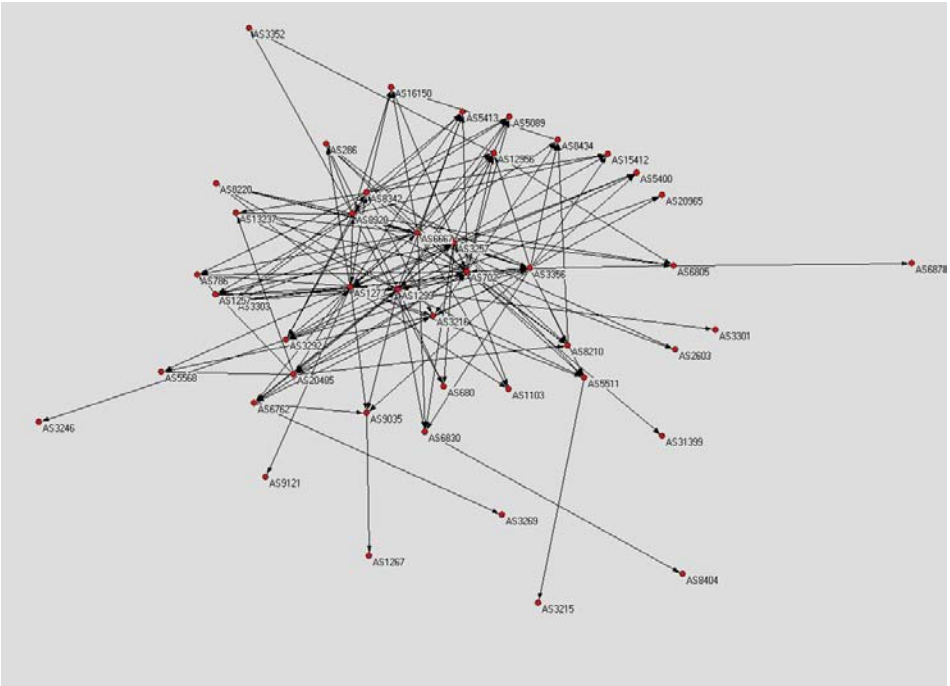
AS6830 - November 29, 2007



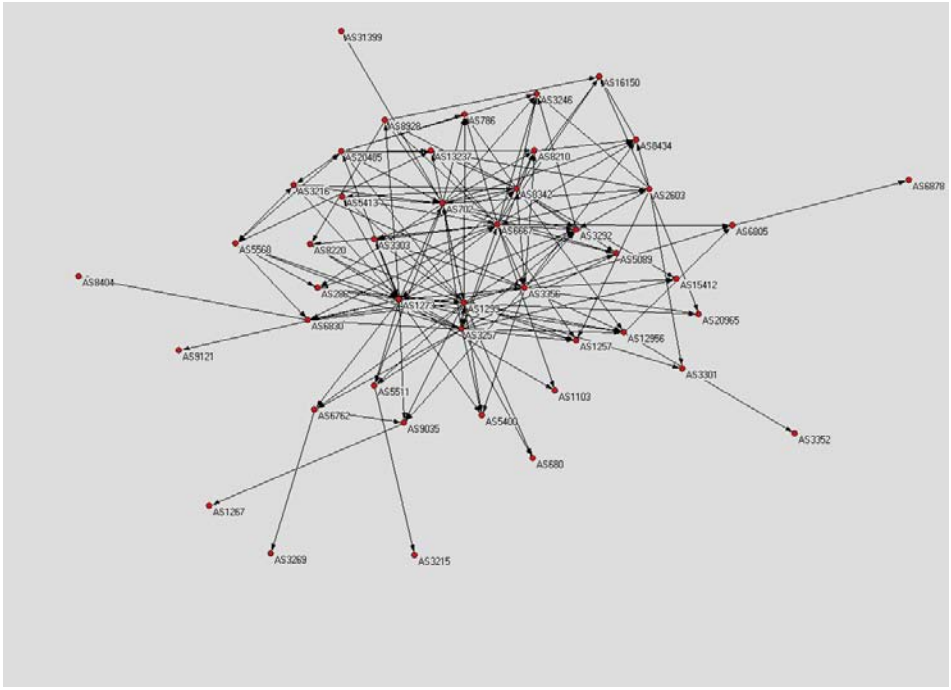
AS6830 - December 19, 2007



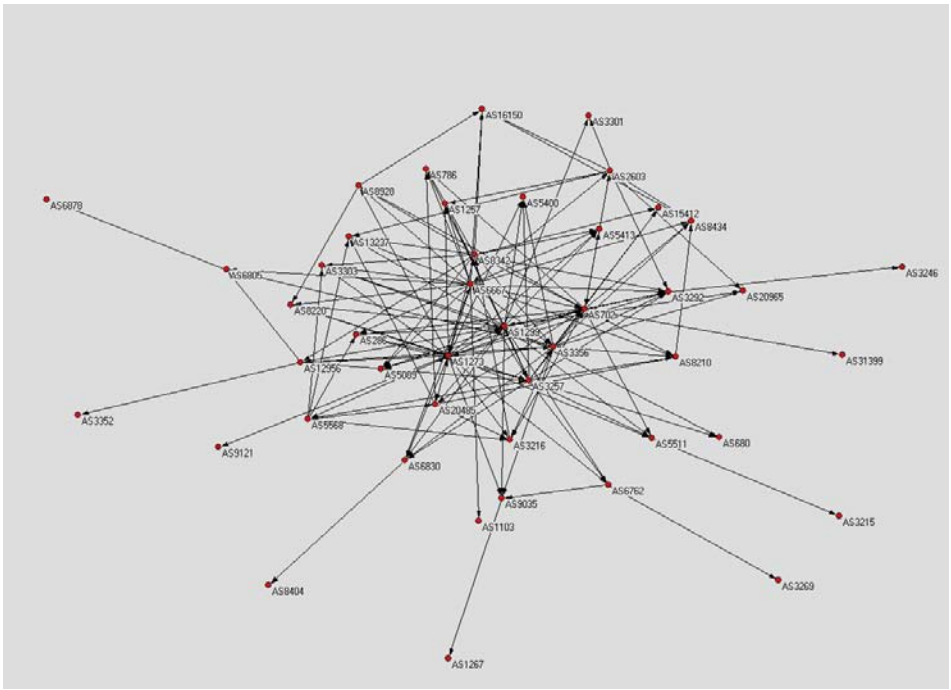
AS3216 - November 29, 2007



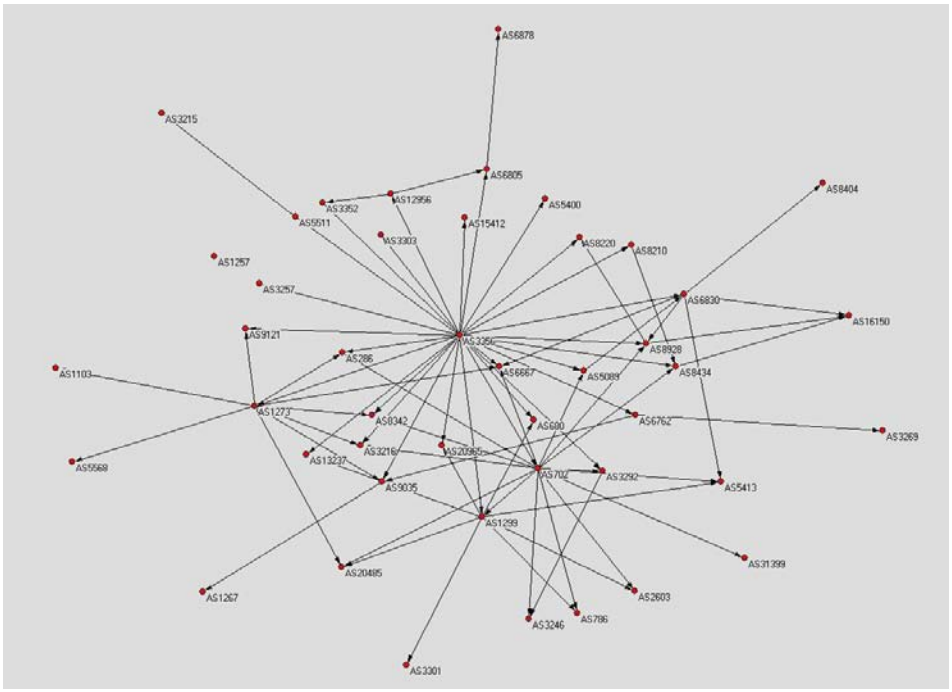
AS3216 - December 19, 2007



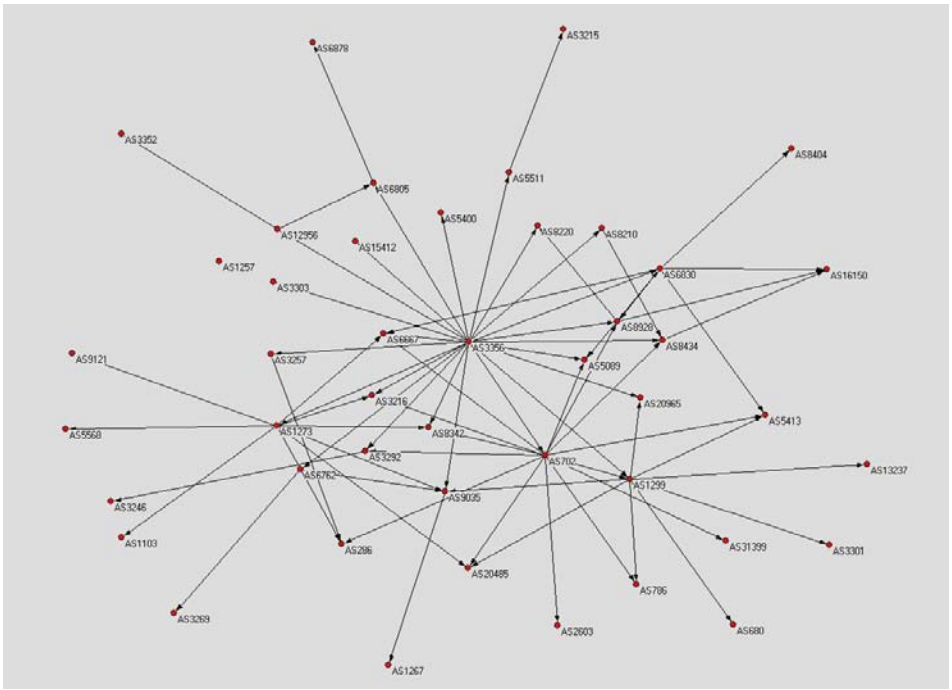
AS5568 - November 29, 2007



AS5568 - December 19, 2007



AS3356 - November 29, 2007



AS3356 - December 19, 2007

Appendix B

Scripts Used to Process Topological Data

This is the source code of the UNIX, Perl and Matlab scripts used in the experiment described in Chapter 4. A brief description of these scripts was provided in sections 4.5 and 4.6. Some scripts need the list of ASs to be processed. This list is provided in a text file named AS_list. Whenever a script calls for a list of ASs, it must be understood that the script is asking for this file.

B.1 UNIX script: get_as_data.sh

```
#!/bin/tcsh
#------------------------------------------------------------
# get_as_data.sh
#------------------------------------------------------------
# This script obtains the raw AS data from the RIPE Whois
# database.
#------------------------------------------------------------
# Inputs:
#   1. The AS list stored in the "AS_list" file and using
#      format AS<AS number>, e.g.: AS1234
# Outputs:
#   1. The raw data from each AS is stored in a file with
#      the same name followed by the extension .data. All
#      the files as stored in the "data" directory.
#------------------------------------------------------------
# Created by: David Arjona
# Date: 5/April/2007
#------------------------------------------------------------

#------------------------------------------------------------
# Note that we use backquotes for the cat command.
#------------------------------------------------------------
foreach as_num (`cat ./AS_list`)
   whois -h whois.ripe.net $as_num >data/$as_num.data
end
```

B.2 Perl script: DGPS.pl

```
#!/usr/bin/perl -w
#================================================================
#
# DGPS.pl (Destination Groups Perl script)
#
#================================================================
# This script has double functionality. When the script has no
# parameters (or the parameter is zero) both functionalities are
# activated. This is the default behavior for this script. When
# the parameters are equal to 1 or 2, only one of the following
# functionalities will be activated:
#
# 1. Creates Fundamental Destination Groups for a list of
# Autonomous Systems (AS).
# 2. Creates adjacency matrices for a list of destinations.
#
# Notes: Before running this script you may want to download
# fresh data from the RIPE Whois database by running the Unix
# script get_as_data.sh
#----------------------------------------------------------------
# Inputs:
# 1. List of ASs to be processed. This list should be
#    stored in text file "./AS_list". This file is also
#    used by Unix script get_as_data.sh.
# 2. Raw data files for each AS in the list. This files
#    must be stored in a "data" directory and named after
#    their corresponding AS followed by the extension
#    .data, e.g.:
#       ./data/AS123.data
#       ./data/AS4455.data
#    Notes: This is the same convention followed by the Unix
#    script get_as_data.sh.
# 3. An optional parameter which may be equal to 0, 1 or 2; this
#    activates respectivelly either all functionalities, just
#    the 1st or only the 2nd one.
#
# Outputs:
# 1. The  Fundamental Destination Groups are stored in file
#    "DG_CLASS.dump" when the 1st functionality is activated.
# 2. When the 2nd functionality is activated the adjacency
#    matrices are stored in several .out and .mat files in the
#   "data" directory.
#    Note: The .out files can be loaded in Matlab and the .mat
#    files can be loaded in Pajek.
# 3. The generic Adjacency Matrix is stored in file "DGPS.out".
# 4. Other messages, including warnings and errors are stored
#    in file "DGPS.err".
# 5. A list of the Import and Export groups are stored in file
#    "DGPS.dump".
#----------------------------------------------------------------
# Created by: David Arjona
# Date: 27/April/2007
#----------------------------------------------------------------

#----------------------------------------------------------------
# Classes are initialized here:
#    - Dest_Gp: This class stores and find the Fundamental
#         Groups from the Destination Groups that will be found
#         by subroutine Process_Destination_Groups().
#----------------------------------------------------------------
use DG_CLASS;
$Dest_Gp = DG_CLASS->new;
$Dest_Gp->disp_total;

#----------------------------------------------------------------
# Subroutines' prototypes are declared here. The implementations
# come after the main program.
#----------------------------------------------------------------
sub Process_Export_Line(@);
sub Process_Import_Line(@);
sub Process_Destination_Groups();
sub Process_Adjacency($);
sub Sanitize_Import_Group();
sub Sanitize_Export_Group();
sub Process_Community_Attr($@);
sub Dump_Adjacency_Matrix();
sub Dump_Exp_Imp_Groups();



#================================================================
################################################################
#================================================================
# Main Program starts here
#================================================================
################################################################
#================================================================


#----------------------------------------------------------------
#----------------------------------------------------------------
# Initilize Section:
#----------------------------------------------------------------
#----------------------------------------------------------------

#----------------------------------------------------------------
# When no option is provided we will used the default value 0.
#----------------------------------------------------------------
$option = shift;

if (! $option) {
   $option = 0;
}

#----------------------------------------------------------------
# File "DGPS.err" is the recipient of status, warnings and error
# messages.
#----------------------------------------------------------------
open(MESSAGES, ">DGPS.err") or die "Can't create DGPS.err file: $!\n";

#----------------------------------------------------------------
# AS_list is the file that contains the list of ASs to be
# processed. We will store this list in the array "ASlist"
# that has "length" number of elements or ASs.
#----------------------------------------------------------------
open(AS_LIST_FILE, "AS_list") or die "Can't open AS_list file: $!\n";

while ($ASnum = <AS_LIST_FILE>) {
   chomp($ASnum);
   $length = push(@ASlist, $ASnum);
}

close(AS_LIST_FILE);
print {"MESSAGES"} "Length of the AS list: $length\n";

#----------------------------------------------------------------
# matrixrow: this array or array contains the adjacency info
#    between the AS that is currently being processed and the
#    other ASs in "ASlist". Here we intialize it to zeros.
# Export_Groups: This is a hash of hash of arrays. The hash
#    table is for each AS and the arrays store the routes
#    exported to each AS.
# Import_Groups: This is a hash of hash of arrays. The hash
#    table is for each AS and the arrays store the routes
#    imported from each AS.
#----------------------------------------------------------------
for $x (0 .. ($length-1)) {
   for $y (0 .. ($length-1)) {
      $matrixrow[$x][$y] = 0;
   }
}

$Export_Groups{@ASlist}{@ASlist} = () x @ASlist x @ASlist;
$Import_Groups{@ASlist}{@ASlist} = () x @ASlist x @ASlist;
#----------------------------------------------------------------
#----------------------------------------------------------------
# Initilize Section Ends
#----------------------------------------------------------------
#----------------------------------------------------------------



#----------------------------------------------------------------
#----------------------------------------------------------------
# Main Destination Groups Loop:
#----------------------------------------------------------------
#----------------------------------------------------------------
# This loop processes the data files for each AS in the
# "ASlist". First, it opens the data file and searches for a
# connection between the current AS and the other ASs in the
# "ASlist". When a connection exists, the routing or destination
# information is stored in the Export_Groups and Import_Groups
# data structures so this information can be used later by the
# remaining procedures.
#----------------------------------------------------------------
for ($i=0; $i<$length; $i++) {
   #-------------------------------------------------------------
   # switch_1: this variable helps to identify when an export
   #    rule spans multiple lines. Initialy this variable is
   #    OFF=0, but it will be set to ON=1 when an export rule
   #    found. This variable is set back to off when the export
   #    rule finishes.
```

```
   # brckts_1: this variable helps to identify when an export
   #    rule spans multiple lines. This variable is set to 1 if
   #    an export rule opens a bracket ( { ), and it will be set
   #    back to zero when the bracket closes ( } ).
   #-------------------------------------------------------------

   $switch_1 = 0;
   $brckts_1 = 0;

   #-------------------------------------------------------------
   # The information for each AS is contained in the file
   # AS<number>.data, which is stored in the "data"
   # directory. For example:
   #   data/AS1299.data
   #   data/AS123.data
   #-------------------------------------------------------------
   $file1 = "data/" . $ASlist[$i] . ".data";
   print {"MESSAGES"} "\nCurrently processing $file1 :\n";

   open(EXPORT_FILE, "$file1") or die "Can't open $file1: $!\n";

   #-------------------------------------------------------------
   # Inner Loop:
   #-------------------------------------------------------------
   # Finds connections between the current AS and the other
   # ASs in the "ASlist".
   #-------------------------------------------------------------
   while ($line = <EXPORT_FILE>) {
      #----------------------------------------------------------
      # Store each word in "line" in the line array: "@larray".
      #----------------------------------------------------------
      @larray = split(" ", $line);

      #----------------------------------------------------------
      # "@larray" returns false on empty lines. This check
      # avoids warning messages on blank lines.
      #----------------------------------------------------------
      if (@larray) {
         #-------------------------------------------------------
         # We are looking for an "export" line. When we find it
         # we delete the "export:" part and set the "$switch" to
         # true (1).
         #-------------------------------------------------------
         if ($larray[0] eq "export:") {
            shift(@larray);
            $switch_1 = 1;
         }

         #-------------------------------------------------------
         # We enter inside this if statement if we just found
         # an export rule or if we are still processing an
         # export rule that spans multiple lines.
         #-------------------------------------------------------
         if ($switch_1) {
            $switch_1 = Process_Export_Line(@larray);
         }
      }
   }
   #-------------------------------------------------------------
   # Inner Loop Ends
   #-------------------------------------------------------------

   #-------------------------------------------------------------
   # Sanitize the Export_Groups and Import_Groups data structure
   # before continuing the processing.
   #-------------------------------------------------------------
   Sanitize_Export_Group();
   Sanitize_Import_Group();

   #-------------------------------------------------------------
   # Display the Adjacency Matrix in the output screen.
   #-------------------------------------------------------------
   printf("%8s | @{$matrixrow[$i]} |\n", $ASlist[$i]);
}

#---------------------------------------------------------------
# Output the generic Adjacency Matrix (file DGPS.out) and the
# list of the Import and Export groups (file DGPS.dump).
#---------------------------------------------------------------
Dump_Adjacency_Matrix();
Dump_Exp_Imp_Groups();

#---------------------------------------------------------------
#---------------------------------------------------------------
# Main Destination Groups Loop Ends
#---------------------------------------------------------------
#---------------------------------------------------------------

#---------------------------------------------------------------
#---------------------------------------------------------------
# Further Processing Section:
#---------------------------------------------------------------
#---------------------------------------------------------------

#---------------------------------------------------------------
# The 1st functionality (creating Fundamental Destination
# Groups) is processed here.
#---------------------------------------------------------------
if (($option == 0) || ($option == 1)) {
   Process_Destination_Groups();
   $Dest_Gp->dump_all_DG;
   $Dest_Gp->disp_total;
}

#---------------------------------------------------------------
# The 2nd functionality (generating adjacency matrices for a
# list of destinations) is processed here.
#---------------------------------------------------------------
if (($option == 0) || ($option == 2)) {
   foreach $the_seed (@ASlist) {
      Process_Adjacency($the_seed);
   }
}

#---------------------------------------------------------------
#---------------------------------------------------------------
# Further Processing Section Ends
#---------------------------------------------------------------
#---------------------------------------------------------------



close(MESSAGES);
print "\n *** DGPS.pl Finished! ***\n";

#===============================================================
################################################################
#===============================================================
# Main Program finishes here
#===============================================================
################################################################
#===============================================================




#===============================================================
#
# Funtion: Process_Export_Line
#
#===============================================================
# This subroutine checks if the current export rule applies to
# any of the ASs included in the original AS list. When a
# desired AS is found, we also check if the import rule is also
# valid using the Process_Import_Line function.
#---------------------------------------------------------------
# Inputs:
# 1. The export line without the "export:" word.
#---------------------------------------------------------------
# Outputs:
# 1. A "0" is returned if we have found a valid destination
#    group, otherwise we return a "1" which means that we should
#    process the next line.
#---------------------------------------------------------------

sub Process_Export_Line(@) {

#---------------------------------------------------------------
# When the first element in the $_ array is "{" the export
# rules will follow in the subsequent lines.
#---------------------------------------------------------------
if    ($_[0] eq "{") {
   print {"MESSAGES"}
   " Warning: A bracket ({) has been opened in the export file.\n";
   $brckts_1 = 1;
   return 1;
}

#---------------------------------------------------------------
# When the first element in the $_ array is "}" no more export
# rules  will follow.
```

```
#-------------------------------------------------------------
elsif ($_[0] eq "}") {
   print {"MESSAGES"}
   " Warning: A bracket (}) has been closed in the export file.\n";
   $brckts_1 = 0;
   return 0;
}

#-------------------------------------------------------------
# When the first element in the "$_" array is "to" the export
# rules  are contained in this line (and maybe the following
# ones). There is also the posibility that the first element in
# the "$_" array is "+".
#-------------------------------------------------------------
elsif ($_[0] eq "to") # ||
#        ($_[0] eq "+")  ) {
   #----------------------------------------------------------
   # For Loop
   #----------------------------------------------------------
   # Check if the current export rule refers to any of the ASs
   # in the list.
   #----------------------------------------------------------
   for ($j=0; $j<$length; $j++) {
      if ($_[1] eq $ASlist[$j]) {
         #-------------------------------------------------
         # We have found a matching AS:
         # Now process the corresponding import rule for the
         # matched AS.
         #-------------------------------------------------
         # switch_2: this variable helps to identify when an
         #    import rule spans multiple lines, just like the
         #    similar "switch_1" variable.
         # brckts_2: this variable helps to identify when an
         #    import rule spans multiple lines, just like the
         # match: this variable allow us to exit the while loop
         #    when a matching importing rule is found.
         #    back to zero when the bracket closes ( } ).
         # file2: this variable stores the name of the data
         #    file that must contain the import rule.
         #-------------------------------------------------
         $switch_2 = 0;
         $brckts_2 = 0;
         $match    = 0;

         $file2 = "data/" . $ASlist[$j] . ".data";

         open(IMPORT_FILE, "$file2") or die "Can't open $file2: $!\n";


         #-------------------------------------------------
         # File Loop:
         #-------------------------------------------------
         # Finds connections between the current AS and the other
         # ASs in the "ASlist".
         #-------------------------------------------------
         while ($import_line = <IMPORT_FILE>) {
            #----------------------------------------------
            # Store each word in "import_line" in the line
            # array: "@import_array".
            #----------------------------------------------
            @import_array = split(" ", $import_line);

            #----------------------------------------------
            # "@import_array" returns false on empty lines. This
            # check avoids warning messages on blank lines.
            #----------------------------------------------
            if (@import_array) {
               #-------------------------------------------
               # We are looking for an "import" line. When we
               # find it we delete the "import:" part and set
               # the "$switch" to true (1).
               #-------------------------------------------
               if ($import_array[0] eq "import:") {
                  shift(@import_array);
                  $switch_2 = 1;
               }

               #-------------------------------------------
               # We enter inside this if statement if we find
               # an import rule or if we are still processing an
               # import rule that spans multiple lines.
               #-------------------------------------------
               if ($switch_2) {
                  $switch_2 = Process_Import_Line(@import_array);
               }
            }
         }
         #-------------------------------------------------
         # File Loop Ends
         #-------------------------------------------------

         #-------------------------------------------------
         # Find the Exporting Group
         # But only if we have found a match from the importing
         # AS.
         #-------------------------------------------------
         if ($match) {
            #----------------------------------------------
            # annfnd: This variable is set to ON (1) when the
            #    "announce" word is found.
            # push_1: This variable is set to ON (1) if a route
            #    group has been pushed into "%Export_Groups".
            #----------------------------------------------
            $annfnd = 0;
            $push_1 = 0;

            for ($k=2; $k<=$#_; $k++) {
               if ($annfnd) {
                  if ($brckts_1) {
                     #-----------------------------------
                     # When the data file uses brackets, the
                     # lines finish with a semicolon (;). Remove
                     # it before entering the data.
                     #-----------------------------------
                     $string = $_[$k];
                     $string =~ s/;//;
                     push( @{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} },
                           $string
                     );
                  }
                  else {
                     push( @{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} },
                           "$_[$k]"
                     );
                  }

                  $push_1 = 1;
               }
               elsif ($_[$k] eq "announce") {
                  $annfnd = 1;
               }
            }

            #----------------------------------------------
            # If we haven't pushed yet it means that the export
            # groups are probably in the following line.
            #----------------------------------------------
            # Note:
            # This instruction checks if "%Export_Groups" is
            # empty: if ( $#{ $Export_Groups{$ASlist[$j]} } eq -1)
            #----------------------------------------------
            if (! $push_1) {
               return 1;
            }
         }
         #-------------------------------------------------
         # We have found a matching AS but we are still inside
         # the For loop. We need to exit from this loop because
         # there is no point on continue searching for another
         # AS.
         #-------------------------------------------------
         if ($brckts_1) {
            return 1;
         }
         else {
            return 0;
         }
      }
   }
   #----------------------------------------------------------
   # For Loop Ends
   #----------------------------------------------------------

   #----------------------------------------------------------
   # When "$brckts_1" is ON we should return 1 so we can
   # continue analyzing exporting rules. Otherwise we just
   # return zero.
   #----------------------------------------------------------
   if ($brckts_1) {
      return 1;
   }
   else {
      return 0;
   }
```

```
}

#-------------------------------------------------------------
# When the first element in the $_ array is "+" it means
# that the export rule started in the previous line but the
# exporting groups are in the current or next line.
#-------------------------------------------------------------
elsif ($_[0] eq "+") {
   #----------------------------------------------------------
   # Find the Exporting Group
   #----------------------------------------------------------
   for ($k=1; $k<=$#_; $k++) {
      if ($annfnd) {
         push( @{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} }, "$_[$k]");
         $push_1 = 1;
      }
      elsif ($_[$k] =~ /announce\b/) {
         $annfnd = 1;
      }
   }

   #----------------------------------------------------------
   # If we have NOT pushed any data
   # we must return 1 so we can continue the analysis of
   # exporting rules.
   #----------------------------------------------------------
   if ($push_1) {
      return 0;
   }
   else {
      return 1;
   }
}

#-------------------------------------------------------------
# When the first element in the $_ array is "announce" it means
# that the export rule started in the previous line but the
# exporting groups are in the current line.
#-------------------------------------------------------------
elsif ($_[0] eq "announce") {
   for ($k=1; $k<=$#_; $k++) {
      push( @{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} }, "$_[$k]");
    }
    return 0;
}

#-------------------------------------------------------------
# If everything fails we have an export rule which we have not
# found before or we have commited a mistake.
#-------------------------------------------------------------
else {
   print {"MESSAGES"} "\n ERROR: This is an odd export rule!!!\n";
   print {"MESSAGES"} "     *** @_ ***\n";

   #----------------------------------------------------------
   # When "$brckts_1" is ON we should return 1 so we can
   # continue analyzing exporting rules. Otherwise we just
   # return zero.
   #----------------------------------------------------------
   if ($brckts_1) {
      return 1;
   }
   else {
      return 0;
   }
}

}
#==============================================================
# Process_Export_Line Ends
#==============================================================




#==============================================================
#
# Funtion: Process_Import_Line
#
#==============================================================
# This subroutine checks if the current import rule applies to
# the AS that originated the current export rule.
#--------------------------------------------------------------
# Inputs:
# 1. The import line without the "import:" word.
#--------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned if a match has been found, otherwise
#    we only return a zero (False).
#--------------------------------------------------------------

sub Process_Import_Line(@) {

#--------------------------------------------------------------
# accpt: This variable is set to ON (1) when the "accept" word
#         is found.
# push_2: This variable is set to ON (1) if a route group has
#         been pushed into "%Import_Groups".
#--------------------------------------------------------------
my $accpt  = 0;
my $push_2 = 0;


#--------------------------------------------------------------
# IF statement:
#--------------------------------------------------------------
# When the first element in the $_ array is "{" the import
# rules will follow in the subsequent lines.
#--------------------------------------------------------------
if    ($_[0] eq "{") {
   print {"MESSAGES"}
   "   Warning:  Open bracket ({) in the import file.\n";
   $brckts_2 = 1;
   return 1;
}

#--------------------------------------------------------------
# When the first element in the $_ array is "}" no more import
# rules  will follow.
#--------------------------------------------------------------
elsif ($_[0] eq "}") {
   print {"MESSAGES"}
   "   Warning: Close bracket (}) in the import file.\n";
   $brckts_2 = 0;
   return 0;
}

#--------------------------------------------------------------
# When the first element in the "$_" array is "from" the import
# rules  are contained in this line (and maybe the following
# ones). There is also the posibility that the first element in
# the "$_" array is "+".
#--------------------------------------------------------------
elsif ( ($_[0] eq "from") ||
        ($_[0] eq "From") )
#        ($_[0] eq "+")  ) {
   if ($_[1] eq $ASlist[$i]) {
      print {"MESSAGES"} "    - Connected to $ASlist[$j]\n";
      $matrixrow[$i][$j] = 1;

      #-----------------------------------------------------
      # Find the Importing Group
      #-----------------------------------------------------
      for ($k=2; $k<=$#_; $k++) {
         if ($accpt) {
            if ($brckts_2) {
               #-----------------------------------------
               # When the data file uses brackets, the lines
               # finish with a semicolon (;). Remove it before
               # entering the data.
               #-----------------------------------------
               $string = $_[$k];
               $string =~ s/;//;
               push( @{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} },
                     $string
               );
            }
            else {
               push( @{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} },
                     "$_[$k]"
               );
            }
            $push_2 = 1;
            $match  = 1;
         }
         elsif ($_[$k] =~ /accept\b/) {
            $accpt = 1;
         }
      }

      #-----------------------------------------------------
      # If we have NOT pushed any data we must return 1.
      #-----------------------------------------------------
```

```
      if (! $push_2) {
         return 1;
      }

   }

   #-------------------------------------------------------------
   # When "$brckts_2" is ON we must return 1 so we can continue
   # the analysis of exporting rules.
   #-------------------------------------------------------------
   if ($brckts_2) {
      return 1;
   }
   else {
      return 0;
   }
}

#-------------------------------------------------------------------
# When the first element in the $_ array is "action" OR "+" it
# means that the import rule started in the previous line and it
# may or may not continue in the following line.
#-------------------------------------------------------------------
elsif ( ($_[0] eq "action") ||
        ($_[0] eq "+")      ) {
   #-------------------------------------------------------------
   # Find the Importing Group
   #-------------------------------------------------------------
   for ($k=1; $k<=$#_; $k++) {
      if ($accpt) {
         push( @{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} }, "$_[$k]");
         $push_2 = 1;
         $match  = 1;
      }
      elsif ($_[$k] =~ /accept\b/) {
         $accpt = 1;
      }
   }

   #-------------------------------------------------------------
   # If we have NOT pushed any data
   # we must return 1 so we can continue the analysis of
   # exporting rules.
   #-------------------------------------------------------------
   if ($push_2) {
      return 0;
   }
   else {
      return 1;
   }
}

#-------------------------------------------------------------------
# When the first element in the $_ array is "accept" it means
# that the import rule started in the previous line but the
# importing groups are in the current line.
#-------------------------------------------------------------------
elsif ($_[0] eq "accept") {
   for ($k=1; $k<=$#_; $k++) {
      push( @{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} }, "$_[$k]");
   }

   $match = 1;
   return 0;
}

#-------------------------------------------------------------------
# If everything fails we have an import rule which we have not
# found before or we have commited a mistake.
#-------------------------------------------------------------------
else {
   print {"MESSAGES"} "\n ERROR: This is an odd import rule!!!\n";
   print {"MESSAGES"} "     *** @_ ***\n";

   #-------------------------------------------------------------
   # When "$brckts_2" is ON we should return 1 so we can
   # continue analyzing exporting rules. Otherwise we just
   # return zero.
   #-------------------------------------------------------------
   if ($brckts_2) {
      return 1;
   }
   else {
      return 0;
   }
}
}
#============================================================
# Process_Import_Line Ends
#============================================================




#============================================================
#
# Funtion: Process_Adjacency
#
#============================================================
# This subroutine creates the adjacency matrix for the AS passed
# as an argument (seed value).
#
#------------------------------------------------------------
# Inputs:
# 1. The seed value ($_[0]).
#------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned by default.
# 2. A .out output file is generated to load an adjacency matrix
#    into Matlab.
# 3. A .mat output file is generated to load an adjacency matrix
#    into Pajek.
#------------------------------------------------------------

sub Process_Adjacency($) {

my $seed  = shift;

print {"MESSAGES"} "\nProcessing Adjacency for $seed:\n";
print "Processing Adjacency for $seed.\n";

#------------------------------------------------------------
# The returned adjacency information for each AS is stored in
# the files $seed.out and $seed.mat, which are stored in the
# "data" directory For example:
#   data/AS1299.out
#   data/AS1299.mat
#------------------------------------------------------------
$ofile = "data/" . $seed . ".out";
$pajek = "data/" . $seed . ".mat";

open(ADJ_FILE, ">$ofile") or die "Can't open $ofile: $!\n";
open(PJK_FILE, ">$pajek") or die "Can't open $pajek: $!\n";

#------------------------------------------------------------
# Add additional info for the Pajek .mat file:
#------------------------------------------------------------
print {"PJK_FILE"} "*Vertices $length\n";

for ($i=1; $i<=$length; $i++) {
   print {"PJK_FILE"} " $i \"$ASlist[$i-1]\" 0.1 0.2 0.5\n";
}

print {"PJK_FILE"} "*Matrix\n";

#------------------------------------------------------------
# Main Loop:
#------------------------------------------------------------
# Processes the destinations for each AS in the "ASlist".
#------------------------------------------------------------
for ($i=0; $i<$length; $i++) {
   MAIN_LOOP: for ($j=0; $j<$length; $j++) {
      #---------------------------------------------------
      # If there is no adjacency from the Export/Import rules
      # we do not need to continue processing this data.
      #---------------------------------------------------
      if (! $matrixrow[$i][$j]) {
         print {"ADJ_FILE"} "0 ";
         print {"PJK_FILE"} "0 ";
         next MAIN_LOOP;
      }

      $exp_lgth = $#{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} };
      $imp_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} };

      #---------------------------------------------------
      # When the Import and Export Groups are both equal to ONLY
      # "ANY", we assume that the "$seed" value passes between
      # the two ASs.
      #---------------------------------------------------
      if ( ($exp_lgth == 0)   &&   ($imp_lgth == 0)   &&
           ( ($Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /ANY/)  ||
```

```
            ($Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /Any/)
          ) &&
          ( ($Import_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /ANY/) ||
            ($Import_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /Any/)
          ) ) {
         print {"ADJ_FILE"} "1 ";
         print {"PJK_FILE"} "1 ";
         next MAIN_LOOP;
      }

      #------------------------------------------------------------
      # When the Import and Export Groups are equal and of
      # length 1, we just use the Destination Group common to
      # both, and check if they are passing the "$seed" value.
      #------------------------------------------------------------
      if ( ($exp_lgth == 0) && ($imp_lgth == 0) &&
           ($Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0]  eq
            $Import_Groups{$ASlist[$i]}{$ASlist[$j]}[0])
         ) {
         $adj_value = $Dest_Gp->find_seed(
                         $seed,
                         $Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] );

         print {"ADJ_FILE"} "$adj_value ";
         print {"PJK_FILE"} "$adj_value ";
         next MAIN_LOOP;
      }

      #------------------------------------------------------------
      # Let's process the Export Groups: If we find a
      # destination that contains the seed we then process the
      # Import Groups.
      #------------------------------------------------------------
      foreach $DG_1 (@{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} }) {
         if ($Dest_Gp->find_seed( $seed, $DG_1 )) {
            #------------------------------------------------------
            # Let's process the Import Groups: If we find a
            # destination that contains the seed we assume that
            # the "$seed" value is passing between these ASs.
            #------------------------------------------------------
            foreach $DG_2 (@{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} }) {
               if ($Dest_Gp->find_seed( $seed, $DG_2 )) {
                  print {"ADJ_FILE"} "1 ";
                  print {"PJK_FILE"} "1 ";
                  next MAIN_LOOP;
               }
            }
         }
      }

      #------------------------------------------------------------
      # If we get to this line, the "$seed" value is not being
      # passed between the ASs.
      #------------------------------------------------------------
      print {"ADJ_FILE"} "0 ";
      print {"PJK_FILE"} "0 ";
   }

   print {"ADJ_FILE"} "\n";
   print {"PJK_FILE"} "\n";
}
#------------------------------------------------------------------
# Main Loop Ends
#------------------------------------------------------------------

close(ADJ_FILE);
close(PJK_FILE);

return 1;
}
#==================================================================
# Process_Adjacency Ends
#==================================================================




#==================================================================
#
# Funtion: Process_Destination_Groups
#
#==================================================================
# This subroutine processes the information stored at the
# Export_Groups and Import_Groups data structures. Hopefully,
# this data structures store valid destinations that will be
# used to find Destination Groups and Fundamental Groups when
# we call the method "add_DG" from the class "Dest_Gp".
#------------------------------------------------------------------
# Inputs:
# 1. None
#------------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned by default.
#------------------------------------------------------------------

sub Process_Destination_Groups() {

#------------------------------------------------------------------
# Main Loop:
#------------------------------------------------------------------
# Processes the destinations for each AS in the "ASlist".
#------------------------------------------------------------------
for ($i=0; $i<$length; $i++) {
   print {"MESSAGES"}
       "\nProcessing Destination Groups for $ASlist[$i]:\n";
   print "Processing Destination Groups for $ASlist[$i].\n";

   for ($j=0; $j<$length; $j++) {

      $exp_lgth = $#{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} };
      $imp_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$j]} };

      #---------------------------------------------------------
      # If any of the groups is empty, we cannot build
      # destination groups so we skip the processing of the
      # current AS.
      #---------------------------------------------------------
      if (($exp_lgth == -1)  || ($imp_lgth == -1)) {
         next;
      }

      #---------------------------------------------------------
      # When the Import and Export Groups are both equal to ONLY
      # "ANY", we cannot create destination groups so we skip
      # the processing.
      #---------------------------------------------------------
      if ( ($exp_lgth == 0)   &&   ($imp_lgth == 0)   &&
           ( ($Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /ANY/) ||
             ($Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /Any/)
           ) &&
           ( ($Import_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /ANY/) ||
             ($Import_Groups{$ASlist[$i]}{$ASlist[$j]}[0] =~ /Any/)
           ) ) {
         printf {"MESSAGES"}
            "%11s: Both destination groups are equal to ANY\n",
            $ASlist[$j];
         next;
      }

      #---------------------------------------------------------
      # When the Import and Export Groups are equal and of
      # length 1, we just need add the Destination Group common
      # to both.
      #---------------------------------------------------------
      if ( ($exp_lgth == 0) && ($imp_lgth == 0) &&
           ($Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0]  eq
            $Import_Groups{$ASlist[$i]}{$ASlist[$j]}[0])
         ) {
         #------------------------------------------------------
         # Add the contents of the Export (or Import) Group
         # to the list of Destination Groups and compute the
         # Fundamental Groups that results from adding them.
         #------------------------------------------------------
         $Dest_Gp->add_DG( $Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] );

         printf {"MESSAGES"}
            "%11s: $Export_Groups{$ASlist[$i]}{$ASlist[$j]}[0] equal\n",
            $ASlist[$j];
         next;
      }

      #---------------------------------------------------------
      # Let's store the Export_Groups in a temporary
      # destinatons' array.
      #---------------------------------------------------------
      @Temporary_DG = @{ $Export_Groups{$ASlist[$i]}{$ASlist[$j]} };

      #---------------------------------------------------------
      # Let's process the Import Groups.
      #---------------------------------------------------------
      IMPORT_LOOP: for ($k=0; $k<=$imp_lgth; $k++) {
         for ($l=0; $l<=$exp_lgth; $l++) {
            if ( $Temporary_DG[$l] eq
```

```
                $Import_Groups{$ASlist[$i]}{$ASlist[$j]}[$k]
             ) {
             next IMPORT_LOOP;
          }
       }
       #-----------------------------------------------------
       # We have found a destination that it is in the Import
       # Groups but not in the temporary destinations' array.
       #-----------------------------------------------------
       push(@Temporary_DG, $Import_Groups{$ASlist[$i]}{$ASlist[$j]}[$k]);
    }

    if (@Temporary_DG) {
       #-----------------------------------------------------
       # Add the contents of the temporary destinations' array
       # to the list of Destination Groups and compute the
       # Fundamental Groups that results from adding them.
       #-----------------------------------------------------
       $Dest_Gp->add_DG( @Temporary_DG );

       printf {"MESSAGES"} "%11s: @Temporary_DG\n", $ASlist[$j];
    }
    else {
       print {"MESSAGES"}
          " Warning: Trying to process an empty destination.\n"
    }
  }
}
#-------------------------------------------------------------
# Main Loop Ends
#-------------------------------------------------------------

return 1;
}
#=============================================================
# Process_Destination_Groups Ends
#=============================================================
```

```
#=============================================================
#
# Funtion: Sanitize_Export_Group
#
#=============================================================
# This subroutine sanitizes the Export_Groups data structure
# before processing the data.
#-------------------------------------------------------------
# Inputs:
# 1. None
#-------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned by default.
#-------------------------------------------------------------

sub Sanitize_Export_Group() {

  for ($n=0; $n<$length; $n++) {
    $array_lgth = $#{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} };

    #-----------------------------------------------------
    # If the group is empty, continue to the next one.
    #-----------------------------------------------------
    if ($array_lgth == -1) {
       next;
    }

    for ($p=0; $p<=$array_lgth; $p++) {
       #-----------------------------------------------------
       # The occurrence of "community.contains(..)" is solved
       # through the function "Process_Community_Attr".
       #-----------------------------------------------------
       if ($Export_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] =~ /community\.contains/) {
          @{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} } =
             Process_Community_Attr( $ASlist[$i],
                                     @{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} }
             );
          $array_lgth = $#{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} };
       }

       #-----------------------------------------------------
       # Modify the occurrence of:
       #   <X> AND NOT AS8434:RS-MARTIANS
       # to just <X>.
       #-----------------------------------------------------
       if ( ($Export_Groups{$ASlist[$i]}{$ASlist[$n]}[$p]   eq "AND")   &&
            ($Export_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+1] eq "NOT")   &&
            ($Export_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+2] =~ /AS8434/ )  )
       {
          splice( @{$Export_Groups{$ASlist[$i]}{$ASlist[$n]}}, $p, 3);
          $array_lgth = $#{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} };
          $p--;
       }
       # print {"MESSAGES"}
       "@{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} } \n";
    }


    #-----------------------------------------------------
    # Eliminate duplicated information in the Export Group.
    #-----------------------------------------------------
    for ($p=0; $p<=$array_lgth; $p++) {
       for ($q=($p+1); $q<=$array_lgth; $q++) {
          if ( $Export_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] eq
               $Export_Groups{$ASlist[$i]}{$ASlist[$n]}[$q] ) {
             splice( @{$Export_Groups{$ASlist[$i]}{$ASlist[$n]}}, $q, 1);
             $array_lgth = $#{ $Export_Groups{$ASlist[$i]}{$ASlist[$n]} };
             $q--;
          }
       }
    }
  }

  return 1;
}
#=============================================================
# Sanitize_Export_Group Ends
#=============================================================
```

```
#=============================================================
#
# Funtion: Process_Community_Attr
#
#=============================================================
# This subroutine processed a line in the Export_Groups data
# structure when this line contains an export rule of the kind:
#    "community.contains(20965:155, ..., 20965:21320)"
# In order to process this kind of export rule we must read
# again from the data file and find which ASs are included in
# the corresponding community.
#-------------------------------------------------------------
# Inputs:
# 1. The AS we are processing now. We need this information so
#    we can read from the correct data file.
# 2. The community information contained in the line that is
#    currently being processed.
#-------------------------------------------------------------
# Outputs:
# 1. An array that contains the results of processing this
#    information and that should replace the current values
#    stored in the Export_Groups data structure.
#-------------------------------------------------------------

sub Process_Community_Attr($@) {
   my ($community_file, @attr_list) = @_;
   my $rtrn_array = ();

   #-----------------------------------------------------
   # The first element in the returned array is the AS under
   # processing.
   #-----------------------------------------------------
   push(@rtrn_array, $community_file);

   #-----------------------------------------------------
   # First, we need to clean up the line with community
   # attributes so we only keep numbers with the format:
   #    20965:155
   #-----------------------------------------------------
   $attr_list[0] =~ s/community\.contains\(//;

   foreach $attribute (@attr_list) {
      #-----------------------------------------------------
      # Eliminate commas "," and end parentesis ")".
      #-----------------------------------------------------
      $attribute =~ s/(,|\))//;
   }
```

```
   #-------------------------------------------------------------
   # Open the file that contains the community attributes
   # information.
   #-------------------------------------------------------------
   $file3 = "data/" . $community_file . ".data";
   print {"MESSAGES"} "     - Processing community attribute in: $file3\n";

   open(COMM_FILE, "$file3") or die "Can't open $file3: $!\n";

   #-------------------------------------------------------------
   # Scan each line of the COMM_FILE file.
   #-------------------------------------------------------------
   while ($comm_line = <COMM_FILE>) {
      #----------------------------------------------------------
      # Store each word in "comm_line" in the line array:
      # "@comm_array".
      #----------------------------------------------------------
      @comm_array = split(" ", $comm_line);

      #----------------------------------------------------------
      # "@comm_array" returns false on empty lines. This check
      # avoids warning messages on blank lines.
      #----------------------------------------------------------
      if (@comm_array) {
         #-------------------------------------------------------
         # We are looking for an "import" line. When we find one
         # we first clean the community.append attribute from
         # the characters "(" and ");", and then we compare this
         # number with the ones provided as parameters. If we
         # find a match we include the AS in the array that we
         # return.
         #-------------------------------------------------------
         if ($comm_array[0] eq "import:") {
            $comm_array[6] =~ s/\(//;
            $comm_array[6] =~ s/\);//;

            foreach $attribute (@attr_list) {
               if ($attribute eq $comm_array[6]) {
                  push(@rtrn_array, $comm_array[2]);
                  last;
               }
            }
         }
      }
   }

   return @rtrn_array;
}
#===============================================================
# Process_Community_Attr Ends
#===============================================================




#===============================================================
#
# Funtion: Sanitize_Import_Groups
#
#===============================================================
# This subroutine sanitizes the Import_Groups data structure
# before processing the data.
#---------------------------------------------------------------
# Inputs:
# 1. None
#---------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned by default.
#---------------------------------------------------------------

sub Sanitize_Import_Group() {

   for ($n=0; $n<$length; $n++) {

      $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };

      #-------------------------------------------------------
      # If the group is empty, continue to the next one.
      #-------------------------------------------------------
      if ($array_lgth == -1) {
         next;
      }

      for ($p=0; $p<=$array_lgth; $p++) {
         #----------------------------------------------------
         # Eliminate strange characters from the destination
         # data, e.g.: <^, +$> or *$>.
         #----------------------------------------------------
         $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] =~ s/<\^//;
         $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] =~ s/\+\$>//;
         $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] =~ s/\*\$>//;

         #----------------------------------------------------
         # Modify the occurrence of "any and not community(..)"
         # to just "ANY".
         #----------------------------------------------------
      if ( ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p]   eq "any")  &&
           ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+1] eq "and")  &&
           ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+2] eq "not")  &&
           ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+3] =~ /community/) ) {
         $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] = "ANY";
         splice( @{$Import_Groups{$ASlist[$i]}{$ASlist[$n]}}, ($p+1), 3);
         $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };
      }

         #----------------------------------------------------
         # Modify the occurrence of
         # "(ANY AND NOT AS5413:RS-NOT)" to just "ANY".
         #----------------------------------------------------
      if ( ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p]   eq "\(ANY")  &&
           ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+1] eq "AND")    &&
           ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+2] eq "NOT")    )
      #    ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+2] eq "AS5413:RS-NOT\)") {
         $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] = "ANY";
         splice( @{$Import_Groups{$ASlist[$i]}{$ASlist[$n]}}, ($p+1), 3);
         $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };
      }

         #----------------------------------------------------
         # The following options are exclusive from each other.
         # Hence we use the elfsif switch command.
         #----------------------------------------------------
         # Modify the occurrence of either:
         #   <X> AND NOT {0.0.0.0/0}
         #   <X> AND NOT AS5413:RS-NOT
         # to just <X>.
         #----------------------------------------------------
      if ( ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p]   eq "AND")  &&
           ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+1] eq "NOT")  )
      #    ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+2] eq "{0.0.0.0/0}")
      #    ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+2] eq "AS5413:RS-NOT") {
         splice( @{$Import_Groups{$ASlist[$i]}{$ASlist[$n]}}, $p, 3);
         $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };
         $p--;
      }
         #----------------------------------------------------
         # Modify the occurrence of "<X> NOT {0.0.0.0/0}"
         # to just <X>.
         #----------------------------------------------------
      elsif ( ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p]   eq "NOT")  &&
              ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p+1] eq "\{0\.0\.0\.0/0}")  )
      {
         splice( @{$Import_Groups{$ASlist[$i]}{$ASlist[$n]}}, $p, 2);
         $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };
         $p--;
      }
         #----------------------------------------------------
         # Delete "AND".
         #----------------------------------------------------
         elsif ($Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p]   eq "AND") {
            splice( @{$Import_Groups{$ASlist[$i]}{$ASlist[$n]}}, $p, 1);
            $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };
            $p--;
         }
      }

      $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };

      #-------------------------------------------------------
      # Eliminate duplicated information in the Import Group.
      #-------------------------------------------------------
      for ($p=0; $p<=$array_lgth; $p++) {
         for ($q=($p+1); $q<=$array_lgth; $q++) {
            if ( $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$p] eq
                 $Import_Groups{$ASlist[$i]}{$ASlist[$n]}[$q] ) {
               splice( @{$Import_Groups{$ASlist[$i]}{$ASlist[$n]}}, $q, 1);
               $array_lgth = $#{ $Import_Groups{$ASlist[$i]}{$ASlist[$n]} };
               $q--;
            }
         }
      }
   }
}
```

```
   return 1;
}
#============================================================
# Sanitize_Import_Group Ends
#============================================================




#============================================================
#
# Funtion: Dump_Adjacency_Matrix
#
#============================================================
# This subroutine dumps the general Adjacency Matrix in the
# DGPS.out file.
#------------------------------------------------------------
# Inputs:
# 1. None
#------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned by default.
#------------------------------------------------------------

sub Dump_Adjacency_Matrix() {

open(MATRIX, ">DGPS.out") or die "Can't create DGPS.out file: $!\n";

foreach $row (@matrixrow) {
   print {"MATRIX"} "@$row\n";
}

close(MATRIX);

return 1;

}
#============================================================
# Dump_Adjacency_Matrix Ends
#============================================================




#============================================================
#
# Funtion: Dump_Exp_Imp_Groups
#
#============================================================
# This subroutine dumps the Export_Groups and Import_Groups
# data structure in the DGPS.dump file.
#------------------------------------------------------------
# Inputs:
# 1. None
#------------------------------------------------------------
# Outputs:
# 1. A 1 (True) is returned by default.
#------------------------------------------------------------

sub Dump_Exp_Imp_Groups() {

open(DUMP, ">DGPS.dump") or die "Can't create DGPS.dump file: $!\n";


for ($m=0; $m<$length; $m++) {
   print {"DUMP"} "\n++++++++++++++++++++++++++++++++++++";
   print {"DUMP"} "\n             $ASlist[$m]";
   print {"DUMP"} "\n++++++++++++++++++++++++++++++++++++\n";

   for ($n=0; $n<$length; $n++) {
      $e_lgth = $#{ $Export_Groups{$ASlist[$m]}{$ASlist[$n]} };
      $i_lgth = $#{ $Import_Groups{$ASlist[$m]}{$ASlist[$n]} };

      print {"DUMP"} "\nExport Group for $ASlist[$n]:";

      for ($p=0; $p<=$e_lgth; $p++) {
         print {"DUMP"} " $Export_Groups{$ASlist[$m]}{$ASlist[$n]}[$p]";
      }

      print {"DUMP"} "\nImport Group for $ASlist[$n]:";

      for ($p=0; $p<=$i_lgth; $p++) {
            print {"DUMP"} " $Import_Groups{$ASlist[$m]}{$ASlist[$n]}[$p]";
      }
   }
   print {"DUMP"} "\n";
}


close(DUMP);

return 1;

}
#============================================================
# Dump_Exp_Imp_Groups Ends
#============================================================
```

B.3 Perl module: DG_CLASS.pm

```
package  DG_CLASS;
require  Exporter;

our @ISA = qw(Exporter);

#============================================================
#
# Class: DG_CLASS
#
#============================================================
# This class stores and manages the data structure that stores
# the Destination Groups and also finds the Fundamental Groups.
# The Fundamental Groups are stored in the DG_CLASS.dump file.
#------------------------------------------------------------
# Data Structure description:
#
# Dest_Grp: Structure that holds the Destination Groups' info.
#           It contains 3 elements:
#    name_counter: This counter is used to create new names for
#           the Fundamental Groups when they are divided into
#           more specific (smaller) sets.
#    num_of_Dest_Groups: This variable contains the total number
#           of ACTIVE Destination Groups.
#    Destination_Groups: This hash holds the information for all
#           the Destination Groups. It contains 4 elements:
#       STATUS = Set to 1 when the Fundamental Group is active,
#                otherwise is set to 0.
#       SET1   = The 1st group from where this Fundamental
#                Group descends.
#       SET2   = The 2nd group from where this Fundamental
#                Group descends.
#       AS_SET = List of ASs that belong to this Fundamental
#                Group. This list may be empty if the Group is
#                not longer "fundamental", but it may keep this
#                list for the original destinations in order to
#                process seeds (see method find_seed).
#------------------------------------------------------------
# Created by: David Arjona
# Date: 12/June/2007
#------------------------------------------------------------

use IP_CLASS;
$AS_num = IP_CLASS->new;




#============================================================
# new: Constructor for class DG_CLASS.
#============================================================
# Input:
#    None.
#------------------------------------------------------------
# Output:
#    The class.
#------------------------------------------------------------
sub new {
   my $obj = shift;

   my $class = ref($obj) || $obj;

   open(MSG, ">DG_CLASS.err") or die "Can't create DG_CLASS.err file: $!\n";

   print {"MSG"} "================================================\n";
   print {"MSG"} "  This is the log file for module: DG_CLASS.pm\n";
   print {"MSG"} "================================================\n\n";

   #------------------------------------------------------------
   # The Dest_Grp data structure is created here.
```

```
   #-------------------------------------------------------------
   my $Dest_Grp = {
       name_counter        => 1,
       num_of_Dest_Groups  => 0,
       #Destination_Groups  => (),
       #-------------------------------------------------------------
       # The following hash structure is created when each
       # Destination Group is added. The following is just an
       # example and reminder of how this structure is formed.
       #-------------------------------------------------------------
       #   Destination_Groups => {
       #       STATUS => 1
       #       SET1   => "AS-A",
       #       SET2   => "AS-B",
       #       AS_SET => (AS123, AS345, AS6789),
       #   },
       #-------------------------------------------------------------
   };


   bless($Dest_Grp, $class);
   return $Dest_Grp;
}




#===============================================================
# increment: This method increases only the variable
#    num_of_Dest_Groups. It was created for TESTING purposes
#    ONLY.
#===============================================================
# Input:
#    None.
#---------------------------------------------------------------
# Output:
#    This method returns 1 by default.
#---------------------------------------------------------------
sub increment {
   my $obj = shift;

   $obj->{num_of_Dest_Groups} ++;

   return 1;
}




#===============================================================
# disp_all_DG: This method PRINTS all the Destination
#    Group stored.
#===============================================================
# Input:
#    None.
#---------------------------------------------------------------
# Output:
#    This method returns 1 by default.
#---------------------------------------------------------------
sub disp_all_DG {
   my $obj = shift;

   foreach $row ( @{$obj->{Destination_Groups}} ) {
      print "$obj->{$row}{STATUS}  $row\n";
   }
   print "\n";

   return 1;
}




#===============================================================
# disp_last_DG: This method ONLY returns the last Destination
#    Group stored.
#===============================================================
# Input:
#    None.
#---------------------------------------------------------------
# Output:
#    This method returns the last Destination Group in the data
#    structure.
#---------------------------------------------------------------
sub disp_last_DG {
   my $obj = shift;

   return $obj->{Destination_Groups}[-1];
}




#===============================================================
# disp_total: This method PRINTS and returns the total number of
#    Destination Groups currently stored.
#===============================================================
# Input:
#    None.
#---------------------------------------------------------------
# Output:
#    This method returns the total number of Destination
#    Groups.
#---------------------------------------------------------------
sub disp_total {
   my $obj = shift;

   print
   "Total number of active Destination Groups: $obj->{num_of_Dest_Groups}\n";

   return $obj->{num_of_Dest_Groups};
}




#===============================================================
# dump_all_DG: This method dumps all valid Destination
#    Groups (Fundamental Groups) on file "DG_CLASS.dump".
#===============================================================
# Input:
#    None.
#---------------------------------------------------------------
# Output:
#    This method returns 1 by default.
#---------------------------------------------------------------
sub dump_all_DG {
   my $obj = shift;

   open(DUMP, ">DG_CLASS.dump") or die
   "Can't create DG_CLASS.dump file: $!\n";

   foreach $row ( @{$obj->{Destination_Groups}} ) {
      if ( $obj->{$row}{STATUS} ) {
         print {"DUMP"} "$row: \n";
         my $counter = 0;

         foreach $ASnum ( @{$obj->{$row}{AS_SET}} ) {
            print {"DUMP"} " $ASnum";
            $counter++;

            if (! ($counter % 10)) {
               print {"DUMP"} "\n";
            }
         }
            print {"DUMP"} "\n";
      }
   }

   close(DUMP);

   return 1;
}




#===============================================================
# find_seed: This method finds if a seed value (an AS number) is
#     being passed in a destination group.
#===============================================================
#===============================================================
# There are two ways of finding if an AS belongs to a
# destination group:
# 1. If the destination group already exists in the data
#    structure we may just need to check if the AS is stored in
#    this group.
# 2. Otherwise, find this information by using the peval tool
```

```
#    and then add this destination group to the database (but
#    without activating it).
#------------------------------------------------------------------
# Input:
#    1. A seed value ($_[0]) which is just an AS number of the
#       form AS1234.
#    2. A destination group of the form AS-PIPEX.
#------------------------------------------------------------------
# Output:
#    This method returns 1 if the seed value (AS) was found in
#    the destination group, or 0 otherwise.
#------------------------------------------------------------------
sub find_seed {
   my $obj  = shift;
   my $seed = shift;
   my $DG   = shift;

   #-----------------------------------------------------------
   # Chek if this is a new destination (has not been added
   # as a Destination Group previously).
   #-----------------------------------------------------------
   if ($obj->find_DG($DG) )   #!!!!!!!! ($#{$obj->{$DG}{AS_SET}} eq -1) {
      foreach $AS ( @{$obj->{$DG}{AS_SET}} ) {
         if ($AS eq $seed) {
            print {"MSG"} "Seed value $seed returned 1 for $DG.\n";
            return 1;
         }
      }
   }
   else
   #-----------------------------------------------------------
   # Let's use the peval tool.
   #------------------------------------------------------------ {
         #-----------------------------------------------------
         # Using the "peval" tool obtain the list of ASs for the
         # current destination.
         #-----------------------------------------------------
         @array1 = split(" ", qx/peval -no-as $DG/);

         #-----------------------------------------------------
         # If we get the most common result for peval:
         #      ((AS12495 AS15431 ... AS25587 AS30746 ))
         #
         # Eliminate the double parenthesis in the 1st element
         # of array1 and also eliminate the last.
         #-----------------------------------------------------
         if ( $array1[0] =~ /\({2}/ ) {
            $array1[0] =~ s/\({2}//;
            pop(@array1);
            print {"MSG"} "Length of $DG: " . @array1 . "\n";
         }
         #-----------------------------------------------------
         # If we get the other result for peval:
         #      ({217.67.96.0/20, ..., 62.124.0.0/17})
         #
         # Eliminate the parenthesis and key in the 1st and last
         # elements of array1.
         #-----------------------------------------------------
         elsif ( $array1[0] =~ /\(\{/ ) {
            $array1[0] =~ s/\(\{//;
            $array1[-1] =~ s/}\)//;
            @array1 = $AS_num->IP_to_ASN_convert(@array1);
            print {"MSG"} "Length of $DG: " . @array1 . "\n";
         }
         #-----------------------------------------------------
         # When "peval" cannot find the destination it returns
         # "NOT ANY". The seed was not found.
         #-----------------------------------------------------
         elsif ($array1[0] eq "NOT") {
            return 0;
         }
         #-----------------------------------------------------
         # "peval" will return ANY if we query for ANY. And this
         # includes the seed.
         #-----------------------------------------------------
         elsif ($array1[0] eq "ANY") {
            return 1;
         }

         #-----------------------------------------------------
         # Add the Destination Group but do not process it to
         # find fundamental groups. Also notice that the entry
         # is not active (STATUS=0).
         #-----------------------------------------------------
         %{$obj->{$DG}} = (
                    STATUS => 0,
                    SET_1  => "",
                    SET_2  => "",
                    AS_SET => [ @array1 ],
         );

         push( @{$obj->{Destination_Groups}}, $DG);

         #-----------------------------------------------------
         # Find if the seed is included in the list of ASs for
         # this destination group.
         #-----------------------------------------------------
         foreach $AS (@array1) {
            if ($AS eq $seed) {
               print {"MSG"} "Seed value $seed returned 1 for $DG.\n";
               return 1;
            }
         }
   }

   #-----------------------------------------------------------
   # If we get to this line, the seed has not been found.
   #-----------------------------------------------------------
   return 0;
}




#============================================================
# find_DG: This method finds if the Destination Groups has
#     previously been added to the data structure.
#============================================================
# Input:
#    A Destination Group ($_[0]).
#------------------------------------------------------------
# Output:
#    This method returns 1 if the Destination Group already
#    exists and 0 if it does not exist in the Data Base.
#------------------------------------------------------------
sub find_DG {
   my $obj = shift;

   #-----------------------------------------------------------
   # Check if the Destination Group already exists.
   #-----------------------------------------------------------
   for $i (0 .. $#{$obj->{Destination_Groups}} ) {
      if ($_[0] eq $obj->{Destination_Groups}[$i]) {
         return 1;
      }
   }

   return 0;
}




#============================================================
# add_DG: This method adds Destination Groups to the data
#     structure.
#============================================================
# Input:
#    An array of Destinations (@_).
#------------------------------------------------------------
# Output:
#    This method returns 1 by default, but it may return 0 if
#    there was a problem with the execution.
#------------------------------------------------------------
sub add_DG {
   my $obj = shift;

   for $DG ( @_ ) {
      #-----------------------------------------------------
      # Chek if this is a new destination (has not been added
      # as a Destination Group previously).
      #-----------------------------------------------------
      if (! $obj->find_DG($DG) ) {
         #-----------------------------------------------------
         # Using the "peval" tool obtain the list of ASs for the
         # current destination.
         #-----------------------------------------------------
         @array1 = split(" ", qx/peval -no-as $DG/);

         #-----------------------------------------------------
         # If we get the most common result for peval:
         #      ((AS12495 AS15431 ... AS25587 AS30746 ))
```

```
        #
        # Eliminate the double parenthesis in the 1st element
        # of array1 and also eliminate the last.
        #-------------------------------------------------------
        if ( $array1[0] =~ /\({2}/ ) {
           $array1[0] =~ s/\({2}//;
           pop(@array1);
           print {"MSG"} "Length of $DG: " . @array1 . "\n";
        }
        #-------------------------------------------------------
        # If we get the other result for peval:
        #      ({217.67.96.0/20, ..., 62.124.0.0/17})
        #
        # Eliminate the parenthesis and key in the 1st and last
        # elements of array1.
        #-------------------------------------------------------
        elsif ( $array1[0] =~ /\(\{/ ) {
           $array1[0] =~ s/\(\{//;
           $array1[-1] =~ s/}\)//;
           @array1 = $AS_num->IP_to_ASN_convert(@array1);
           print {"MSG"} "Length of $DG: " . @array1 . "\n";
        }
        #-------------------------------------------------------
        # When "peval" cannot find the destination it returns
        # "NOT ANY". In this case we ignore the destination and
        # skip to the next one.
        #-------------------------------------------------------
        elsif ($array1[0] eq "NOT") {
           next;
        }
        #-------------------------------------------------------
        # "peval" will return ANY if we query for ANY.
        #-------------------------------------------------------
        elsif ($array1[0] eq "ANY") {
           next;
        }


        #-------------------------------------------------------
        # Time to process and add the Destination Group.
        #-------------------------------------------------------
        if (@array1) {
           %{$obj->{$DG}} = (
                      STATUS => 1,
                      SET_1  => "",
                      SET_2  => "",
                      AS_SET => [ @array1 ],
           );

           if ( $obj->create_Fundamental_Groups($DG) ) {
              #----------------------------------------------
              # Add $DG to Destination_Groups and increase
              # the number of Destination Groups if $DG is
              # still active (STATUS=1).
              #----------------------------------------------
              push( @{$obj->{Destination_Groups}}, $DG);

              if ( $obj->{$DG}{STATUS} ) {
                 $obj->{num_of_Dest_Groups} ++;
              }

              #########
              # DEBUG #
              #########
              #printf {"MSG"} "%20s %4s %15s %15s \n\n",
              #       $DG,
              #       $obj->{$DG}{STATUS},
              #       $obj->{$DG}{SET_1},
              #       $obj->{$DG}{SET_2};
           }
           else {
              print {"MSG"} " ERROR: $DG was not added to the\n";
              print {"MSG"} "        Destination Groups list.\n"
           }
        }
        else {
           print {"MSG"} " ERROR: This is an Invalid Destination Group!!!\n";
           print {"MSG"} "        *** $DG ***\n";
        }
     }
  }

  return 1;
}
```

```
#============================================================
# create_Fundamental_Groups: This method process the
#     intersection of new Destination Group objects and creates
#     the Fundamental Groups.
#============================================================
#============================================================
# The new Destination Group is always Set A and will be compared
# against the existing Fundamental Groups (FG) one by one.
# The processing in this method is divided in 3 main parts:
# 1. Check if Set A is equal to an existing FG, if so we do not
#    need to continue processing Set A.
# 2. Find the intersection between Set A and the other FGs, also
#    keep track of the B-A groups which also create new FGs.
# 3. Finally find the FG contained in A-B and decide if Set A
#    should continue to be active or should just be kept in
#    memory as a record.
#------------------------------------------------------------
# Input:
#    A Destination Group object.
#------------------------------------------------------------
# Output:
#    This method returns 1 by default, but it may return 0 if
#    there was a problem with the execution.
#------------------------------------------------------------
sub create_Fundamental_Groups {
   my $obj    = shift;
   my $Set_A  = shift;



   #============================================================
   #
   # 1st For Loop:  Is A equal to B ???
   #
   #============================================================
   foreach $Set_B ( @{$obj->{Destination_Groups}} )    {
      #-------------------------------------------------------
      # We do not process elements in the Destination_Groups
      # that are:
      #   - Not valid (STATUS = 0)
      #-------------------------------------------------------
      if (! $obj->{$Set_B}{STATUS} ) {
         next;
      }

      #-------------------------------------------------------
      # A_equals_B : This variable is true if A equals B.
      # intersec   : The intersecting set between A and B.
      #-------------------------------------------------------
      my $A_equals_B = 1;
      my @intersec   = ();

      #-------------------------------------------------------
      # Check if Set_A (the new Destination Group) has the same
      # number of elements than Set_B (the Destination Group
      # currently being processed).
      #-------------------------------------------------------
      if ( $#{$obj->{$Set_A}{AS_SET}} ==
           $#{$obj->{$Set_B}{AS_SET}} ) {
         #----------------------------------------------------
         # Process each element of Set_A looking for matches
         # with Set_B.
         #----------------------------------------------------
         AS_EQ_LOOP: foreach $AS_1 ( @{$obj->{$Set_A}{AS_SET}} ) {
            foreach $AS_2 ( @{$obj->{$Set_B}{AS_SET}} ) {
               if ( $AS_1 eq $AS_2 ) {
                  #-------------------------------------------
                  # We have found a match between ASs. We add
                  # the matched AS to the intersection list.
                  # Then continue processing the next AS in
                  # Set_A.
                  #-------------------------------------------
                  push(@intersec, $AS_1);
                  next AS_EQ_LOOP;
               }
            }
            #----------------------------------------------------
            # We have found an AS which is in A-B. This means
            # that Set_A cannot be equal to Set_B.
            #----------------------------------------------------
            $A_equals_B = 0;
         }
         #########
         # DEBUG #
         #########
```

```
      #print {"MSG"} "              Intersection size: " .
      #         @intersec . "\n";

      #-------------------------------------------------------
      # If after processing each element of Set_A, variable
      # A_equals_B is still true, it means that Set_A and
      # Set_B are identical sets.
      #-------------------------------------------------------
      if ($A_equals_B) {
         #----------------------------------------------------
         # Disable Set_A and exit this method.
         #----------------------------------------------------
         $obj->{$Set_A}{SET_1}  = $Set_B;
         $obj->{$Set_A}{SET_2}  = "identical";
         $obj->{$Set_A}{STATUS} = 0;
         #-#$obj->{$Set_A}{AS_SET} = ();

         return 1;
      }
   }
}
#-----------------------------------------------------------
# End lst For Loop.
#-----------------------------------------------------------



#===========================================================
#
# 2nd For Loop: Find the Intersection between A and B and
#               also B-A.
#
#===========================================================
my @Temp_Groups = ();
my $Deact_Set_A = 0;

foreach $Set_B ( @{$obj->{Destination_Groups}} )    {
   #--------------------------------------------------------
   # We do not process elements in the Destination_Groups
   # that are:
   #   - Not valid (STATUS = 0)
   #   - Descendants of Set_A
   #--------------------------------------------------------
   if ( (! $obj->{$Set_B}{STATUS} )        ||
        ($obj->{$Set_B}{SET_1} eq $Set_A)  ||
        ($obj->{$Set_B}{SET_2} eq $Set_A)  ) {
      next;
   }

   #--------------------------------------------------------
   # intersec:  This is the intersecting set between A and B.
   #--------------------------------------------------------
   my @intersec = ();
   my @b_minus_a = ();

   #--------------------------------------------------------
   # When A and B are not equal there are only 3 options
   # available:
   #   1. A and B does not intersect.
   #   2. A and B are intersecting groups (A intersec B, A-B
   #      and B-A are all non-zero).
   #   3. A is included in B or B is included in A, in which
   #      case either A or B is equal to the intersection.
   #--------------------------------------------------------

   #--------------------------------------------------------
   # Process each element of Set_B looking for matches
   # with Set_A.
   #--------------------------------------------------------
   AS_B_LOOP: foreach $AS_B (@{$obj->{$Set_B}{AS_SET}}) {
      for ($AS_A=$#{$obj->{$Set_A}{AS_SET}}; $AS_A>=0; $AS_A--) {
         if ( $obj->{$Set_A}{AS_SET}[$AS_A] eq $AS_B ) {
            #-----------------------------------------------
            # We have found a match between ASs. We add the
            # matched AS to the intersection list and
            # continue processing the next AS in Set_B.
            #-----------------------------------------------
            push (@intersec, $AS_B);
            next AS_B_LOOP;
         }
      }
      #-----------------------------------------------------
      # We have found an AS which is in B-A.
      #-----------------------------------------------------
      push (@b_minus_a, $AS_B);
   }
   #########
```

```
   # DEBUG #
   #########
   #print {"MSG"} "   $Set_A vs $Set_B intersection size: " .
   #             @intersec . "\n";


   #--------------------------------------------------------
   # Set_A and Set_B do not intersect. Do nothing!
   #--------------------------------------------------------
   if (! @intersec) {
      next;
   }

      #-----------------------------------------------------
      # Set_A and Set_B are intersecting.
      #-----------------------------------------------------
      elsif ( ( $#{$obj->{$Set_A}{AS_SET}} > $#intersec ) &&
              ( $#{$obj->{$Set_B}{AS_SET}} > $#intersec ) ) {
         #--------------------------------------------------
         # Create 2 Fundamental Groups: A intersec B and
         # B-A. And deactivate Set_B (STATUS=0). A-B and
         # deactivating Set_A will be done later.
         #--------------------------------------------------
         $obj->{$Set_B}{STATUS} = 0;

         if ($Set_B =~ /FG/) {
            $obj->{$Set_B}{AS_SET} = ();
         }

         $name = "FG_" . $obj->{name_counter} ++;
         %{$obj->{$name}} = (
                 STATUS => 1,
                 SET_1  => $Set_A,
                 SET_2  => $Set_B,
                 AS_SET => [ @intersec ],
         );
         push( @{$obj->{Destination_Groups}}, $name);

         #########
         # DEBUG #
         #########
         #printf {"MSG"} "%20s %4s %15s %15s  intersec  %d\n",
         #       $name,
         #       $obj->{$name}{STATUS},
         #       $obj->{$name}{SET_1},
         #       $obj->{$name}{SET_2},
         #       ($#{$obj->{$name}{AS_SET}} + 1);


         $name = "FG_" . $obj->{name_counter} ++;
         %{$obj->{$name}} = (
                 STATUS => 1,
                 SET_1 => $Set_A,
                 SET_2 => $Set_B,
                 AS_SET => [ @b_minus_a ],
         );
         push( @{$obj->{Destination_Groups}}, $name);

         #########
         # DEBUG #
         #########
         #printf {"MSG"} "%20s %4s %15s %15s  B-A       %d\n",
         #       $name,
         #       $obj->{$name}{STATUS},
         #       $obj->{$name}{SET_1},
         #       $obj->{$name}{SET_2},
         #       ($#{$obj->{$name}{AS_SET}} + 1);

         #--------------------------------------------------
         # Since we are deactivating Set_B and activating
         # 2 new groups we increase the counter by 1.
         #--------------------------------------------------
         $obj->{num_of_Dest_Groups} ++;
         #--------------------------------------------------
         # All intersections are temporarly stored in the
         # "intersec" array.
         #--------------------------------------------------
         push(@Temp_Groups, @intersec);
         $Deact_Set_A = 1;
      }

      #-----------------------------------------------------
      # Set_A is included in Set_B (A is a subset of B).
      #-----------------------------------------------------
      elsif ( $#{$obj->{$Set_A}{AS_SET}} == $#intersec ) {
         $obj->{$Set_B}{STATUS} = 0;
```

```
            if ($Set_B =~ /FG/) {
               $obj->{$Set_B}{AS_SET} = ();
            }

            $obj->{$Set_A}{SET_1}  = $Set_B;
            $obj->{$Set_A}{SET_2}  = "subset of B";


            $name = "FG_" . $obj->{name_counter} ++;
            %{$obj->{$name}} = (
                    STATUS => 1,
                    SET_1  => $Set_A,
                    SET_2  => $Set_B,
                    AS_SET => [ @b_minus_a ],
            );
            push( @{$obj->{Destination_Groups}}, $name);

            #########
            # DEBUG #
            #########
            #printf {"MSG"} "%20s %4s %15s %15s  B-A       %d\n",
            #       $name,
            #       $obj->{$name}{STATUS},
            #       $obj->{$name}{SET_1},
            #       $obj->{$name}{SET_2},
            #       ($#{$obj->{$name}{AS_SET}} + 1);

            #-----------------------------------------------------
            # All intersections are temporarly stored in the
            # "intersec" array.
            #-----------------------------------------------------
            push(@Temp_Groups, @intersec);
         }

         #-----------------------------------------------------
         # Set_B is included in Set_A (B is a subset of A).
         #-----------------------------------------------------
         elsif ( $#{$obj->{$Set_B}{AS_SET}} == $#intersec ) {
            $obj->{$Set_B}{SET_1} = $Set_A;
            $obj->{$Set_B}{SET_2} = "subset of A";

            #########
            # DEBUG #
            #########
            #printf {"MSG"} "%20s %4s %15s %15s            %d\n",
            #       $Set_B,
            #       $obj->{$Set_B}{STATUS},
            #       $obj->{$Set_B}{SET_1},
            #       $obj->{$Set_B}{SET_2},
            #       ($#{$obj->{$Set_B}{AS_SET}} + 1);

            #-----------------------------------------------------
            # All intersections are temporarly stored in the
            # "intersec" array.
            #-----------------------------------------------------
            push(@Temp_Groups, @intersec);
            $Deact_Set_A = 1;
         }

         #-----------------------------------------------------
         # If we get to this line we must have an error!
         #-----------------------------------------------------
         else {
            $obj->{$Set_A}{STATUS} = 0;

            print {"MSG"} " ERROR: Unforseen Problem while processing\n";
            print {"MSG"} "        the intersection of the Fundamental\n";
            print {"MSG"} "        Groups.\n\n";
            return 0;
         }
      }
      #-----------------------------------------------------------
      # End 2nd For Loop
      #-----------------------------------------------------------



      #============================================================
      #
      # Process A-B: Which is equal to:
      #    A - ( (A intersec B1) U (A intersec B2) U ...)
      #
      #============================================================
      my @a_minus_b = ();

      if (@Temp_Groups) {
         INTSC_LOOP: foreach $A_elm ( @{$obj->{$Set_A}{AS_SET}} ) {
            foreach $intersec_element ( @Temp_Groups )  {
               if ( $A_elm eq $intersec_element ) {
                  next INTSC_LOOP;
               }
            }
            #-----------------------------------------------------
            # We have found an AS which is in A-B.
            #-----------------------------------------------------
            push(@a_minus_b, $A_elm);
         }

         #-----------------------------------------------------
         # Let's create the new A-B Fundamental Group.
         #-----------------------------------------------------
         if (@a_minus_b) {
            $name = "FG_" . $obj->{name_counter} ++;
            %{$obj->{$name}} = (
                    STATUS => 1,
                    SET_1  => $Set_A,
                    SET_2  => "A-B",
                    AS_SET => [ @a_minus_b ],
            );
            push( @{$obj->{Destination_Groups}}, $name);
            $obj->{num_of_Dest_Groups} ++;

            #########
            # DEBUG #
            #########
            #printf {"MSG"} "%20s %4s %15s %15s            %d\n",
            #       $name,
            #       $obj->{$name}{STATUS},
            #       $obj->{$name}{SET_1},
            #       $obj->{$name}{SET_2},
            #       ($#{$obj->{$name}{AS_SET}} + 1);

            #-----------------------------------------------------
            # Finally, deactivate Set_A (STATUS=0).
            #-----------------------------------------------------
            $obj->{$Set_A}{STATUS} = 0;
            #-#$obj->{$Set_A}{AS_SET} = ();
            }
         }

         #-----------------------------------------------------
         # If A-B = 0 but Deact_Set_A is true, it means Set_A has
         # been already covered by other Desitnation Groups.
         # Deactivate Set_A.
         #-----------------------------------------------------
         if ( $Deact_Set_A && (! @a_minus_b) ) {
            $obj->{$Set_A}{STATUS} = 0;
            #-#$obj->{$Set_A}{AS_SET} = ();
         }

         return 1;
      }

      1;
```

B.4 Perl module: IP_CLASS.pm

```
package  IP_CLASS;
require  Exporter;

our @ISA    = qw(Exporter);

#============================================================
#
# Class: IP_CLASS
#
#============================================================
# This class stores and manages the data structure that stores
# the IP Prefixes that correspond to an AS number. It also
# processes an array of IP prefixes and returns an array of
# corresponding AS.
#------------------------------------------------------------
# Data Structure description:
#
# IP_structure: The main data structure, it only contains 1
#        element.
#    IP_prefix: This hash structure contains the AS to which an
#           IP belongs.
#------------------------------------------------------------
# Created by: David Arjona
# Date: 3/July/2007
#------------------------------------------------------------
```

```
#============================================================
# new: Constructor for class DG.
#============================================================
# Input:
#    None.
#------------------------------------------------------------
# Output:
#    The class.
#------------------------------------------------------------
sub new {
   my $obj = shift;

   my $class = ref($obj) || $obj;

   open(ASN, ">IP_CLASS.err") or die "Can't create IP_CLASS.err file: $!\n";

   print {"ASN"} "================================================\n";
   print {"ASN"} "  This is the log file for module: IP_CLASS.pm\n";
   print {"ASN"} "================================================\n";

   #------------------------------------------------------------
   # The Dest_Grp data structure is created here.
   #------------------------------------------------------------
   my $IP_structure = {
       IP_prefix => (),
   };

   bless($IP_structure, $class);
   return $IP_structure;
}
```

```
#============================================================
# IP_to_ASN_convert: This method processes an array of IP
#     prefixes and builds a corresponding array of AS.
#============================================================
# Input:
#    An array of IP prefixes (@_).
#------------------------------------------------------------
# Output:
#    An array of ASs.
#------------------------------------------------------------
sub IP_to_ASN_convert {
   my $obj = shift;

   #------------------------------------------------------------
   # This array will be returned at the end of this method.
   #------------------------------------------------------------
   my @AS_array = ();

   #########
   # DEBUG #
   #########
   my $counter = 0;

   print  {"ASN"} "\n\nProcessing new array:\n";


   for $prefix ( @_ ) {
      $add_to_array = 1;

      #------------------------------------------------------------
      # Eliminate the comma at the end of each prefix.
      #------------------------------------------------------------
      $prefix =~ s/,//;

      #------------------------------------------------------------
      # Chek if this is a new destination (has not been added
      # to the IP_prefix hash previously).
      #------------------------------------------------------------
      if (exists $obj->{IP_prefix}{$prefix}) {
         $temp_AS = $obj->{IP_prefix}{$prefix};
      }
      #------------------------------------------------------------
      # When the destination is new we need to obtain the ASN
      # related to the destination.
      #------------------------------------------------------------
      else {
         $temp_AS = $obj->get_ASN($prefix);
         $obj->{IP_prefix}{$prefix} = $temp_AS;
      }

      #------------------------------------------------------------
      # Before adding the AS number to the array (AS_array),
      # check that this AS is not duplicated.
      #------------------------------------------------------------
      foreach $element (@AS_array) {
         if ($element eq $temp_AS) {
            $add_to_array = 0;
            last;
         }
      }

      #------------------------------------------------------------
      # When the AS number is not duplicated we can add it to
      # AS_array.
      #------------------------------------------------------------
      if ($add_to_array) {
         push(@AS_array, $temp_AS);

         #########
         # DEBUG #
         #########
         print {"ASN"} "$temp_AS ";
         $counter++;
         if (! ($counter % 10)) {
            print {"ASN"} "\n";
         }
      }
   }

   return @AS_array;
}
```

```
#============================================================
# get_ASN: This finds the AS that announces an IP prefix. To
#     obtains this information it uses the whois query into the
#     riswhois service of the NCC RIPE.
#============================================================
# Input:
#    An IP prefix ($_[0]).
#------------------------------------------------------------
# Output:
#    An AS number, if the query is not successful a default
#    value of zero (0) is returned (and a Warning message is
#    generated in the IP_CLASS.err file.
#------------------------------------------------------------
sub get_ASN {
   my $obj    = shift;

   #------------------------------------------------------------
   # This is the return value and it is initialized to zero.
   #------------------------------------------------------------
   my $rtn_value = 0;

   #------------------------------------------------------------
   # The info from whois is stored in Temp_Info.
   #------------------------------------------------------------
   @Temp_Info = split(" ", qx/whois -h riswhois.ripe.net $_[0]/);


   #------------------------------------------------------------
   # Search for the AS in Temp_Info.
   #------------------------------------------------------------
   for $i (0 .. $#Temp_Info) {
      #------------------------------------------------------------
      # The info is stored in the following way:
      #       route:        213.137.192.0/19
      #       origin:       AS13201
      #------------------------------------------------------------
      if ( ($Temp_Info[$i]   eq $_[0])     &&
           ($Temp_Info[$i+1] eq "origin:") ) {
         $rtn_value = $Temp_Info[$i+2];
         last;
      }
      #------------------------------------------------------------
      # When we cannot find the specific IP value we use the
      # closes prefix available in Temp_Info:
      #       route:        212.0.0.0/7
      #       origin:       AS3303
      #------------------------------------------------------------
      elsif ($Temp_Info[$i] eq "origin:") {
         $rtn_value = $Temp_Info[$i+1];
```

```
      }
   }

   if (! $rtn_value) {
      print {"ASN"} " Warning: Could not find $_[0]\n";
   }
   return $rtn_value;
}

1;
```

B.5 Perl script: Preprocess_AS3303.pl

```
#!/usr/bin/perl -w
#================================================================
#
# Preprocess_AS3303.pl
#
#================================================================
# This script adds export rules to the data file for AS3303
# (Swisscom) because the operators of this AS have decided to
# condense all peering export rules using the following line:
#
#    export:    to AS-SWCMPEERS announce AS-SWCMGLOBAL
#
# Notes: This script modifies file AS3303.data
#----------------------------------------------------------------
# Inputs:
# 1. Data file for AS3303:
#       ./data/AS3303.data
# 2. Result from the peval tool for AS-SWCMPEERS.
#
# Outputs:
# 1. The export rules are added to file AS3303.data.
# 2. Other messages, including warnings and errors are displayed
#    in the screen.
#----------------------------------------------------------------
# Created by: David Arjona
# Date: 21/August/2007
#----------------------------------------------------------------



#================================================================
####################################################################
#================================================================
# Main Program starts here
#================================================================
####################################################################
#================================================================


#----------------------------------------------------------------
#----------------------------------------------------------------
# Initilize Section:
#----------------------------------------------------------------
#----------------------------------------------------------------

#----------------------------------------------------------------
# AS_list is the file that contains the list of ASs to be
# processed. We will store this list in the array "ASlist"
# that has "length" number of elements or ASs.
#----------------------------------------------------------------
open(AS_LIST_FILE, "AS_list") or die "Can't open AS list file: $!\n";

while ($ASnum = <AS_LIST_FILE>) {
   chomp($ASnum);
   $length = push(@ASlist, $ASnum);
}

close(AS_LIST_FILE);
print "Length of the AS list: $length\n";

#----------------------------------------------------------------
# $file1: this is just a rename for the AS3303.data file.
# @array1: this is the result from the peval tool for
#    AS-SWCMPEERS.
#----------------------------------------------------------------
$file1 = "data/AS3303.data";
open(EXPORT_FILE, ">>$file1") or die "Can't open $file1: $!\n";
print "\nCurrently processing $file1 :\n";

@array1 = split(" ", qx/peval -no-as AS-SWCMPEERS/);

#----------------------------------------------------------------
#----------------------------------------------------------------
# Initilize Section Ends
#----------------------------------------------------------------
#----------------------------------------------------------------



#----------------------------------------------------------------
#----------------------------------------------------------------
# Main Loop:
#----------------------------------------------------------------
#----------------------------------------------------------------
# This loop processes each AS stored in "@array1"
#----------------------------------------------------------------
AS_LOOP: for $AS_num (@array1) {
   #-------------------------------------------------------------
   # The inner loop matches each "$AS_num" to an AS which
   # belongs to the "ASlist".
   #-------------------------------------------------------------
   for ($i=0; $i<$length; $i++) {
      #----------------------------------------------
      # If we find a match we add the export rule to the data
      # file and continue analyzing the next AS.
      #----------------------------------------------
      if ($ASlist[$i] eq $AS_num) {
         print {"EXPORT_FILE"} "export: to $AS_num announce AS-SWCMGLOBAL\n";
         next AS_LOOP;
      }
   }
}
#----------------------------------------------------------------
#----------------------------------------------------------------
# Main Destination Groups Loop Ends
#----------------------------------------------------------------
#----------------------------------------------------------------


close(EXPORT_FILE);
print "\n *** Preprocess_AS3303.pl Finished! ***\n";

#================================================================
####################################################################
#================================================================
# Main Program finishes here
#================================================================
####################################################################
```

B.6 Perl script: Preprocess_AS3356.pl

```
#================================================================
#!/usr/bin/perl -w
#================================================================
#
# Preprocess_AS3356.pl
#
#================================================================
# This script eliminates a specific set of characters from the
# AS3356.data file. This set of characters has the following
# pattern:
#
#    ^0-[d][d]
#
# Where [d] is any decimal character (0 to 9).
# This script creates a copy of the AS3356.data file in the
# current directory.
#----------------------------------------------------------------
# Inputs:
# 1. Data file for AS3356:
#       ./data/AS3356.data
#
# Outputs:
# 1. Characters are eliminated from the input file.
# 2. A copy of the input file is created in the current working
#    directory.
#----------------------------------------------------------------
# Created by: David Arjona
# Date: 17/December/2007
#----------------------------------------------------------------



#================================================================
####################################################################
#================================================================
# Main Program starts here
#================================================================
####################################################################
```

```
#============================================================


#------------------------------------------------------------
#------------------------------------------------------------
# Initilize Section:
#------------------------------------------------------------
#------------------------------------------------------------

#------------------------------------------------------------
# First we need to make a copy of the original data file. We
# will read from this copy and modify the original file. This
# helps to run the script transparently to the user and keeps a
# safe copy of the original data file.
#------------------------------------------------------------
`cp data/AS3356.data .`;

#------------------------------------------------------------
# $file1: this is just a rename for the AS3356.data read file.
# $file2: this is a rename for the data/AS3356.data write file.
#------------------------------------------------------------
$file1 = "AS3356.data";
open(READ_FILE, "$file1") or die "Can't open $file1: $!\n";
print "\nCurrently reading from $file1 :\n";

$file2 = "data/AS3356.data";
open(WRITE_FILE, ">$file2") or die "Can't open $file2: $!\n";
print "\nCurrently writing to $file2 :\n";

#------------------------------------------------------------
#------------------------------------------------------------
# Initilize Section Ends
#------------------------------------------------------------
#------------------------------------------------------------



#------------------------------------------------------------
#------------------------------------------------------------
# Main Loop:
#------------------------------------------------------------
#------------------------------------------------------------
# This loop processes each AS stored in "@array1"
#------------------------------------------------------------
while ($line = <READ_FILE>) {
   #---------------------------------------------------------
   # Store each word in "line" in the line array: "@larray".
   #---------------------------------------------------------
   @larray = split(" ", $line);

   #---------------------------------------------------------
   # "@larray" returns false on empty lines. This check
   # avoids warning messages on blank lines.
   #---------------------------------------------------------
   if (@larray) {
      #------------------------------------------------------
      # Try to find and delete the following pattern on each
      # element of "larray": ^0-[d][d]
      # Where [d] could be any numerical character.
      #------------------------------------------------------
      for ($i=0; $i<=$#larray; $i++) {
         $larray[$i] =~ s/\^0-\d\d//;

         #---------------------------------------------------
         # We do not write a white space after the final
         # word of a line.
         #---------------------------------------------------
         if ($i == $#larray) {
            print {"WRITE_FILE"} "$larray[$i]";
         }
         else {
            print {"WRITE_FILE"} "$larray[$i] ";
         }
      }
      print {"WRITE_FILE"} "\n";
   }
}
#------------------------------------------------------------
#------------------------------------------------------------
# Main Loop Ends
#------------------------------------------------------------
#------------------------------------------------------------



close(READ_FILE);
close(WRITE_FILE);

print "\n *** Preprocess_AS3356.pl Finished! ***\n";

#============================================================
############################################################
#============================================================
# Main Program finishes here
#============================================================
############################################################
#============================================================
```

B.7 Perl script: AMG2.pl

```
#!/usr/bin/perl -w
#------------------------------------------------------------
# AMG2.pl (Adjacency Matrix Generator 2 Perl script)
#------------------------------------------------------------
# This script creates adjacency matrices for a list of
# destinations.
#------------------------------------------------------------
# Inputs:
# 1. List of ASs to be processed. This list should be
#    stored in text file "./AS_list". This file is also
#    used by other scripts implemented before.
# 2. BGP's AS paths stored in text files in the "data"
#    directory, e.g.:
#       ./data/rrc00.20071129.1559.txt
#       ./data/rtvw2.20071129.1207.txt
# Outputs:
# 1. The Adjacency Matrices are stored in several .out and .mat
#    files in the "data" directory.
#    Note: The .out files can be loaded in Matlab and the .mat
#    files can be loaded in Pajek.
# 2. A list of all the "invalid paths" is stored in the file
#    "inv_path.txt". We define an invalid path as a path that
#    starts and finishes with ASs in the AS_list but that have
#    intermediate ASs which do not belong to AS_list, this
#    creates a path that passes through invalid ASs.
# 3. Other messages, including warnings and errors are stored
#    in file "AMG2.err".
#------------------------------------------------------------
# Created by: David Arjona
# Date: 18/October/2007
#------------------------------------------------------------

#------------------------------------------------------------
# Classes are initialized here:
#    - Rtg_Proc: This class stores and find the routing data
#         from the valid paths.
#------------------------------------------------------------
use RD_CLASS;
$Rtg_Proc = RD_CLASS->new;



#============================================================
############################################################
#============================================================
# Main Program starts here
#============================================================
############################################################
#============================================================


#------------------------------------------------------------
#------------------------------------------------------------
# Initilize Section:
#------------------------------------------------------------
#------------------------------------------------------------

#------------------------------------------------------------
# File "AMG2.err" is the recipient of , status, warnings and
# error messages.
#------------------------------------------------------------
open(MESSAGES, ">AMG2.err") or die "Can't create AMG2.err file: $!\n";

#------------------------------------------------------------
# AS_list is the file that contains the list of ASs to be
# processed. We will store this list in the array "@ASlist"
# that has "$length" number of elements or ASs.
# NOTE: Because AS numbers are stored "just" as numbers in the
# data files, we need to delete the "AS" prefix from each AS
# stored in the file AS_list.
#------------------------------------------------------------
open(AS_LIST_FILE, "AS_list") or die "Can't open AS_list file: $!\n";

while ($ASnum = <AS_LIST_FILE>) {
```

```
   chomp($ASnum);
   $ASnum =~ s/AS//;
   $length = push(@ASlist, $ASnum);
}

close(AS_LIST_FILE);

print {"MESSAGES"} "Length of the AS list: $length\n";
print "\nThis is an Adjacency Matrix of dimensions: $length by $length:\n\n";

#-------------------------------------------------------------
# Initialize all the adjacency matrices ($ADJ_Matrices) to zeros
# and the hash that converts ($AS_to_idx) AS names into numeric
# indices to array @ASlist.
#-------------------------------------------------------------
$z = 0;

foreach $ASname (@ASlist) {
   for $x (0 .. ($length-1)) {
      for $y (0 .. ($length-1)) {
         $ADJ_Matrices{$ASname}[$x][$y] = 0;
      }
   }

   $AS_to_idx{$ASname} = $z++;
}

#-------------------------------------------------------------
# Find the data files we need to analyze. These files are stored
# in the "data" directory.
#-------------------------------------------------------------
@Thelist = glob("data/*.txt");

foreach $file (@Thelist) {
   push(@Filelist, $file);
}

#-------------------------------------------------------------
#-------------------------------------------------------------
# Initilize Section Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



#-------------------------------------------------------------
#-------------------------------------------------------------
# Main Loop:
#-------------------------------------------------------------
#-------------------------------------------------------------
# This loop processes each of the data files looking for ASs in
# "ASlist". It first needs to open the data file and then
# search for a connection between the current AS and the
# other ASs in the "ASlist".
#-------------------------------------------------------------
foreach $file1 (@Filelist) {
   #----------------------------------------------------------
   # The file is opened for reading.
   #----------------------------------------------------------
   print {"MESSAGES"} "\nProcessing $file1 :\n";
   print "Processing $file1 :\n";

   open(DATA_FILE, "$file1") or die "Can't open $file1: $!\n";

   #----------------------------------------------------------
   # Second Loop:
   #----------------------------------------------------------
   # Finds connections between the current AS and the other
   # ASs in the "ASlist".
   #----------------------------------------------------------
   while ($line = <DATA_FILE>) {
      #-------------------------------------------------------
      # Store each word in "line" in the array "larray".
      #-------------------------------------------------------
      @larray = split(" ", $line);

      #-------------------------------------------------------
      # "@larray" returns false on empty lines. This check
      # avoids warning messages on blank lines.
      #-------------------------------------------------------
      if (@larray) {
         $dest_found = 0;
         @AS_array = ();

         #----------------------------------------------------
         # The following loop looks for a destination which
         # matches any of the ASs in the ASlist.
         #----------------------------------------------------
         for $i (0 .. $#larray) {
            if ( $Rtg_Proc->belongs_to_AS_list($larray[$i]) ) {
               #-------------------------------------------
               # When we find a destination we add the rest
               # of the array to "@AS_array".
               #-------------------------------------------
               push(@AS_array, @larray[$i .. $#larray]);
               $dest_found = 1;

               last;
            }
         }

         #----------------------------------------------------
         # We only continue processing when a valid destination
         # has been found.
         #----------------------------------------------------
         if ($dest_found) {
            #-------------------------------------------------
            # The following loop looks for the originator in
            # "@AS_array" that matches any AS in the ASlist.
            #-------------------------------------------------
            for ($i=$#AS_array; $i>=0; $i--) {
               if ( $Rtg_Proc->belongs_to_AS_list($AS_array[$i]) ) {
                  #----------------------------------------
                  # When we find the originator we delete the
                  # rest of the array from "@AS_array".
                  #----------------------------------------
                  splice(@AS_array, ($i+1));

                  last;
               }
            }

            #-------------------------------------------------
            # If "@AS_array" has more than one element process
            # the path stored in this array.
            #-------------------------------------------------
            if ($#AS_array) {
               $Rtg_Proc->process_AS_array(@AS_array);
            }
         }
      }
   }
   #----------------------------------------------------------
   # Second Loop Ends
   #----------------------------------------------------------

   #----------------------------------------------------------
   # The files are so large that it is a good idea to close
   # them now.
   #----------------------------------------------------------
   close(DATA_FILE);
   print {"MESSAGES"} "   Closing $file1.\n";
}
#-------------------------------------------------------------
#-------------------------------------------------------------
# Main Loop Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



#-------------------------------------------------------------
#-------------------------------------------------------------
# Further Processing Section:
#-------------------------------------------------------------
#-------------------------------------------------------------
$valid_paths = 1;
$Rtg_Proc->dump_paths($valid_paths);

#-------------------------------------------------------------
# Let's obtain the routing information for each one of the ASs
# and convert it into an adjacency matrix. These matrices will
# be stored in separate files that can be read by Matlab (*.out)
# and Pajek (*.mat).
#-------------------------------------------------------------
foreach $ASname (@ASlist) {
   print {"MESSAGES"} "\nProcessing Adjacency for $ASname:\n";
   print "Processing Adjacency for $ASname.\n";

   @routes = $Rtg_Proc->get_valid_routes($ASname);

   for $i (0 .. $#routes) {
      #----------------------------------------------------
      # "@routes" is a two-dimensional array, hence we use
```

```
      # "$route" as a temporary index into the first dimension
      # of this array.
      # The first element of each two dimensional array
      # describes the route between each AS ("$ASname") and the
      # first element, so we initialize "$prev_vertex" to
      # "$ASname".
      #-------------------------------------------------------
      $route       = $routes[$i];
      $prev_vertex = $ASname;

      for $j (0 .. $#{$route}) {
         $ADJ_Matrices{$ASname}
                      [$AS_to_idx{$prev_vertex}]
                      [$AS_to_idx{$route->[$j]}]      = 1;

         $prev_vertex = $route->[$j];
      }
   }

   #-------------------------------------------------------------
   # The returned adjacency information for each AS is stored in
   # the files *.out and *.mat, which are stored in the "data"
   # directory For example:
   #   data/1299.out
   #   data/1299.mat
   #-------------------------------------------------------------
   $ofile = "data/" . $ASname . ".out";
   $pajek = "data/" . $ASname . ".mat";

   open(ADJ_FILE, ">$ofile") or die "Can't open $ofile: $!\n";
   open(PJK_FILE, ">$pajek") or die "Can't open $pajek: $!\n";

   #-------------------------------------------------------------
   # Add additional info for the Pajek .mat file:
   #-------------------------------------------------------------
   print {"PJK_FILE"} "*Vertices $length\n";

   for ($i=1; $i<=$length; $i++) {
      print {"PJK_FILE"} " $i \"$ASlist[$i-1]\" 0.1 0.2 0.5\n";
   }

   print {"PJK_FILE"} "*Matrix\n";

   #-------------------------------------------------------------
   # Write the adjacency matrix into both files.
   #-------------------------------------------------------------
   for ($i=0; $i<$length; $i++) {
      for ($j=0; $j<$length; $j++) {
         print {"ADJ_FILE"} "$ADJ_Matrices{$ASname}[$i][$j] ";
         print {"PJK_FILE"} "$ADJ_Matrices{$ASname}[$i][$j] ";
      }
      print {"ADJ_FILE"} "\n";
      print {"PJK_FILE"} "\n";
   }

   close(ADJ_FILE);
   close(PJK_FILE);
}

#-----------------------------------------------------------------
#-----------------------------------------------------------------
# Further Processing Section Ends
#-----------------------------------------------------------------
#-----------------------------------------------------------------



close(MESSAGES);
print "\n *** AMG2.pl Finished! ***\n";

#================================================================
################################################################
#================================================================
# Main Program finishes here
#================================================================
################################################################
#================================================================
```

B.8 Perl module: RD_CLASS.pm

```
package  RD_CLASS;
require  Exporter;

our @ISA = qw(Exporter);

#================================================================
#
# Class: RD_CLASS
#
#================================================================
# This class stores and manages the data structure that stores
# the Routing Data for each AS.
#----------------------------------------------------------------
# Data Structure description:
#
# Rtg_Data: Structure that holds the Routing Data.
#           It contains 2 elements:
#    the_AS_list: List of ASs that are valid for this study.
#    Autonomous_System: This array holds the information for all
#           the Autonomous Systems. It is really an array of
#           anonimous hashes. Each hash contains the following
#           elements:
#        AS_NAME    = The AS from which all the paths originate.
#        VALID_NUM  = The number of different routing paths
#                     included for this AS.
#        VALID_PATH = An array of routing paths (also an array).
#----------------------------------------------------------------
# Created by: David Arjona
# Date: 24/October/2007
#----------------------------------------------------------------



#================================================================
# new: Constructor for class RD_CLASS.
#================================================================
# Input:
#    None.
#----------------------------------------------------------------
# Output:
#    The class.
#----------------------------------------------------------------
sub new {
   my $obj = shift;

   my $class = ref($obj) || $obj;

   #-------------------------------------------------------------
   # This file store all the invalid paths found while running
   # this script.
   #-------------------------------------------------------------
   open(INVALID, ">inv_path.txt") or die "Can't create inv_path.txt file: $!\n";

   #-------------------------------------------------------------
   # This file contains all the logs and related information
   # generated while processing the data files.
   #-------------------------------------------------------------
   open(MSG, ">RD_CLASS.err") or die "Can't create RD_CLASS.err file: $!\n";

   print {"MSG"} "================================================\n";
   print {"MSG"} "   This is the log file for module: RD_CLASS.pm\n";
   print {"MSG"} "================================================\n\n";

   #-------------------------------------------------------------
   # File "AS_list" contains the list of AS that should be
   # included in "the_AS_list".
   #-------------------------------------------------------------
   open(AS_LIST_FILE, "AS_list") or die "Can't open AS list file: $!\n";

   while ($ASnum = <AS_LIST_FILE>) {
      chomp($ASnum);
      $ASnum =~ s/AS//;
      push(@the_list, $ASnum);
   }

   close(AS_LIST_FILE);

   #-------------------------------------------------------------
   # The Rtg_Data data structure is created here.
   #-------------------------------------------------------------
   my $Rtg_Data = {
       the_AS_list   => [ @the_list ],
       #----------------------------------------------------------
       # The following hash structure is created when each
       # Routing Data is added. The following is just an
       # example and reminder of how this structure is formed.
       #----------------------------------------------------------
       #   Autonomous_System[0] = {
       #      AS_NAME    => 1801,
       #      VALID_NUM  => 3,
       #      VALID_PATH => ((123, 345), 6789, (702, 1299, 2122)),
       #   },
       #----------------------------------------------------------
```

```
   };


   bless($Rtg_Data, $class);
   return $Rtg_Data;
}




#============================================================
# belongs_to_AS_list: This method verifies that the passed AS
#    belongs to the valid list.
#============================================================
# Input:
#    1. An Autonomous System Number (ASN).
#------------------------------------------------------------
# Output:
#    This method returns 1 if the ASN belongs to the valid list,
#    or 0 (zero) otherwise.
#------------------------------------------------------------
sub belongs_to_AS_list {
   my $obj = shift;
   my $ASN = shift;

   #-----------------------------------------------------------
   # Check if "ASN" is equal to any of the ASs in "the_AS_list".
   #-----------------------------------------------------------
   foreach $AS ( @{$obj->{the_AS_list}} ) {
      if ($AS eq $ASN) {
         return 1;
      }
   }

   return 0;
}




#============================================================
# find_AS: This method finds if the Autonomous System has
#     previously been created into the data structure.
#============================================================
# Input:
#    An Autonomous System ($_[0]).
#------------------------------------------------------------
# Output:
#    This method returns
#
#
#------------------------------------------------------------
sub find_AS {
   my $obj = shift;

   #-----------------------------------------------------------
   # Check if the AS already exists in the data structure.
   #-----------------------------------------------------------
   for $l (0 .. $#{$obj->{Autonomous_System}} ) {
      if ($_[0] eq $obj->{Autonomous_System}[$l]{AS_NAME}) {
         return $l;
      }
   }

   return -1;
}




#============================================================
# find_similar_paths: This method scans through the paths (valid
#     or invalid) of an Autonomous System, looking for a match
#     between the path's elements and another autonomous system
#     which is passed as an argument. It returns an array with
#     the indices of the similar paths found.
#============================================================
# Input:
#    1. The Autonomous System ($ASN) we need to process.
#    2. The AS ($AS) we are trying to match in the current
#       paths.
#------------------------------------------------------------
# Output:
#    An array (@rt_array) which contains the index of the paths
#    where a matching AS was found.
#------------------------------------------------------------
sub find_similar_paths {
   my $obj   = shift;
   my $idx   = shift;
   my $AS    = shift;

   my @rt_array = ();

   #-----------------------------------------------------------
   # Search through all the valid paths.
   #-----------------------------------------------------------
   FIRST_LOOP: for $m (0 .. $#{$obj->
               {Autonomous_System}[$idx]{VALID_PATH}}) {
      #--------------------------------------------------------
      # Search through all the ASs in the path.
      #--------------------------------------------------------
      for $n (0 .. $#{$obj->{Autonomous_System}[$idx]{VALID_PATH}[$m]}) {
         #-----------------------------------------------------
         # When we find a match we pass the index ($m) to the
         # return array: "@rt_array".
         #-----------------------------------------------------
         if ($AS eq $obj->{Autonomous_System}[$idx]{VALID_PATH}[$m][$n]) {
            push(@rt_array, $m);
            next FIRST_LOOP;
         }
      }
   }

   return @rt_array;
}




#============================================================
# process_similar_paths: This method checks how similar the
#     paths are and takes action to either store the new path
#     without modification, modify a path before storing or
#     return nothing to be stored.
#     The processing for valid and invalid paths is completely
#     different because for invalid paths we are not interested
#     on do further processing at the moment of writing this
#     code.
#     The processing for valid paths needs to verify all the
#     possibilities that can happen between similar paths.
#============================================================
# Input:
#    1. The Autonomous System ($ASN) which stores one of the
#       paths we need to compare.
#    2. The index ($index) of the stored path we need to
#       compare.
#    3. The other path we are comparing (@_) which we call the
#       current path.
#------------------------------------------------------------
# Output:
#    A modified version of the current path array (@_).
#    Depending on the processing it is possible that this array
#    is not modified, slightly modified or even returned as an
#    empty array. The function that calls this method should
#    interpret the returning array (@rt_array) as the result
#    that should be added into the data structure.
#------------------------------------------------------------
sub process_similar_paths {
   my $obj   = shift;
   my $idx   = shift;
   my $index = shift;

   my @rt_array = ();

   #-----------------------------------------------------------
   # A_lgth: The length of the array currently stored in the
   #    data structure.
   # B_lgth: The length of the array we are currently processing
   #    which has been passed as an argument.
   # identical: We assume that both arrays have identical
   #    information until we can demonstrate otherwise.
   #-----------------------------------------------------------
   my $A_lgth = $#{$obj->{Autonomous_System}[$idx]{VALID_PATH}[$index]};
   my $B_lgth = $#_;
   my $identical = 1;

      #--------------------------------------------------------
      # Do both routes start with the same ASs? Then they are
      # either:
      #    a) identical
```

```
#    b) one is included in the other (their lengths should
#       be different)
#    c) they start with similar ASs and then divide and
#       have different destinations.
#-------------------------------------------------------
if ($obj->{Autonomous_System}[$idx]{VALID_PATH}[$index][0] eq $_[0]) {
   #----------------------------------------------------
   # Who is larger?
   # The shortest length is stored in "$length".
   #----------------------------------------------------
   if ($A_lgth >= $B_lgth) {
      #-------------------------------------------------
      # The current path is larger or has same length.
      #-------------------------------------------------
      $length = $B_lgth;
   }
   else {
      #-------------------------------------------------
      # The new path is larger! (It may contain more
      # information).
      #-------------------------------------------------
      $length = $A_lgth;
   }

   #----------------------------------------------------
   # Are the paths identical? (Up to the size of $length).
   #----------------------------------------------------
   for ($l=1; $l<=$length; $l++) {
      if ($obj->{Autonomous_System}[$idx]{VALID_PATH}[$index][$l] ne
          $_[$l]) {
         #-------------------------------------------------
         # The paths are different!
         #-------------------------------------------------
         $identical = 0;
         last;
      }
   }

   #----------------------------------------------------
   # Let's do more processing...
   # Note: The only option we do not need to process is if
   # the path stored has more or equal info than the
   # current path ($identical=true and $A_lgth>=$B_lgth),
   # in which case we just ignore the current path and
   # return an empty array (@rt_array = ()).
   #----------------------------------------------------
   if ($identical) {
      #-------------------------------------------------
      # The current path is larger than the one stored but
      # they contain identical information up to $length.
      # This means the current path has more information
      # and should REPLACE the path stored.
      #-------------------------------------------------
      if ($B_lgth > $A_lgth) {
         @{$obj->{Autonomous_System}[$idx]{VALID_PATH}[$index]} = @_;
      }
   }
   else {
      #-------------------------------------------------
      # The paths have different info. Return the current
      # path so it can be added to the data structure.
      #-------------------------------------------------
      @rt_array = @_;
   }
}

#-------------------------------------------------------
# Both paths have common elements at some point. There are
# 2 options:
#    a) Starting from the lst common element the paths
#       have similar routing information up to certain
#       element. In this case we may delete common
#       elements from the shortest common path.
#    b) The paths are different.
# Where are the common elements?
#-------------------------------------------------------
else {
   EXIT: for ($m=0; $m<=$A_lgth; $m++) {
      for ($n=0; $n<=$B_lgth; $n++) {
         if ($obj->{Autonomous_System}[$idx]{VALID_PATH}[$index][$m]
             eq $_[$n]) {
            #-------------------------------------------
            # We have found the lst common element, let's
            # keep track of their location.
            #-------------------------------------------
            $A_idx = $m;
            $B_idx = $n;
            last EXIT;
         }
      }
   }
}

#-------------------------------------------------------
# Adjust the length so we can find who has the longest
# common path.
#-------------------------------------------------------
$A_lgth = $A_lgth - $A_idx;
$B_lgth = $B_lgth - $B_idx;

#-------------------------------------------------------
# Who is larger?
# The shortest common length is stored in "$length".
#-------------------------------------------------------
if ($A_lgth >= $B_lgth) {
   #----------------------------------------------------
   # The current path is larger or has same length.
   #----------------------------------------------------
   $length = $B_lgth;
}
else {
   #----------------------------------------------------
   # The new path is larger! (It may contain more
   # information).
   #----------------------------------------------------
   $length = $A_lgth;
}

#-------------------------------------------------------
# Are the paths identical? (Up to the size of $length.
#-------------------------------------------------------
for ($l=1; $l<=$length; $l++) {
   if (  $obj->{Autonomous_System}[$idx]
         {VALID_PATH}[$index][$l+$A_idx]
      ne $_[$l+$B_idx] ) {
      #----------------------------------------------
      # The paths are different!
      #----------------------------------------------
      $identical = 0;
      last;
   }
}

#-------------------------------------------------------
# Let's do more processing...
#-------------------------------------------------------
if ($identical) {
   if ($B_lgth > $A_lgth) {
      #----------------------------------------------
      # The current path is longer than the stored
      # path, e.g.:
      #
      #  Path_A = 22 -> 13 -> 7 -> 8 -> 9       (stored)
      #  Path_B =       11 -> 7 -> 8 -> 9 -> 33 (current)
      #
      # We should delete the duplicated info from the
      # stored path and return the current path so it
      # can be added to the data structure.
      #----------------------------------------------
      splice(@{$obj->{Autonomous_System}[$idx]
                    {VALID_PATH}[$index]}, ($A_idx+1));
      @rt_array = @_;
   }
   else {
      #----------------------------------------------
      # The current path is shorter than the stored
      # path, e.g.:
      #
      #  Path_A = 22 -> 13 -> 7 -> 8 -> 9    (stored)
      #  Path_B =       11 -> 7 -> 8         (current)
      #
      # We should delete the duplicated info from the
      # current path and return it, so it can be added
      # to the data structure. The stored path remains
      # unchanged.
      #----------------------------------------------
      @rt_array = @_;
      splice(@rt_array, ($B_idx+1));
   }
}
else {
   #----------------------------------------------
   # The paths have different info. Return the current
   # path so it can be added to the data structure.
   #----------------------------------------------
```

```
            @rt_array = @_;
         }
      }

   return @rt_array;
}



#============================================================
# add_Routing_Data: This method controls how and where data
#    is added into the Rtg_Data data structure.
#    First it needs to verify if the Autonomous System (ASN)
#    already exists in the data structure or if this ASN needs
#    to be created. If this ASN already exist  in the data
#    structure we need to process the path in order to find if
#    there are other paths which contain the same or similar
#    information. Finally, it add new routing data to the data
#    structure once all the check have completed and passed.
#============================================================
# Input:
#    1. An Autonomous System (ASN) to which the path needs to be
#       added.
#    2. The path that needs to be added.
#------------------------------------------------------------
# Output:
#    This method returns 1 by default.
#------------------------------------------------------------
sub add_Routing_Data {
   my $obj = shift;
   my $ASN = shift;

   #-----------------------------------------------------------
   # notvalid: This counter is increased for each not valid
   #    AS which is added into the "@nv_path" array.
   # path: This array stores the ASs which need to be included
   #    as VALID_PATH.
   # nv_path: This array stores the ASs which need to be
   #    included as NONVD_PATH.
   # similar: This array stores the index into similar paths to
   #    the one stored by $path.
   #-----------------------------------------------------------
   my $notvalid = 0;
   my @path     = ();
   my @nv_path  = ();
   my @similar  = ();

   #-----------------------------------------------------------
   # First we need to find the index of the AS we are trying to
   # modify or write to.
   #-----------------------------------------------------------
   my $AS_idx = $obj->find_AS($ASN);

   #-----------------------------------------------------------
   #-----------------------------------------------------------
   # Main IF statement
   #-----------------------------------------------------------
   #-----------------------------------------------------------
   # IF this AS does NOT exists in the "Autonomous_System" hash
   # data structure, we have to create it and add its first
   # path(s).
   #-----------------------------------------------------------
   if ($AS_idx == -1) {
      #--------------------------------------------------------
      # FOR Loop 2: Is the path valid?
      #--------------------------------------------------------
      #--------------------------------------------------------
      # Process each one of the ASs in the routing path (@_)
      # starting from the closest to "ASN" (which is the largest
      # index in the @_ array).
      #--------------------------------------------------------
      VALID2: for ($j=$#_; $j>=0; $j--) {
         #-----------------------------------------------------
         # This if statement skips duplicated ASs in the path.
         # The last element in the array needs to be checked
         # against "$ASN" (in case the ASN is duplicated).
         #-----------------------------------------------------
         if ($j == $#_) {
            if ($_[$j] eq  $ASN) {
               next VALID2;
            }
         }
         else {
            if ($_[$j] eq  $_[($j+1)]) {
               next VALID2;
            }
         }

         #-----------------------------------------------------
         # Is the current AS NOT valid?
         #-----------------------------------------------------
         if (! $obj->belongs_to_AS_list($_[$j]) ) {
            $notvalid++;
         }

         #-----------------------------------------------------
         # Non valid paths are stored in the "@nv_path" array.
         #-----------------------------------------------------
         if ($notvalid) {
            #--------------------------------------------------
            # The first element of a non valid path is either
            # the last element (AS) of the valid path (if it
            # exists), or "ASN".
            #--------------------------------------------------
            if ($notvalid == 1) {
               if (@path) {
                  $nv_path[0] = $path[$#path];
               }
               else {
                  $nv_path[0] = $ASN;
               }
            }

            #--------------------------------------------------
            # Add the current AS to "@nv_path" and check if
            # there are similar invalid paths.
            #--------------------------------------------------
            push(@nv_path, $_[$j]);
         }
         #-----------------------------------------------------
         # Valid paths are stored in the "@path" array.
         #-----------------------------------------------------
         else {
            push(@path, $_[$j]);
         }
      }
      #--------------------------------------------------------
      # End FOR Loop 2: Is the path valid?
      #--------------------------------------------------------


      #--------------------------------------------------------
      # A non valid path is printed in the inv_path.txt file.
      #--------------------------------------------------------
      if (@nv_path) {
         #-----------------------------------------------------
         # Let's print out what we have just created.
         #-----------------------------------------------------
         foreach $AS (@nv_path) {
            print {"INVALID"} "$AS ";
         }
         print {"INVALID"} "\n";
      }

      #--------------------------------------------------------
      # A valid path is stored in the data structure.
      #--------------------------------------------------------
      if (@path) {
         push( @{$obj->{Autonomous_System}}, {
                     AS_NAME    => $ASN,
                     VALID_NUM  => 1,
                     VALID_PATH => [[ @path ]]
         });

         #-----------------------------------------------------
         # Let's print out what we have just created.
         #-----------------------------------------------------
         print {"MSG"} "New Valid path: $ASN";
         foreach $AS (@{$obj->{Autonomous_System}[-1]{VALID_PATH}[0]}) {
            print {"MSG"} "->$AS";
         }
         print {"MSG"} "\n";
      }
   }

   #-----------------------------------------------------------
   # This AS already exists in the data structure, let's see
   # if we need to add this new path.
   #-----------------------------------------------------------
   else {
      #--------------------------------------------------------
      # FOR Loop 1: Is the path valid?
```

```
#-------------------------------------------------------
# Process each one of the ASs in the routing path (@_)
# starting from the closest to "ASN" (which is the largest
# index in the @_ array).
#-------------------------------------------------------
VALID1: for ($j=$#_; $j>=0; $j--) {
   #----------------------------------------------------
   # This if statement skips duplicated ASs in the path.
   # The last element in the array needs to be checked
   # against "$ASN" (in case the ASN is duplicated).
   #----------------------------------------------------
   if ($j == $#_) {
      if ($_[$j] eq  $ASN) {
         next VALID1;
      }
   }
   else {
      if ($_[$j] eq  $_[($j+1)]) {
         next VALID1;
      }
   }

   #----------------------------------------------------
   # Is the current AS NOT valid?
   #----------------------------------------------------
   if (! $obj->belongs_to_AS_list($_[$j]) ) {
      $notvalid++;
   }

   #----------------------------------------------------
   # Non valid paths are stored in array "@nv_path".
   #----------------------------------------------------
   if ($notvalid) {
      #----------------------------------------------------
      # The first element of a non valid path is either
      # the last element (AS) of the valid path (if it
      # exists), or "ASN".
      #----------------------------------------------------
      if ($notvalid == 1) {
         if (@path) {
            $nv_path[0] = $path[$#path];
         }
         else {
            $nv_path[0] = $ASN;
         }
      }

      #----------------------------------------------------
      # Add the current AS to "@nv_path" and check if
      # there are similar invalid paths.
      #----------------------------------------------------
      push(@nv_path, $_[$j]);
   }
   #----------------------------------------------------
   # Valid paths are stored in array "@path".
   #----------------------------------------------------
   else {
      #----------------------------------------------------
      # This is still a valid path. Add the current AS to
      # "@path" and check if there are similar paths.
      #----------------------------------------------------
      push(@path, $_[$j]);

      @tmp_sim = $obj->find_similar_paths($AS_idx, $_[$j]);

      if (@similar) {
         #----------------------------------------------------
         # The following loop checks that only new indexes
         # are copied from "@tmp_sim" to "@similar".
         #----------------------------------------------------
         STORED: for $idx (@tmp_sim) {
            for ($jdx=0; $jdx<=$#similar; $jdx++) {
               if ($idx eq $similar[$jdx]) {
                  next STORED;
               }
               push(@similar, $idx);
            }
         }
      }
      else {
         #----------------------------------------------------
         # "@similar" is empty the first time we need to
         # write to it.
         #----------------------------------------------------
         @similar = @tmp_sim;
      }
   }
   }
   #----------------------------------------------------
   # End FOR Loop 1: Is the path valid?
   #----------------------------------------------------


   #----------------------------------------------------
   # A non valid path is printed in the inv_path.txt file.
   #----------------------------------------------------
   if (@nv_path) {
      #----------------------------------------------------
      # Let's print out the invalid path we have just added.
      #----------------------------------------------------
      foreach $AS ( @nv_path ) {
         print {"INVALID"} "$AS ";
      }
      print {"INVALID"} "\n";
   }

   #----------------------------------------------------
   # If "@similar" contains any index to similar paths,
   # process "@path" against previous stored paths by calling
   # the "process_similar_paths" method.
   #----------------------------------------------------
   if (@similar) {
      foreach $idx (@similar) {
         @path = $obj->process_similar_paths($AS_idx, $idx, @path);

         #----------------------------------------------------
         # If the resulting "@path" is empty it means that
         # we have found a path or groups of paths that
         # contain the same routing information than the
         # original path.
         #----------------------------------------------------
         if (! @path) {
            last;
         }
      }
   }

   #----------------------------------------------------
   # Let's add the resulting path. If "@similar" was true
   # this path may have been modified, otherwise this must be
   # a path that did not belong to the data structure before.
   #----------------------------------------------------
   if (@path) {
      $obj->{Autonomous_System}[$AS_idx]{VALID_NUM}++;
      push( @{$obj->{Autonomous_System}[$AS_idx]{VALID_PATH}}, [ @path ]);

      #----------------------------------------------------
      # Let's print out the valid path we have just added.
      #----------------------------------------------------
      print {"MSG"} "    New Valid path: $ASN";
      foreach $AS
      ( @{$obj->{Autonomous_System}[$AS_idx]{VALID_PATH}
                [$obj->{Autonomous_System}[$AS_idx]{VALID_NUM}-1]} ) {
         print {"MSG"} "->$AS";
      }
      print {"MSG"} "\n";
   }
  }
  #----------------------------------------------------
  #----------------------------------------------------
  # End Main IF statement
  #----------------------------------------------------
  #----------------------------------------------------

  return 1;
}



#============================================================
# process_AS_array: This method, usually called from a main
#    module, is responsible for processing the AS array which
#    is passed as an argument.
#============================================================
# Input:
#    An array of Autonomous Systems.
#------------------------------------------------------------
# Output:
#    This method returns 1 by default.
#------------------------------------------------------------
sub process_AS_array {
   my $obj = shift;
```

```
   #-------------------------------------------------------------
   # For each one of the ASs in the array: if the AS belongs to
   # the AS list, process its routing path from the previous AS
   # in the array to the destination (array's zero element).
   # If the AS does not belong to the AS list, we just skip it
   # and it will be processed when the next AS tries to include
   # it on its routing path.
   #-------------------------------------------------------------
   for ($k=1; $k<=$#_; $k++) {
      if ($obj->belongs_to_AS_list($_[$k])) {
         @rtg_path = @_[0 .. ($k-1)];
         $obj->add_Routing_Data($_[$k], @rtg_path);
      }
   }

   return 1;
}




#===============================================================
# get_valid_routes: This method returns the valid routes for a
#    specific AS wich is passed as an argument. This is the
#    method we should use to read the information contained in
#    the Rtg_Data data structure.
#===============================================================
# Input:
#    An Autonomous System (ASN).
#---------------------------------------------------------------
# Output:
#    This method returns a two-dimensional array which contains
#    the routes stored in the data structure, e.g.:
#       ASN = 1234
#          VALID_PATH[0] = [ 234 9911 ]
#          VALID_PATH[1] = [ 8761 9911 321 ]
#---------------------------------------------------------------
sub get_valid_routes {
   my $obj = shift;
   my $ASN = shift;

   #-------------------------------------------------------------
   # First we need to find the index of the AS we are trying to
   # read from.
   #-------------------------------------------------------------
   my $AS_idx = $obj->find_AS($ASN);

   return @{$obj->{Autonomous_System}[$AS_idx]{VALID_PATH}};
}




#===============================================================
# dump_paths: This method dumps either the valid or the invalid
#     paths in the file RD_CLASS.out.
#     This method has been designed for debugging purposes.
#===============================================================
# Input:
#    None.
#---------------------------------------------------------------
# Output:
#    The output of this method is stored in file RD_CLASS.out.
#    This method returns 1 by default.
#---------------------------------------------------------------
sub dump_paths {
   my $obj   = shift;

   #-------------------------------------------------------------
   # The output file for this method.
   #-------------------------------------------------------------
   open(DUMP, ">RD_CLASS.out") or die "Can't create RD_CLASS.out file: $!\n";

   print {"DUMP"} " ================================================\n";
   print {"DUMP"} "    This is the dump file for module: RD_CLASS.pm\n";
   print {"DUMP"} " ================================================\n\n";

   #-------------------------------------------------------------
   # Let's dump valid paths.
   #-------------------------------------------------------------
   foreach $ASN (@{$obj->{the_AS_list}}) {
      print {"DUMP"} "Valid paths for AS: $ASN\n";

      #----------------------------------------------------------
      # We need to find the index of the AS we are trying
      # to read from.
      #----------------------------------------------------------
      my $AS_idx = $obj->find_AS($ASN);

      #----------------------------------------------------------
      # Scan through all the valid paths.
      #----------------------------------------------------------
      for $m (0 .. $#{$obj->{Autonomous_System}[$AS_idx]{VALID_PATH}}) {
         print {"DUMP"} "   $ASN";
         #-------------------------------------------------------
         # Scan through all the ASs in this path.
         #-------------------------------------------------------
         for $n (0 .. $#{$obj->{Autonomous_System}[$AS_idx]
                         {VALID_PATH}[$m]}) {
            print {"DUMP"}
            "->$obj->{Autonomous_System}[$AS_idx]{VALID_PATH}[$m][$n]";
         }
         print {"DUMP"} "\n";
      }
   }

   close(DUMP);

   return 1;
}

1;
```

B.9 Modified Perl script: skitter.pl

```
#!/usr/bin/perl -w
#
##############################################################################
## Copyright 2004
## The Regents of the University of California
## All Rights Reserved
##
## Permission to use, copy, modify and distribute any part of
## this software package for educational, research and non-profit
## purposes, without fee, and without a written agreement is hereby
## granted, provided that the above copyright notice, this paragraph
## and the following paragraphs appear in all copies.
##
## Those desiring to incorporate this into commercial products or
## use for commercial purposes should contact the Technology Transfer
## Office, University of California, San Diego, 9500 Gilman Drive,
## La Jolla, CA 92093-0910, Ph: (858) 534-5815, FAX: (858) 534-7345.
##
## IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY
## PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL
## DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF THE USE OF THIS
## SOFTWARE, EVEN IF THE UNIVERSITY OF CALIFORNIA HAS BEEN ADVISED
## OF THE POSSIBILITY OF SUCH DAMAGE.
##
## THE SOFTWARE PROVIDED HEREIN IS ON AN "AS IS" BASIS, AND THE
## UNIVERSITY OF CALIFORNIA HAS NO OBLIGATION TO PROVIDE MAINTENANCE,
## SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS. THE UNIVERSITY
## OF CALIFORNIA MAKES NO REPRESENTATIONS AND EXTENDS NO WARRANTIES
## OF ANY KIND, EITHER IMPLIED OR EXPRESS, INCLUDING, BUT NOT LIMITED
## TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A
## PARTICULAR PURPOSE, OR THAT THE USE OF THE SOFTWARE WILL NOT
## INFRINGE ANY PATENT, TRADEMARK OR OTHER RIGHTS.
##
## This software package is developed by the CAIDA development team
## at the University of California, San Diego under the Cooperative
## Association for Internet Data Analysis (CAIDA) Program. Support
## for this effort is provided by the NSF grant ANI-0221172 and by
## CAIDA members.
##
## Report bugs and suggestions to info@caida.org.
##
## Written by Dmitri Krioukov <dima@caida.org> 05/26/04
##
##############################################################################
##############################################################################
#
#  DESCRIPTION:
#              filter AS adjacency files into AS graph adjacency matrices
#
#  INPUT:
#              AS adjacency files:
#              http://www.caida.org/tools/measurement/skitter/as_adjacencies.xml
#
#  OUTPUT:
#              AS graph adjacency matrices in the following format: line
```

```
#                AS_X   AS_Y
#             represents a link between AS_X and AS_Y
#
##############################################################################

#use strict;
use warnings;
use Getopt::Std;
our %opts;
my $input_dir   = ".";
my $input_file  = "skitter_as_links.*.gz";
my $output_dir  = ".";
my $output_file = "skitter_as_graph";
my $start_date  = 00000000;
my $end_date    = 99999999;
my %graph;

my $ASnum  = "";
my $AS     = "";
my $t1     = "";
my $t2     = "";
my $ofile  = "";
my $pajek  = "";
my $length = 0;
my @ASlist = ();
my $jump   = 0;
my $x      = 0;
my $z      = 0;
my %ADJ_Row;
my %AS_to_idx;

sub PrintUsage {
  print " usage: $0 {-d|-i length} [-smpnou] [-l input] [-g output] [-r dates]\n";
  print "  -d:  include direct links\n";
  print "  -i:  include indirect links of maximum 'length'\n";
  print "         'length' = -1 to include all indirect links\n";
  print "  -s:  include links to/from AS-sets\n";
  print "  -m:  include links to/from MOASs\n";
  print "  -p:  include private ASs\n";
  print "  -o:  output = stdout (and input = stdin if no '-u' option)\n";
  print "  -l:  search for link files named $input_file\n";
  print "         below 'input' directory\n";
  print "  -g:  place output files in 'output' directory\n";
  print "  -u:  merge links in all input files into one 'output' file\n";
  print "  -n:  neglect direction in the input files, produce undirected graphs\n";
  print "  -r:  inclusive range of 'dates' of input files\n";
  print "         'dates' must be YYYYMMDD:yyyymmdd (start date:end date)\n";
  return 1;
}

if (!getopts("di:smpuonl:g:r:", \%opts)
     or (!defined $opts{'d'} and !defined $opts{'i'})
     or (defined $opts{'i'} and !($opts{'i'} =~ /[-]*\d+/))
     or (defined $opts{'r'} and !($opts{'r'} =~ /\d{8}:\d{8}/))
   ) {
  exit PrintUsage();
}

if (defined $opts{'o'} and !defined $opts{'u'}) { # standard input/output
  MergeGraph(\*STDIN, \%graph);
  PrintGraph(\*STDOUT, \%graph);
  exit 0;
}

if (defined $opts{'l'}) { # input directory
  $input_dir = $opts{'l'};
}

if (defined $opts{'g'}) { # output directory
  $output_dir = $opts{'g'};
}
if (! -e $output_dir) {
  mkdir $output_dir;
}
elsif (! -d $output_dir) {
  die("$output_dir exists and is not a directory");
}
if (!($output_dir =~ /\/$/)) {
  $output_dir .= '/';
}

if (defined $opts{'r'}) { # range of dates
  ($start_date, $end_date) = split /:/, $opts{'r'};
}

if (defined $opts{'i'}) {
  $output_file .= ".indirect:$opts{'i'}";
}
if (defined $opts{'m'}) {
  $output_file .= ".moass";
}
if (defined $opts{'s'}) {
  $output_file .= ".assets";
}
if (defined $opts{'p'}) {
  $output_file .= ".private";
}
if (defined $opts{'n'}) {
  $output_file .= ".undirected";
}
else {
  $output_file .= ".directed";
}
if (defined $opts{'u'}) {
  $output_file .= ".merge";
  if (defined $opts{'r'}) {
    $output_file .= ".dates:$start_date-$end_date"
  }
}

my %date_files = GetFiles($input_dir, $input_file);

if (defined $opts{'u'}) { # merge links from all input files

  foreach my $date (reverse sort {$a<=>$b} keys %date_files) {
    next if ($date < $start_date or $date > $end_date);
    my $input = $date_files{$date};
    open(IN, "gunzip -c $input |") || die("unable to open \"$input\"");
    MergeGraph(\*IN, \%graph);
    close(IN);
  }

  if (!defined $opts{'o'}) { # print to a file
    my $output = $output_dir . $output_file . ".gz";
    open(OUT, "| gzip -c >$output") || die("unable to open \"$output\"");
    PrintGraph(\*OUT, \%graph);
    close(OUT);
  }
  else {
    PrintGraph(\*STDOUT, \%graph);
  }

  exit 0;
}

# print file-by-file
foreach my $date (reverse sort {$a<=>$b} keys %date_files) {
  next if ($date < $start_date or $date > $end_date);
  my $input = $date_files{$date};
  my $output = $output_dir . $output_file . "." . $date . ".gz";
  open(IN, "gunzip -c $input |") || die("unable to open \"$input\"");
  open(OUT, "| gzip -c >$output") || die("unable to open \"$output\"");
  undef %graph;
  MergeGraph(\*IN, \%graph);
  PrintGraph(\*OUT, \%graph);
  close(IN);
  close(OUT);
}
exit 0;

# search for input files
sub GetFiles {
  my ($dir, $file_pattern) = @_;
  my %date2file;
  foreach my $file (`find $dir -name '$file_pattern'`) {
    if ($file =~ /(\d{8})/) {
      my $date = $1;
      $date2file{$date} = $file;
    }
  }
  return %date2file;
}

# merge links from file into the graph
sub MergeGraph {
  my ($input_fh, $graph) = @_;

  #------------------------------------------------------------
  # Modification 1 by P. D. Arjona-Villicana:
  #------------------------------------------------------------
  # Code to read "ASlist" from AS_list file. And to initialize
  # the other data structures that we will need later.
  #------------------------------------------------------------
  open(AS_LIST_FILE, "AS_list") or die "Can't open AS_list file: $!\n";
```

```
 while ($ASnum = <AS_LIST_FILE>)
{
    chomp($ASnum);
    $ASnum =~ s/AS//;
    $length = push(@ASlist, $ASnum);
 }

 close(AS_LIST_FILE);
 print "\nAS list size: $length\n";

#-------------------------------------------------------------
# The row we will build in the Adjacency Matrix. We only use
# rows because skitter can only produce direct links from
# A to B but cannot guarantee that the routing info may be
# passed to another AS C.
#-------------------------------------------------------------
foreach $AS (@ASlist)
{
   for $x (0 .. ($length-1))
   {
      $ADJ_Row{$AS}[$x] = 0;
   }

   $AS_to_idx{$AS} = $z++;
}
#-------------------------------------------------------------
# Modification 1 Ends
#-------------------------------------------------------------

while (<$input_fh>) { # actual filtering
  s/#.*//;
  next if (/^\s*$/ or /null/);
  next if (!defined $opts{'s'} and (/,/ or /\{/ or /\}/)); # AS-sets
  next if (!defined $opts{'m'} and /_/); # MOASs
  if ( ( (defined $opts{'d'} and /^D\s+(\S+)\s+(\S+)/) # direct links
         or
         (defined $opts{'i'} and /^I\s+(\S+)\s+(\S+)\s+(\d+)/
           and ($opts{'i'} < 0 or $3 <= $opts{'i'})
         ) # indirect links
       )
       and ($1 ne $2) # filter links from a node to itself
     ) {

    #-----------------------------------------------------
    # Modification 2 by P. D. Arjona-Villicana:
    #-----------------------------------------------------
    # Check if $1 and $2 are equal to any of the ASs in
    # "ASlist". And store the routing info into the ADJ_Row
    # hash.
    #-----------------------------------------------------
    $jump = 0;

    foreach $AS (@ASlist)
    {
       if ( ($AS eq $1)      ||
            ($1 =~ /($AS)_/) ||
            ($1 =~ /_$AS/)    )
       {
          $t1 = $AS;
          $jump++;
          next;
       }

       if ( ($AS eq $2)       ||
            ($2 =~ /($AS)_/) ||
            ($2 =~ /_$AS/)    )
       {
          $t2 = $AS;
          $jump++;
          next;
       }
    }


    if ($jump < 2)
    {
       #-----------------------------------------------------
       # This is the most probable outcome.
       #-----------------------------------------------------
       next;
    }
    elsif ($jump == 2)
    {
       #-----------------------------------------------------
       # We have found a direct link.
       #-----------------------------------------------------
          print "$t1->$t2   ";

          $ADJ_Row{$t1}[$AS_to_idx{$t2}] = 1;
       }
       else
       {
          #-----------------------------------------------------
          # $jump is greater than 2. This is not normal!
          #-----------------------------------------------------
          print "\nError 1: Variable jump is greater than 2.\n";
       }
       #-----------------------------------------------------
       # Modification 2 Ends
       #-----------------------------------------------------

       my ($from, $to) = ($1, $2); # ($1, $2) is a link from $1 to $2
       # private ASs:
       next if (!defined $opts{'p'} and ContainsPrivateAS($from, $to));
       if (defined $opts{'n'}) { # neglect direction
($from, $to) = sort MixedSort ($from, $to);
       }
       $graph->{$from}{$to} = 1;
    }
  }

  #-------------------------------------------------------------
  # Modification 3 by P. D. Arjona-Villicana:
  #-------------------------------------------------------------
  # Generate the output files
  #-------------------------------------------------------------
  foreach $AS (@ASlist)
  {
   #-----------------------------------------------------------
   # The returned adjacency information for each AS is stored in
   # the files *.out and *.mat, which are stored in the "data"
   # directory For example:
   #   data/1299.out
   #   data/1299.mat
   #-----------------------------------------------------------
   $ofile = "data/" . $AS . ".out";
   $pajek = "data/" . $AS . ".mat";

   open(ADJ_FILE, ">$ofile") or die "Can't open $ofile: $!\n";
   open(PJK_FILE, ">$pajek") or die "Can't open $pajek: $!\n";

   #-----------------------------------------------------------
   # Add additional info for the Pajek .mat file:
   #-----------------------------------------------------------
   print {"PJK_FILE"} "*Vertices $length\n";

   for ($x=1; $x<=$length; $x++)
   {
      print {"PJK_FILE"} " $x \"$ASlist[$x-1]\" 0.1 0.2 0.5\n";
   }

   print {"PJK_FILE"} "*Matrix\n";

   #-----------------------------------------------------------
   # Write the adjacency matrix into both files.
   #-----------------------------------------------------------
   for ($x=0; $x<$length; $x++)
   {
      if ($AS eq $ASlist[$x])
      {
         for ($z=0; $z<$length; $z++)
         {
            print {"ADJ_FILE"} "$ADJ_Row{$AS}[$z] ";
            print {"PJK_FILE"} "$ADJ_Row{$AS}[$z] ";
         }
      }
      else
      {
         for ($z=0; $z<$length; $z++)
         {
            print {"ADJ_FILE"} "0 ";
            print {"PJK_FILE"} "0 ";
         }
      }

      print {"ADJ_FILE"} "\n";
      print {"PJK_FILE"} "\n";
   }

   close(ADJ_FILE);
   close(PJK_FILE);
  }
  #-------------------------------------------------------------
```

```
  # Modification 3 Ends
  #-------------------------------------------------------------
  return 0;
}

# check for private AS numbers in strings
sub ContainsPrivateAS {
  my @strings = @_;
  foreach my $string (@strings) {
    foreach my $number (split /\D+/, $string) {
      return 1 if ($number > 64511);
    }
  }
  return 0;
}

# print graph with MOASs and AS-sets sorted down
sub PrintGraph {
  my ($output_fh, $graph) = @_;
  foreach my $from (sort MixedSort keys %$graph) {
    foreach my $to (sort MixedSort keys %{$graph->{$from}}) {
      print $output_fh "$from\t$to\n";
    }
  }
  return 0;
}

sub MixedSort {
  return ($a cmp $b) if ( ($a =~ /\D/) and  ($b =~ /\D/) );
  return ($a <=> $b) if (!($a =~ /\D/) and !($b =~ /\D/) );
  return      1      if ( ($a =~ /\D/) and !($b =~ /\D/) );
  return      -1     if (!($a =~ /\D/) and  ($b =~ /\D/) );
}
```

B.10 Perl script: JOINMAT.pl

```
#!/usr/bin/perl -w
#-------------------------------------------------------------
# JOINMAT.pl (Join Matrix Information script)
#-------------------------------------------------------------
# This script builds the Adjacency Matrix from the raw
# data obtained from the RIPE Whois database. Before
# running this script you may want to download fresh data
# from this database by running the Unix script
# get_as_data.sh
#-------------------------------------------------------------
# Inputs:
# 1. List of ASs to be processed. This list should be
#    stored in text file "./AS_list". This file is also
#    used by Unix script get_as_data.sh.
# 2. Raw data files for each AS in the list. This files
#    must be stored in a "data" directory and named after
#    their corresponding AS followed by the extension
#    .data, e.g.:
#       ./data/AS123.data
#       ./data/AS4455.data
#    This is the same convention followed by the Unix
#    script get_as_data.sh.
# Outputs:
# 1. The Adjacency Matrix in the standard Unix prompt.
# 2. The Adjacency Matrix in the output file "AMG.out".
#    This output follows the Matlab notation in order to
#    make it easy to load this data into Matlab, e.g.:
#        > D = load('AMG.out')
# 3. Other messages, including warnings and errors in the
#    file "AMG.err".
#-------------------------------------------------------------
# Created by: David Arjona
# Date: 20/November/2007
#-------------------------------------------------------------



#=============================================================
#=============================================================
#=============================================================
# Main Program starts here
#=============================================================
#=============================================================
#=============================================================



#-------------------------------------------------------------
#-------------------------------------------------------------
# Initialize Section:
#-------------------------------------------------------------
#-------------------------------------------------------------

#-------------------------------------------------------------

#-------------------------------------------------------------
# File "JOINMAT.err" is the recipient of , status, warnings and
# error messages.
#-------------------------------------------------------------
open(MESSAGES, ">JOINMAT.err") or die "Can't create JOINMAT.err file: $!\n";

#-------------------------------------------------------------
# AS_list is the file that contains the list of ASs to be
# processed. We will store this list in the array "@ASlist"
# that has "$length" number of elements or ASs.
#-------------------------------------------------------------
open(AS_LIST_FILE, "AS_list") or die "Can't open AS_list file: $!\n";

while ($ASnum = <AS_LIST_FILE>) {
   chomp($ASnum);
   $length = push(@ASlist, $ASnum);
}

close(AS_LIST_FILE);

print {"MESSAGES"} "Length of the AS list: $length\n";
print "\nThis is an Adjacency Matrix of dimensions: $length by $length:\n\n";

#-------------------------------------------------------------
#-------------------------------------------------------------
# Initialize Section Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



#-------------------------------------------------------------
#-------------------------------------------------------------
# Main Loop:
#-------------------------------------------------------------
#-------------------------------------------------------------
# This loop processes each of the data files looking for ASs in
# "ASlist". It first needs to open the data file and then
# search for a connection between the current AS and the
# other ASs in the "ASlist".
#-------------------------------------------------------------
foreach $ASN (@ASlist) {
   $ASO = $ASN;
   $ASO =~ s/AS//;
   @Matrix_1 = ();
   @Matrix_2 = ();
   @Matrix_3 = ();

   #----------------------------------------------------------
   # The files we need to process are defined here.
   #----------------------------------------------------------
   $file1 =  "../ASdata/data/" . $ASN . ".out";
   $file2 = "../BGPdata/data/" . $ASO . ".out";
   $file3 =  "../SKdata/data/" . $ASO . ".out";


   #----------------------------------------------------------
   # Processing "file1":
   #----------------------------------------------------------
   print {"MESSAGES"} "\nProcessing $file1 :\n";
   print "Processing $file1 :\n";
   open(DATA_FILE, "$file1") or die "Can't open $file1: $!\n";

   while ($line = <DATA_FILE>) {
      #-------------------------------------------------------
      # Store each bool in "line" in the array "larray".
      #-------------------------------------------------------
      @larray = split(" ", $line);

      #-------------------------------------------------------
      # "@larray" is a row in the adjacency matrix which is now
      # being built.
      #-------------------------------------------------------
      push( @Matrix_1, [@larray] );
   }

   close(DATA_FILE);
   print {"MESSAGES"} "   Closing $file1.\n";
   #----------------------------------------------------------
   # Finished processing "file1".
   #----------------------------------------------------------

   #----------------------------------------------------------
   # Processing "file2":
   #----------------------------------------------------------
```

```
print {"MESSAGES"} "\nProcessing $file2 :\n";
print "Processing $file2 :\n";
open(DATA_FILE, "$file2") or die "Can't open $file2: $!\n";

while ($line = <DATA_FILE>) {
   #-------------------------------------------------------------
   # Store each bool in "line" in the array "larray".
   #-------------------------------------------------------------
   @larray = split(" ", $line);

   #-------------------------------------------------------------
   # "@larray" is a row in the adjacency matrix which is now
   # being built.
   #-------------------------------------------------------------
   push( @Matrix_2, [@larray] );
}

close(DATA_FILE);
print {"MESSAGES"} "   Closing $file2.\n";
#-------------------------------------------------------------
# Finished processing "file2".
#-------------------------------------------------------------

#-------------------------------------------------------------
# Processing "file3":
#-------------------------------------------------------------
print {"MESSAGES"} "\nProcessing $file3 :\n";
print "Processing $file3 :\n";
open(DATA_FILE, "$file3") or die "Can't open $file3: $!\n";

while ($line = <DATA_FILE>) {
   #-------------------------------------------------------------
   # Store each bool in "line" in the array "larray".
   #-------------------------------------------------------------
   @larray = split(" ", $line);

   #-------------------------------------------------------------
   # "@larray" is a row in the adjacency matrix which is now
   # being built.
   #-------------------------------------------------------------
   push( @Matrix_3, [@larray] );
}

close(DATA_FILE);
print {"MESSAGES"} "   Closing $file3.\n";
#-------------------------------------------------------------
# Finished processing "file3".
#-------------------------------------------------------------

#-------------------------------------------------------------
#-------------------------------------------------------------
# Join the information from the Adjacency Matrices by
# performing an OR operation.
#-------------------------------------------------------------
#*************************************************************
#-------------------------------------------------------------
for $x (0 .. ($length-1)) {
   for $y (0 .. ($length-1)) {
      $Matrix_1[$x][$y] = $Matrix_1[$x][$y] ||
                          $Matrix_2[$x][$y] ||
                          $Matrix_3[$x][$y];
   }
}
#-------------------------------------------------------------
#*************************************************************
#-------------------------------------------------------------



#-------------------------------------------------------------
# The adjacency information for each AS is stored in
# the files *.out and *.mat, which are stored in the "data"
# directory For example:
#   data/AS1299.out
#   data/AS1299.mat
#-------------------------------------------------------------
$ofile = "data/" . $ASN . ".out";
$pajek = "data/" . $ASN . ".mat";

open(ADJ_FILE, ">$ofile") or die "Can't open $ofile: $!\n";
open(PJK_FILE, ">$pajek") or die "Can't open $pajek: $!\n";

#-------------------------------------------------------------
# Add additional info for the Pajek .mat file:
#-------------------------------------------------------------
print {"PJK_FILE"} "*Vertices $length\n";

for ($i=1; $i<=$length; $i++) {
   print {"PJK_FILE"} " $i \"$ASlist[$i-1]\" 0.1 0.2 0.5\n";
}

print {"PJK_FILE"} "*Matrix\n";

#-------------------------------------------------------------
# Write the adjacency matrix into both files.
#-------------------------------------------------------------
for ($i=0; $i<$length; $i++) {
   for ($j=0; $j<$length; $j++) {
      print {"ADJ_FILE"} "$Matrix_1[$i][$j] ";
      print {"PJK_FILE"} "$Matrix_1[$i][$j] ";
   }
   print {"ADJ_FILE"} "\n";
   print {"PJK_FILE"} "\n";
}

close(ADJ_FILE);
close(PJK_FILE);

}
#-------------------------------------------------------------
#-------------------------------------------------------------
# Main Loop Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



close(MESSAGES);
print "\n *** JOINMAT.pl Finished! ***\n";

#=============================================================
#################################################################
#=============================================================
# Main Program finishes here
#=============================================================
#################################################################
#=============================================================
```

B.11 Perl script: ORMAT.pl

```
#!/usr/bin/perl -w
#-------------------------------------------------------------
# ORMAT.pl (OR All Matrices Information script)
#-------------------------------------------------------------
# This combines the Adjacency Matrices stored in the "data"
# directory by performing an OR operation among these matrices.
#-------------------------------------------------------------
# Inputs:
# 1. List of ASs to be processed. This list should be
#    stored in text file "./AS_list".
# 2. Matix files for each AS in the list. This files must be
#    stored in the "data" directory, named after their
#    corresponding AS and follow the conventions of a MATLAB
#    matrix file, e.g.:
#       ./data/AS123.out
#       ./data/AS4455.out
# Outputs:
# 1. The Combined Adjacency Matrix in MATLAB format: ORMAT.out
#    This output follows the Matlab notation in order to make
#    it easy to load this data into Matlab, e.g.:
#        > D = load('AMG.out')
# 2. The Combined Adjacency Matrix in Pajek format: ORMAT.mat
# 3. Other messages, including warnings and errors in the
#    file "ORMAT.err".
#-------------------------------------------------------------
# Created by: David Arjona
# Date: 16/January/2008
#-------------------------------------------------------------



#=============================================================
#################################################################
#=============================================================
# Main Program starts here
#=============================================================
#################################################################
#=============================================================



#-------------------------------------------------------------
#-------------------------------------------------------------
# Initialize Section:
#-------------------------------------------------------------
```

```
#-------------------------------------------------------------

#-------------------------------------------------------------
# File "ORMAT.err" is the recipient of , status, warnings and
# error messages.
#-------------------------------------------------------------
open(MESSAGES, ">ORMAT.err") or die "Can't create ORMAT.err file: $!\n";

#-------------------------------------------------------------
# AS_list is the file that contains the list of ASs to be
# processed. We will store this list in the array "@ASlist"
# that has "$length" number of elements or ASs.
#-------------------------------------------------------------
open(AS_LIST_FILE, "AS_list") or die "Can't open AS_list file: $!\n";

while ($ASnum = <AS_LIST_FILE>) {
   chomp($ASnum);
   $length = push(@ASlist, $ASnum);
}

close(AS_LIST_FILE);

print {"MESSAGES"} "Length of the AS list: $length\n";
print "\nThis is an Adjacency Matrix of dimensions: $length by $length:\n\n";

#-------------------------------------------------------------
# Initialize the Main Adjacency Matrix ($MAM) to zeros.
#-------------------------------------------------------------
for $x (0 .. ($length-1)) {
   for $y (0 .. ($length-1)) {
      $MAM[$x][$y] = 0;
   }
}

#-------------------------------------------------------------
#-------------------------------------------------------------
# Initialize Section Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



#-------------------------------------------------------------
#-------------------------------------------------------------
# Main Loop:
#-------------------------------------------------------------
#-------------------------------------------------------------
# This loop processes each of the data files looking for ASs in
# "ASlist". It first needs to open the data file and then
# search for a connection between the current AS and the
# other ASs in the "ASlist".
#-------------------------------------------------------------
foreach $ASN (@ASlist) {
   @Matrix_1 = ();

   #----------------------------------------------------------
   # The file we need to process is defined here.
   #----------------------------------------------------------
   $file1 =  "data/" . $ASN . ".out";

   #----------------------------------------------------------
   # Processing "file1":
   #----------------------------------------------------------
   print {"MESSAGES"} "\nProcessing $file1 :\n";
   print "Processing $file1 :\n";
   open(DATA_FILE, "$file1") or die "Can't open $file1: $!\n";

   while ($line = <DATA_FILE>) {
      #-------------------------------------------------------
      # Store each bool in "line" in the array "larray".
      #-------------------------------------------------------
      @larray = split(" ", $line);

      #-------------------------------------------------------
      # "@larray" is a row in the adjacency matrix which is now
      # being built.
      #-------------------------------------------------------
      push( @Matrix_1, [@larray] );
   }

   close(DATA_FILE);
   print {"MESSAGES"} "   Closing $file1.\n";
   #----------------------------------------------------------
   # Finished processing "file1".
   #----------------------------------------------------------

   #----------------------------------------------------------
   # Join the information from @Matrix_1 and @MAM by
   # performing an OR operation.
   #----------------------------------------------------------
   #**********************************************************
   #----------------------------------------------------------
   for $x (0 .. ($length-1)) {
      for $y (0 .. ($length-1)) {
         $MAM[$x][$y] = $MAM[$x][$y] || $Matrix_1[$x][$y];
      }
   }
   #----------------------------------------------------------
   #**********************************************************
   #----------------------------------------------------------
}
#-------------------------------------------------------------
#-------------------------------------------------------------
# Main Loop Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



#-------------------------------------------------------------
#-------------------------------------------------------------
# Further Processing Section:
#-------------------------------------------------------------
#-------------------------------------------------------------

#-------------------------------------------------------------
# The combined (OR) information is stored in two files:
#    ORMAT.out
#    ORMAT.mat
#-------------------------------------------------------------
$ofile = "ORMAT.out";
$pajek = "ORMAT.mat";

open(ADJ_FILE, ">$ofile") or die "Can't open $ofile: $!\n";
open(PJK_FILE, ">$pajek") or die "Can't open $pajek: $!\n";

#-------------------------------------------------------------
# Add additional info for the Pajek .mat file:
#-------------------------------------------------------------
print {"PJK_FILE"} "*Vertices $length\n";

for ($i=1; $i<=$length; $i++) {
   print {"PJK_FILE"} " $i \"$ASlist[$i-1]\" 0.1 0.2 0.5\n";
}

print {"PJK_FILE"} "*Matrix\n";

#-------------------------------------------------------------
# Write the Main Adjacency Matrix (MAM) into both files.
#-------------------------------------------------------------
for $x (0 .. ($length-1)) {
   for $y (0 .. ($length-1)) {
      print {"ADJ_FILE"} "$MAM[$x][$y] ";
      print {"PJK_FILE"} "$MAM[$x][$y] ";
   }
   print {"ADJ_FILE"} "\n";
   print {"PJK_FILE"} "\n";
}

close(ADJ_FILE);
close(PJK_FILE);

#-------------------------------------------------------------
#-------------------------------------------------------------
# Further Processing Section Ends
#-------------------------------------------------------------
#-------------------------------------------------------------



close(MESSAGES);
print "\n *** ORMAT.pl Finished! ***\n";

#=============================================================
#############################################################
#=============================================================
# Main Program finishes here
#=============================================================
#############################################################
#=============================================================
```

B.12 Perl script: WHOSWHO.pl

```
#!/usr/bin/perl -w
#-----------------------------------------------------------------
# WHOSWHO.pl
#-----------------------------------------------------------------
# This script builds the Adjacency Matrix from the raw
# data obtained from the RIPE Whois database. Before
# running this script you may want to download fresh data
# from this database by running the Unix script
# get_as_data.sh
#-----------------------------------------------------------------
# Inputs:
# 1. List of ASs to be processed. This list should be
#    stored in text file "./AS_list". This file is also
#    used by Unix script get_as_data.sh.
# 2. Raw data files for each AS in the list. This files
#    must be stored in a "data" directory and named after
#    their corresponding AS followed by the extension
#    .data, e.g.:
#       ./data/AS123.data
#       ./data/AS4455.data
#    This is the same convention followed by the Unix
#    script get_as_data.sh.
# Outputs:
# 1. The Adjacency Matrix in the standard Unix prompt.
# 2. The Adjacency Matrix in the output file "AMG.out".
#    This output follows the Matlab notation in order to
#    make it easy to load this data into Matlab, e.g.:
#        > D = load('AMG.out')
# 3. Other messages, including warnings and errors in the
#    file "AMG.err".
#-----------------------------------------------------------------
# Created by: David Arjona
# Date: 20/November/2007
#-----------------------------------------------------------------

#-----------------------------------------------------------------
# Classes are initialized here:
#    - Dest_Gp: This class stores and find the Fundamental
#         Groups from the Destination Groups that will be found
#         by subroutine Process_Destination_Groups().
#-----------------------------------------------------------------

#-----------------------------------------------------------------
# Subroutines' prototypes are declared here. The implementations
# come after the main program.
#-----------------------------------------------------------------



#=================================================================
#################################################################
#=================================================================
# Main Program starts here
#=================================================================
#################################################################
#=================================================================



#-----------------------------------------------------------------
#-----------------------------------------------------------------
# Initialize Section:
#-----------------------------------------------------------------
#-----------------------------------------------------------------

#-----------------------------------------------------------------
# File "AMG2.err" is the recipient of , status, warnings and
# error messages.
#-----------------------------------------------------------------
open(MESSAGES, ">WHOSWHO.err") or die "Can't create WHOSWHO.err file: $!\n";

#-----------------------------------------------------------------
# AS_list is the file that contains the list of ASs to be
# processed. We will store this list in the array "@ASlist"
# that has "$length" number of elements or ASs.
# NOTE: Because AS numbers are stored "just" as numbers in the
# data files, we need to delete the "AS" prefix from each AS
# stored in the file AS_list.
#-----------------------------------------------------------------
open(AS_LIST_FILE, "AS_list") or die "Can't open AS_list file: $!\n";

while ($ASnum = <AS_LIST_FILE>) {
   chomp($ASnum);
   $length = push(@ASlist, $ASnum);
}
close(AS_LIST_FILE);

print {"MESSAGES"} "Length of the AS list: $length\n";
print "\nThis is an Adjacency Matrix of dimensions: $length by $length:\n\n";

#-----------------------------------------------------------------
#-----------------------------------------------------------------
# Initialize Section Ends
#-----------------------------------------------------------------
#-----------------------------------------------------------------



#-----------------------------------------------------------------
#-----------------------------------------------------------------
# Main Loop:
#-----------------------------------------------------------------
#-----------------------------------------------------------------
# This loop processes each of the data files looking for ASs in
# "ASlist". It first needs to open the data file and then
# search for a connection between the current AS and the
# other ASs in the "ASlist".
#-----------------------------------------------------------------
foreach $ASN (@ASlist) {
   $ASO = $ASN;
   $ASO =~ s/AS//;
   @Matrix_1 = ();
   @Matrix_2 = ();
   @Matrix_3 = ();

   #--------------------------------------------------------------
   # The files we need to process are defined here.
   #--------------------------------------------------------------
   $file1 =  "../ASdata/data/" . $ASN . ".out";
   $file2 = "../BGPdata/data/" . $ASO . ".out";
   $file3 =  "../SKdata/data/" . $ASO . ".out";


   #--------------------------------------------------------------
   # Processing "file1":
   #--------------------------------------------------------------
   print {"MESSAGES"} "\nProcessing $file1 :\n";
   print "Processing $file1 :\n";
   open(DATA_FILE, "$file1") or die "Can't open $file1: $!\n";

   while ($line = <DATA_FILE>) {
      #-----------------------------------------------------------
      # Store each bool in "line" in the array "larray".
      #-----------------------------------------------------------
      @larray = split(" ", $line);

      #-----------------------------------------------------------
      # "@larray" is a row in the adjacency matrix which is now
      # being built.
      #-----------------------------------------------------------
      push( @Matrix_1, [@larray] );
   }

   close(DATA_FILE);
   print {"MESSAGES"} "   Closing $file1.\n";
   #--------------------------------------------------------------
   # Finished processing "file1".
   #--------------------------------------------------------------

   #--------------------------------------------------------------
   # Processing "file2":
   #--------------------------------------------------------------
   print {"MESSAGES"} "\nProcessing $file2 :\n";
   print "Processing $file2 :\n";
   open(DATA_FILE, "$file2") or die "Can't open $file2: $!\n";

   while ($line = <DATA_FILE>) {
      #-----------------------------------------------------------
      # Store each bool in "line" in the array "larray".
      #-----------------------------------------------------------
      @larray = split(" ", $line);

      #-----------------------------------------------------------
      # "@larray" is a row in the adjacency matrix which is now
      # being built.
      #-----------------------------------------------------------
      push( @Matrix_2, [@larray] );
   }

   close(DATA_FILE);
   print {"MESSAGES"} "   Closing $file2.\n";
   #--------------------------------------------------------------
```

```
   # Finished processing "file2".
   #-------------------------------------------------------------

   #-------------------------------------------------------------
   # Processing "file3":
   #-------------------------------------------------------------
   print {"MESSAGES"} "\nProcessing $file3 :\n";
   print "Processing $file3 :\n";
   open(DATA_FILE, "$file3") or die "Can't open $file3: $!\n";

   while ($line = <DATA_FILE>) {
      #----------------------------------------------------------
      # Store each bool in "line" in the array "larray".
      #----------------------------------------------------------
      @larray = split(" ", $line);

      #----------------------------------------------------------
      # "@larray" is a row in the adjacency matrix which is now
      # being built.
      #----------------------------------------------------------
      push( @Matrix_3, [@larray] );
   }

   close(DATA_FILE);
   print {"MESSAGES"} "   Closing $file3.\n";
   #-------------------------------------------------------------
   # Finished processing "file3".
   #-------------------------------------------------------------

   #-------------------------------------------------------------
   # Join the information from the Adjacency Matrices by
   # performing an OR operation.
   #-------------------------------------------------------------
   for $x (0 .. ($length-1)) {
      for $y (0 .. ($length-1)) {
         $Matrix_1[$x][$y] = (1 * $Matrix_1[$x][$y]) +
                             (2 * $Matrix_2[$x][$y]) +
                             (4 * $Matrix_3[$x][$y]);
      }
   }


   #-------------------------------------------------------------
   # The adjacency information for each AS is stored in
   # the files *.out and *.mat, which are stored in the "data"
   # directory For example:
   #   data/AS1299.out
   #   data/AS1299.mat
   #-------------------------------------------------------------
   $ofile = "data/" . $ASN . ".out";
   $pajek = "data/" . $ASN . ".mat";

   open(ADJ_FILE, ">$ofile") or die "Can't open $ofile: $!\n";
   open(PJK_FILE, ">$pajek") or die "Can't open $pajek: $!\n";

   #-------------------------------------------------------------
   # Add additional info for the Pajek .mat file:
   #-------------------------------------------------------------
   print {"PJK_FILE"} "*Vertices $length\n";

   for ($i=1; $i<=$length; $i++) {
      print {"PJK_FILE"} " $i \"$ASlist[$i-1]\" 0.1 0.2 0.5\n";
   }

   print {"PJK_FILE"} "*Matrix\n";

   #-------------------------------------------------------------
   # Write the adjacency matrix into both files.
   #-------------------------------------------------------------
   for ($i=0; $i<$length; $i++) {
      for ($j=0; $j<$length; $j++) {
         print {"ADJ_FILE"} "$Matrix_1[$i][$j] ";
         print {"PJK_FILE"} "$Matrix_1[$i][$j] ";
      }
      print {"ADJ_FILE"} "\n";
      print {"PJK_FILE"} "\n";
   }

   close(ADJ_FILE);
   close(PJK_FILE);

}
#----------------------------------------------------------------
#----------------------------------------------------------------
# Main Loop Ends
#----------------------------------------------------------------
#----------------------------------------------------------------
close(MESSAGES);
print "\n *** JOINMAT.pl Finished! ***\n";

#================================================================
################################################################
#================================================================
# Main Program finishes here
#================================================================
################################################################
#================================================================
```

B.13 Matlab script: AMA

```
function k = AMA(D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title: AMA (Adjacency Matrix Analyzer)
%---------------------------------------------------------------
%
%
%---------------------------------------------------------------
% Inputs:
% 1. The adjacency matrix of a directed multigraph (D). If
%    this matrix is not provided as an argument, this
%    program will automatically look for the output of the
%    AMG.pl Perl script (~/ASdata/AMG.out).
%---------------------------------------------------------------
% Outputs:
% 1. The arc-strong connectivity (k)
%---------------------------------------------------------------
% Created on: 12/Apr/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



%---------------------------------------------------------------
% Verify if D was provided as an argument, or use the
% default matrix from file ~/ASdata/AMG.out.
%---------------------------------------------------------------
if nargin == 0
    D = load('~/ASdata/AMG.out');
end


%---------------------------------------------------------------
% Eliminate stub nodes from the matrix:
% A) A row of all zeros.
% B) A column of all zeros.
%---------------------------------------------------------------
[n,v] = size(D);
X = any(D,2);
disp('  The following rows (in zero) contain only zeros:');
disp(X);

for m = v:-1:1
    if ~X(m)
        D(m,:) = [];
        D(:,m) = [];
    end
end


[n,v] = size(D);
X = any(D);
disp('  The following columns (in zero) contain only zeros:');
disp(X);

for m = v:-1:1
    if ~X(m)
       D(m,:) = [];
       D(:,m) = [];
    end
end


%---------------------------------------------------------------
% Find the strong components of the digraph. A strong
% component is defined as a block where all the vertices
% can be reached from each other. A strong digraph has one
% strong component by definition.
%---------------------------------------------------------------
Y = strong_components(D);
%[max_value, max_node] = max(Y);
max_value = max(Y);
disp('  These are the strong components of the Digraph:');
```

```
disp(Y);


if (max_value > 1)
    k = zeros(1,max_value);

    %-------------------------------------------------------
    % Find the arc-strong strong connectivity for each
    % block. The variable "m" is used for each block.
    %-------------------------------------------------------
    for m = 1:max_value

        [n,v] = size(D);
        X = D;

        %-------------------------------------------------------
        % Build the reduced matrix for each block. The
        % variable "n" is used to check each of the
        % elements of array "Y" and to delete rows from
        % "X".
        %-------------------------------------------------------
        for n = v:-1:1
            if Y(n) ~= m
                X(n,:) = [];
                X(:,n) = [];
            end
        end

        disp(' Calculating arc-strong connectivity for:');
        disp(X);

        k(m) = Fnd_Arc_Strg_Conn(X);
    end
else
    %-------------------------------------------------------
    % There is only one strong block, find its arc-strong
    % connectivity.
    %-------------------------------------------------------
    k = Fnd_Arc_Strg_Conn(D);
end
```

B.14 Matlab script: strong_components

```
function id = strong_components(D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title: Strong components of a digraph (Tarjan's
%        algorithm)
%-------------------------------------------------------------
% This program computes the strong components of a digraph.
% This algorithm is heavily based on the Tarjan's algorithm
% example (Program 19.11) which is provided in Section 19.8
% of Algorithms in Java Part 5 (Graph Algorithms) by Robert
% Sedgewick.
%-------------------------------------------------------------
% Inputs:
% 1. The adjacency matrix of a directed multigraph (D).
%-------------------------------------------------------------
% Outputs:
% 1. The strong components of D in an array. Each element
%    of this array corresponds to the nodes in the
%    adjacency matrix, and its value corresponds to the
%    block to which the vertex belongs.
%-------------------------------------------------------------
% Created on: 16/Apr/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%      0 5 1 3 2 4 6   8  10  12
% D = [0 1 1 0 0 0 1 0 0 0 0 0 0;
% 5    0 0 0 0 0 1 0 0 0 0 0 0 0;
% 1    0 0 0 0 0 0 0 0 0 0 0 0 0;
% 3    0 1 0 0 1 0 0 0 0 0 0 0 0;
% 2    1 0 0 1 0 0 0 0 0 0 0 0 0;
% 4    0 0 0 1 1 0 0 0 0 0 0 1 0;
% 6    0 0 0 0 0 1 0 0 0 1 0 0 0;
%      0 0 0 0 0 0 1 0 1 0 0 0 0;
% 8    0 0 0 0 0 0 0 1 0 1 0 0 0;
%      0 0 0 0 0 0 0 0 0 0 1 1 0;
% 10   0 0 0 0 0 0 0 0 0 0 0 0 1;
%      0 0 0 0 0 0 0 0 0 0 0 0 1;
% 12   0 0 0 0 0 0 0 0 0 1 0 0 0]

%-------------------------------------------------------------
% 1st Step: Find the dimensions of D:
%-------------------------------------------------------------
%   v = number of vertices
%   a = number of arcs
%-------------------------------------------------------------
[n,v] = size(D);
%a = sum(sum(D))

%-------------------------------------------------------------
% 2nd Step: Initialize all the data structures and
% variables:
% id  : This data structure will store the strong component
%       ID to which each vertex from D belongs to.
% low : This data structure stores the highest back link
%       (earliest visit) from the current vertex.
% pre : This data structure stores the order in which each
%       vertex is visited (processed).
% cnt : Counter, initialized to zero.
% snt : Block counter, initialized to zero.
% whilebool : Boolean used to execute the while loop at
%       least once.
%-------------------------------------------------------------
id  = zeros(v,1) - 1;
pre = id;
low = id;
S   = [];
cnt = 1;
snt = 1;
whilebool = 1;

%-------------------------------------------------------------
% 3rd Step: Tarjan's algorithm.
%-------------------------------------------------------------
for x = 1:v
   if (pre(x) == -1)
      SCRec(x);
   end
end
```

```
function SCRec(w)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title: Strong Components Recursive function
%-------------------------------------------------------------
% This is the recursive part of Tarjan's algorithm.
% (Program 19.11, Algorithms in Java by R. Sedgewick).
%-------------------------------------------------------------
% Inputs:
% 1. The starting vertex (w)
%-------------------------------------------------------------
% Outputs:
% 1. None
%-------------------------------------------------------------
% Created on: 16/Apr/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------------
% Variables are initialized and the counter is
% incremented.
%-------------------------------------------------------------
min    = cnt;
pre(w) = cnt;
low(w) = cnt;
cnt    = cnt+1;

%-------------------------------------------------------------
% Push the sarting vertex into the stack.
%-------------------------------------------------------------
push(w);

%-------------------------------------------------------------
% Look into each of the adjacent vertices to w, and if
% necessary, call "SCRec" again (recursive call).
%-------------------------------------------------------------
for t = find(D(w,:))
    if (pre(t) == -1)
        SCRec(t);
    end
    %---------------------------------------------------------
    % We changed the original program so we also check
    % that node t has not been allocated a block before
    % changing the value of "min".
    %---------------------------------------------------------
    if (low(t) < min) && (id(t) == -1)
        min = low(t);
    end
end

%-------------------------------------------------------------
```

```
    % If we need to adjust the value of "low(w)", it means
    % that we need to jump back to the previous vertex.
    %-----------------------------------------------------
    if (min < low(w))
        low(w) = min;
        return;
    end

    %-----------------------------------------------------
    % Here we build the strong blocks as we pop vertices
    % from the stack. Notice that because "whilebool" has
    % been initialized to 1, we will always execute this
    % loop at least once.
    %-----------------------------------------------------
    while (whilebool)
        t = pop;
        id(t)  = snt;
        if (t == w)
            whilebool = 0;
        end
    end

    %-----------------------------------------------------
    % Return "whilebool" to its original true value. And
    % increment the block counter.
    %-----------------------------------------------------
    whilebool = 1;
    snt = snt+1;

    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Nested function "SCRec" ends
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




    function push(val)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Title: Pushes value into the stack
    %-----------------------------------------------------
    % This function pushes a value into a stack.
    %-----------------------------------------------------
    % Inputs:
    % 1. The value to be pushed (val)
    %-----------------------------------------------------
    % Outputs:
    % 1. None
    %-----------------------------------------------------
    % Created on: 16/Apr/2007
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    z = length(S);
    S(z+1) = val;
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Nested function "push" ends
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




    function k = pop
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Title: Pops value from a stack
    %-----------------------------------------------------
    % This function pops a value from a stack: The value
    % is returned and eliminated at the same time.
    %-----------------------------------------------------
    % Inputs:
    % 1. None.
    %-----------------------------------------------------
    % Outputs:
    % 1. The popped value.
    %-----------------------------------------------------
    % Created on: 16/Apr/2007
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    z     = length(S);
    k     = S(z);
    S(z) = [];
    end
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Nested function "pop" ends
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

end
```

B.15 Matlab script: Fnd_Arc_Strg_Conn

```
function k = Fnd_Arc_Strg_Conn(D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title: Find Arc-strong Connectivity
%-------------------------------------------------------------
% This program finds the arc-strong connectivity of a
% directed multigraph (parallel edges are allowed). This
% method is based on proposition 7.4.1 (Digraphs By Bang-
% Jensen and Gutin).
%-------------------------------------------------------------
% Inputs:
% 1. The adjacency matrix of a directed multigraph (D)
%-------------------------------------------------------------
% Outputs:
% 1. The arc-strong connectivity (k)
%-------------------------------------------------------------
% Created on: 6/Mar/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%D = [0 0 1 1 0; 1 0 0 0 0; 0 0 0 0 1; 0 0 1 0 0; 0 1 1 0 0]
%D = [0 1 4 0 0 0;
%     0 0 0 1 3 0;
%     0 0 0 3 1 0;
%     0 1 0 0 0 1;
%     0 0 1 0 0 4;
%     3 0 0 0 0 0]

%-------------------------------------------------------------
% 1st Step: Find the dimensions of D:
%   v = number of vertices
%-------------------------------------------------------------
[v,n] = size(D);

%-------------------------------------------------------------
% 2nd Step: Find the maximum (s,t)-flow between vertices
% 1 and 2, 2 and 3,..., n-1 and n, n and 1.
%-------------------------------------------------------------
loop      = (1:n);
loop(n+1) = 1;
st_flow   = zeros(1,n);

for i = loop(1:n)
    %---------------------------------------------------------
    % Since the value of i and loop(i) is the same (except
    % for n+1) we use the short name.
    %---------------------------------------------------------
    st_flow(i) = preflow_push(i, loop(i+1), D);
end

%-------------------------------------------------------------
% 3rd Step: The arc-strong connectivity is the minimum
% (s,t)-flow obtained from the 2nd step.
%-------------------------------------------------------------
k = min(st_flow);
```

B.16 Matlab script: preflow_push

```
function max_flow = preflow_push(s, t, D)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title: Preflow-Push Algorithm
%-------------------------------------------------------------
% This function finds the maximum (s,t)-flow between two
% specified vertices following the preflow-push algorithm
% by Goldberg and Tarjan (section 3.6.3, "Digraphs" by
% Bang-Jensen and Gutin; section 3.4, "Combinatorial
% Optimization" by Cook, Cunningham, Pulleyblank and
% Schrijver); section 22.3, "Algorithms in Java" by Robert
% Sedgewick.
%-------------------------------------------------------------
% Inputs:
% 1. The starting vertex (s)
% 2. The terminating vertex (t)
% 3. The adjacency matrix of a directed multigraph (D)
%-------------------------------------------------------------
% Outputs:
% 1. The maximum (s,t)-flow between s and t (max_flow)
%-------------------------------------------------------------
% Created on: 7/Mar/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------------
% 1st Step: Find the dimensions of D:
%-------------------------------------------------------------
%   v = number of vertices
%   a = number of arcs
```

```
%-------------------------------------------------------------
[n,v] = size(D);
%a = sum(sum(D))


%-------------------------------------------------------------
% 2nd Step: Initialize the data structures.
%-------------------------------------------------------------
% D = Adjacency matrix
%     The s column needs to be set to zero in order to
%     avoid cycles when calculating the flow from s to t.
%-------------------------------------------------------------
D(:,s) = 0;

%-------------------------------------------------------------
% X = flow matrix
%     This is the flow that will be pushed through the
%     network. Initially, the algorithm tries to push as
%     much as possible from vertex s.
%-------------------------------------------------------------
X  = zeros(v,v);
X(s,:) = D(s,:);

%-------------------------------------------------------------
% bx = balance vector
%      Represents the amount of flow leaving an arc. A
%      negative value means flow entering an arc. Nodes
%      that have negative values are called ACTIVE.
%      Initially s will possess a maximum flow, which is
%      represented by a positive value. After the algorithm
%      finishes, s will posses the maximum final flow, t
%      will normally have the same value but negative and
%      the other nodes should store zeros (not active).
%      Note: bx(t) has to be initialized to zero because
%      if there is a direct link between s and t, we do not
%      want t to become an active node, we just need to
%      keep track of this link.
% active_nodes = stores the nodes that have negative flow
%      values (Active nodes). Because node t should be the
%      ONLY node with a negative flow value at the end
%      of this algorithm, we will need to zero this element
%      so it never becomes active.
%-------------------------------------------------------------
bx = sum(X')-sum(X);
bx(t) = 0;
active_nodes = find(bx < 0);

%-------------------------------------------------------------
% h = height function
%     Initialized to zero (0) with the exception of
%     h(s) = v (number of vertices). This initialization
%     differs from the one provided by the Digraphs book
%     but it is recomended by "Combinatorial Optimization"
%     Section 3.4.
%-------------------------------------------------------------
h  = zeros(1,v);
h(s) = v;


%-------------------------------------------------------------
% 3rd Step: Main loop.
%-------------------------------------------------------------
% Auxiliary variables:
% p = The current ACTIVE node.
% still_active = this variable reminds us that flow in the
%     active node (p) is still negative. We could just
%     check that the flow value is negative, but for future
%     flexibility we will keep this variable.
%-------------------------------------------------------------
while active_nodes
    %---------------------------------------------------------
    % The statement "p=active_nodes(1)" picks the next
    % active node in the order it was entered in the
    % adjacency matrix.
    % Maximum distance: The next active node is the one
    % with the maximum height function. This order of
    % selecting the next active node helps to reduce the
    % maximum number of computations to O(n^3) according
    % to Combinatorial Optimization's Theorem 3.34.
    %---------------------------------------------------------
    [max_value, max_node] = max(h(active_nodes));
    p = active_nodes(max_node);

    still_active = 1;

    %---------------------------------------------------------
    % Find an admissible arc pq and push as much flow as
    % possible through all its pq arcs. Remember that flow
    % has a negative value in bx.
    % If pq is already saturated there is no point on
    % trying to push on it.
    %---------------------------------------------------------
    for q = find(D(p,:))
       if (h(p) >= h(q)+1) && (D(p,q) ~=  X(p,q))
           still_active = push(p, q, bx(p));
       end
       %------------------------------------------------------
       % If p is not active, there is no point on continue
       % the FOR loop.
       %------------------------------------------------------
       if ~still_active
           break;
       end
    end

    %---------------------------------------------------------
    % If after pushing through all the pq arcs we are still
    % active, we may need to push back. We use a positive
    % (double negative) flow value to let the push function
    % know that we want to push back.
    % Note that we use the transpose of D when finding q,
    % this means that node q normally points to node p.
    %---------------------------------------------------------
    if still_active
       for q = find(D(:,p))'
           %------------------------------------------------
           % Be careful not to push back on nodes that are
           % bi-connected (both arcs, pq and qp, exist in
           % the original graph), this nodes have already
           % been processed by the 1st push. Hence, ONLY
           % arcs qp are admissible when pushing back.
           % We also need to check that arc qp is already
           % carrying flow or we will push back on a zero
           % flow.
           %------------------------------------------------
           if (h(p) >= h(q)+1) && (D(p,q) == 0) && (X(q,p) ~= 0)
               still_active = push(p, q, -bx(p));
           end
           %------------------------------------------------
           % If p is not active, there is no point on
           % continue the FOR loop.
           %------------------------------------------------
           if ~still_active
              break;
           end
       end
    end

    %---------------------------------------------------------
    % There are no more admissible arcs but the node is
    % still active (needs to carry flow), so we must lift.
    % This strategy is called the vertex-based preflow-
    % push algorithm according to "Algorithms in Java",
    % page 416.
    %---------------------------------------------------------
    if still_active
        lift(p);
    end

    %---------------------------------------------------------
    % Find the active nodes for the next iteration.
    %---------------------------------------------------------
    active_nodes = find(bx < 0);
end
%-------------------------------------------------------------
% Finish Main loop.
%-------------------------------------------------------------
%z = X
max_flow = sum(X(s,:));




    function active = push(a, b, flow)
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Title: Push Operation (nested function)
    %---------------------------------------------------------
    % This function pushes as much flow as possible between
    % nodes p and q. The maximum push allowed is the
    % minimum between the residual link pq and the
    % requested flow (Section 3.6.3, Digraphs By Bang-
    % Jensen and Gutin).
    %---------------------------------------------------------
```

```
% Inputs:
% 1. The starting vertex (a)
% 2. The terminating vertex (b)
% 3. The requested flow (flow)
%-------------------------------------------------------
% Outputs:
% 1. Active is true when the residual link could not
%    carry more flow (node p should still be active)
%    and false when all the requested flow could be
%    passed to node q (node q will become an active
%    node.
%-------------------------------------------------------
% Created on: 9/Mar/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------
% A negative flow value means that we need to push
% more flow forward. A positive flow value means that
% we must push back.
%-------------------------------------------------------
if flow < 0
    %---------------------------------------------------
    % If the flow in the opposite direction (b->a) is
    % smaller than the flow we are trying to push
    % (a->b), we just need to eliminate the previous
    % flow and push as much as possible.
    %---------------------------------------------------
    if X(b,a) < -flow
        X(b,a) = 0;
        %-----------------------------------------------
        % Find if the capacity of the pq link is enough
        % to carry ALL the flow.
        %-----------------------------------------------
        if (D(a,b) - X(a,b)) >= -flow
            %-------------------------------------------
            % Add all the flow and deactivate p.
            %-------------------------------------------
            X(a,b) = X(a,b) - flow;
            active = 0;
        else
            %-------------------------------------------
            % Add as much flow as possible.
            %-------------------------------------------
            X(a,b) = D(a,b);
            active = 1;
        end

    else % X(b,a) >= -flow
        %-----------------------------------------------
        % We do not need to add more flow in pq (a->b),
        % we just need to eliminate flow in the
        % opposite direction (b->a).
        %-----------------------------------------------
        X(b,a) = X(b,a) + flow;
        active = 0;
    end
else
    %---------------------------------------------------
    % This is a push-back operation (flow is positive).
    %---------------------------------------------------
    if X(b,a) < flow
        %-----------------------------------------------
        % The flow we need to push back is larger than
        % the capacity of the link, Eliminate all the
        % flow and continue eliminating.
        % This condition should rarely happen.
        %-----------------------------------------------
        X(b,a) = 0;
        active = 1;
    else % X(b,a) >= flow
        %-----------------------------------------------
        % Eliminate as much flow as possible.
        %-----------------------------------------------
        X(b,a) = X(b,a) - flow;
        active = 0;
    end
end

%-------------------------------------------------------
% Update the balance vector, do not forget that node t
% should have a negative flow value when this algorithm
% completes. So we need to modify it to be zero.
%-------------------------------------------------------
bx = sum(X') - sum(X);
bx(t) = 0;

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nested function "push" ends
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




function q = lift(p)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Title: Lift Operation (nested function)
%-------------------------------------------------------
% This function lifts the height of p and returns the
% node q that is now under p (Section 3.6.3, Digraphs
% By Bang-Jensen and Gutin).
%-------------------------------------------------------
% Inputs:
% 1. Vertex (p)
%-------------------------------------------------------
% Outputs:
% 1. Vertex that is now under p (q).
%-------------------------------------------------------
% Created on: 9/Mar/2007
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------------------------------------
% Temporarly make h(p)=0, so we do not consider the
% height of p when finding the minimum height node.
%-------------------------------------------------------
temp = h(p);
h(p) = 0;

%-------------------------------------------------------
% Find nodes which are higher than or at the same level
% as node p (temp). The result is stored in eval.
%-------------------------------------------------------
eval = find(h >= temp);

%-------------------------------------------------------
% For the nodes that are higher than p, find one with
% the smallest height, this is node q.
%-------------------------------------------------------
[min_height, q] = min(h(eval));

%-------------------------------------------------------
% Update the height function vector for node p, that
% is, lift p.
%-------------------------------------------------------
h(p) = min_height + 1;

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nested function "lift" ends
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

end
```

Appendix C

ChainRtg C++ Program

This is the C++ code of the ChainRtg program described in Chapter 6. This program's algorithm was described in section 6.2. This appendix includes the four modules needed to implement the ChainRtg program:

- BFSmod.cpp
- Chainclass.cpp
- Arcdbclass.cpp
- queue.cpp

This modules were compiled using Borland's C++ Builder Version 10, and tested in a Dell Inspiron 6000 laptop with a 2 GHz Intel Pentium processor, 2 GBytes of RAM and running the Microsoft Windows XP Home Edition operating system. This modules were also compiled in a Solaris Workstation by using the followin instruction:

```
CC -o ChainRtg.exe BFSmod.cpp
```

C.1 C++ module: BFSmod.cpp

```
#include <iostream>
#include <fstream>
#include "queue.cpp"
#include "Chainclass.cpp"
using namespace std;



////////////////////////////////////////////////////////////////////////////
//
//
//          MODIFIED BREADTH FIRST SEARCH (BFS) ALGORITHM
//
//
// This class and "main" program implement a modified version of the BFS
// algorithm. The modifications will be implemented (mostly) in a dependant
// module which will contain the functionality needed to implement the Chain
// Routing algorithm.
//
//--------------------------------------------------------------------------
//
// Created by: David Arjona Villicana
// Created on: 17/Oct/2008
// Last modified on:
//
//
////////////////////////////////////////////////////////////////////////////




////////////////////////////////////////////////////////////////////////////
//
//   CONSTANTS
//
////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////////
// MAX_NUM_VERTICES: Maximum number of vertices, in the adjacency
//    matrix, that this program supports.
//-------------------------------------------------------------------
const MAX_NUM_VERTICES = 100;

/////////////////////////////////////////////////////////////////////
// MAX_NUM_CHARS: Maximum number of characters supported for a
//    filename string.
//-------------------------------------------------------------------
const MAX_NUM_CHARS = 250;

/////////////////////////////////////////////////////////////////////
// ASN_CHARS: Maximum number of characters supported for an ASN name
//    string.
//-------------------------------------------------------------------
const ASN_CHARS = 8;

//#pragma hdrstop
//--------------------------------------------------------------------------




////////////////////////////////////////////////////////////////////////////
//
//   CLASS: digraph_class
//
////////////////////////////////////////////////////////////////////////////
//
// Private functions and variables:
//
// Public functions and variables:
//
////////////////////////////////////////////////////////////////////////////
class digraph_class : chain_class {

   int   v_size,   // The number of vertices (order) of the digraph/matrix
         a_size,   // The number of arcs (size) of the digraph
         ignored,  // The number of arcs pointing back to the source vertex
         src,      // Source vertex
         des;      // Destination vertex

   int   distance[MAX_NUM_VERTICES],    // Dist. between src and any vertex
         predecesor[MAX_NUM_VERTICES];  // Shortest pred. of any vertex to src

   int   *Adj_List[MAX_NUM_VERTICES],   // Adjacency list representation of D
         Idx_lngth[MAX_NUM_VERTICES];   // Length of each array of Adj_List

   char  *ASN_List[MAX_NUM_VERTICES];   // Stores names of each ASN

   bool process_Adj_line(char adj_line[ ]);
   bool process_ASN_line(char adj_line[ ], int line_num);

   void build_default_ASN_List();

public:
       digraph_class();
       ~digraph_class();

   void display_values();

   bool load_Adj_Matrix(char the_file[ ]);

   bool BFS(int s_vertex, int t_vertex);

};



////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::digraph_class
//-----------------------------------------------------------------
// Constructor for this class.
//-----------------------------------------------------------------
inline digraph_class::digraph_class() {
   cout << "   Executing constructor\n\n";

   v_size  = 0;
   a_size  = 0;
   ignored = 0;
}



////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::~digraph_class
//-----------------------------------------------------------------
// Destructor for this class.
//-----------------------------------------------------------------
inline digraph_class::~digraph_class() {
   int   i;

   cout << "   Executing destructor\n\n";

//   for (i=0; i<v_size; i++)
//   {
//      delete Adj_List[i];
//      delete ASN_List[i];
//   }
}



////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::display_all_values
//-----------------------------------------------------------------
// Displays relevant private variables of this class.
//-----------------------------------------------------------------
inline void digraph_class::display_values() {
   cout << "\n SUMMARY:\n";
   cout << "   Num. of vertices = " << v_size << "\n";
   cout << "   Num. of arcs     = " << a_size << "\n";
   cout << "   Ignored arcs     = " << ignored << "\n\n";
}



////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::process_Adj_line
//-----------------------------------------------------------------
//    This is the function that converts a string line of 1s and 0s,
//    into an entry in the Adj_List data structure.
//    In this function:
//       "x" represents the absolute position in the line.
//       "y" represents the row position in the matrix.
//       "j" keeps track of the last element in the "wrk_list" array.
//       "k" is only used to copy "wrk_list" into "Adj_list".
//    This is a summary of how this process works: In each line we
```

```
//    are mostly interested in the 1s, but we need to keep track of
//    the 0s because we need to know the position of the 1s (variable
//    "y" helps us to keep track). When we find a "1" we need to tell
//    the "wrk_list" in which position we have found it, and we also
//    need to keep track of how many "positions" have we stored in
//    "wrk_list" (using "j").
//    Once we have finished processing the line we must copy the
//    information stored in "wrk_list" to the last element (v_size)
//    of the Adj_List data structure.
//-------------------------------------------------------------------
// Input:
//    adj_line: The line (string) to be processed.
// Output:
//    Returns true if the conversion operation has been successful.
//-------------------------------------------------------------------
inline bool digraph_class::process_Adj_line(char adj_line[ ]) {
   int     x=0, y=0, j=0, k,
           wrk_list[MAX_NUM_VERTICES];

   // The following while loop processes the information in the
   // adj_line and converts it into an adjacency list which is stored
   // in wrk_list[].
   while (adj_line[x] != '\0') {
      switch (adj_line[x]) {
         case '0':
            y++;
            break;

         case '1':
            wrk_list[j] = y;
            y++;
            j++;
            break;

         case ' ':
            break;

         default:
            cerr << "ERROR: Invalid character at the Adjacency matrix in\n";
            cerr << "       row (" << v_size << ") and column (" << x << ")\n\n";
            return false;
      }
      x++;
   }

   // Let's add the adjacency data to the Adj_List 2-dimensional
   // array
   Idx_lngth[v_size] = j;
   Adj_List[v_size]  = new int[j];

   for (k=0; k<j; k++)
      Adj_List[v_size][k] = wrk_list[k];

   a_size += j;
   v_size++;

   return true;
}



//////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::process_ASN_line
//-------------------------------------------------------------------
//    This is the function that reads the ASN from a Pajek file and
//    stores it in the Adj_List structure.
//    In this function:
//       "x" represents the absolute position in the line.
//       "j" keeps track of the last element in the "wrk_list" array.
//       "k" is only used to copy "wrk_list" into "ASN_list".
//    This is a summary of how this process works: In each line we
//    are only interested in the characters stored between the double
//    quotes ("). The "on_off" switch helps us to store this chars in
//    the "wrk_list".
//    Once we have found the second (") we can stop processing the
//    line and we must copy the information stored in "wrk_list" to
//    the corresponding element (line_num) of the ASN_List data
//    structure.
//-------------------------------------------------------------------
// Input:
//    asn_line: The line (string) to be processed.
//    line_num: The position where this line should be inserted into
//       the ASN_List data structure.
// Output:
//    Returns true if the conversion operation has been successful.
//-------------------------------------------------------------------
inline bool digraph_class::process_ASN_line(char asn_line[ ], int line_num) {
   bool    on_off = false;
   int     x=0, j=0, k;
   char    wrk_list[ASN_CHARS];

   // The following while loop processes the information in the
   // asn_line
   while (asn_line[x] != '\0') {
      // The following if statement will only be true the 2nd time
      // this loop finds a double quote ("). We just break out of the
      // loop and copy the name we have got.
      if ((asn_line[x] == '"') && on_off) {
         break;
       }

      // The following if statement will obtain the ASN name that has
      // been place between the double quotes (").
      if (on_off) {
           wrk_list[j] = asn_line[x];
           j++;
      }

      // The following if statement will only be true the 1st time
      // this loop finds a double quote ("). We set on_off.
      if ((asn_line[x] == '"') && (!on_off)) {
         on_off = true;
      }

      x++;
   }
   wrk_list[j] = '\0';
   j++;

   // Let's add the adjacency data to the ASN_List character array.
    ASN_List[line_num]  = new char[j];

   for (k=0; k<j; k++)
      ASN_List[line_num][k] = wrk_list[k];

cout << " " << ASN_List[line_num] << "\n";
   return true;
}


//////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::build_default_ASN_list
//-------------------------------------------------------------------
// This function builds a default ASN_List of 3 letters using the
// following sequence: AAA, AAB, AAC, ... AAZ, ABA, ABB, ...
// NOTE: We must set the vertex size (v_size) before calling this
// function.
//-------------------------------------------------------------------
// Input:
//    None.
// Output:
//    None.
//////////////////////////////////////////////////////////////////////
inline void digraph_class::build_default_ASN_List() {
   int     j, k;
   char    name[ ]="AAA";

   // Loop through each one of the vertices in the adjacency matrix.
   for (j=0; j<v_size; j++) {
      // Copy the string in "name" to "ASN_List".
      ASN_List[j]  = new char[4];

      for (k=0; k<4; k++)
         ASN_List[j][k] = name[k];

      // Increment the value (letter) of "name".
      if ( (!(j % 25)) && j ) {
         name[1]++;
         name[2] = 'A';
      }
      else {
         name[2]++;
      }
   }
}


//////////////////////////////////////////////////////////////////////
//
//   Function: digraph_class::load_Adj_Matrix
//-------------------------------------------------------------------
// Loads an adjacency matrix from an external file into the  Adj_List
```

```
// data structure.
// Two kinds of input files are currently supported: Pajek files
// (.mat) and Matlab files (.out) that only contain the adjacency
// matrix.
// All the files are read through 4 different stages:
//    - initial: used to find the kind of file we are reading.
//    - AS_sorting: this stage is only used to read ASNs from Pajek
//      files (Matlab files do not contain any ASN names).
//    - Adj_matrix: used to read the adjacency matrix.
//    - last: used to process empty lines at the end of the file.
//-----------------------------------------------------------------
// Input:
//    the file[]: The name of the file that has to be processed.
// Output:
//    Returns true if the reading and loading operations have been
//    successful.
//-----------------------------------------------------------------
inline bool digraph_class::load_Adj_Matrix(char the_file[ ]) {
   int     sort_line = 0;
   bool    build_ASN_List = false;
   char    the_line[MAX_NUM_CHARS];
   enum    stages {initial, Adj_matrix, AS_sorting, last};
   stages  current_stage = initial;


   // Lets open the input file and check that it does exist.
   ifstream input_file(the_file);
   if (!input_file) {
      cerr << "ERROR: Input file (" << the_file << ") does not exist!\n\n";
      return false;
   }


   ////////////////////////////////////////////////////////////////
   // Main WHILE loop of this function.
   // Lets process the input file (line by line) using the stages
   // defined by "enum stages".
   while (input_file) {
      input_file.getline(the_line, MAX_NUM_CHARS);

      switch (current_stage) {
         case initial:
            // This stage will discover the kind of file received.
            // If this is a Pajek file we must jump to the
            // "AS_sorting" stage and look for the list of ASs that
            // need to be loaded into ASN_List 2-dimensional array.
            if ((the_line[0] == '*') && (the_line[1] == 'V')) {
               current_stage = AS_sorting;
               break;
            }

            // If this is an adjacency matrix file (Matlab format)
            // we must jump to the "Adj_matrix" stage.
            if ((the_line[0] == '0') || (the_line[0] == '1')) {
               current_stage = Adj_matrix;
               build_ASN_List = true;
            }
            // If this is an adjacency matrix file, we must start
            // processing it immediatelly. Hence we juts pass to the
            // "Adj_matrix" stage and do not need to use a "break"
            // here.

         case Adj_matrix:
            // If this is an empty line it means that we have reached
            // the end of the file, jump to the "last" stage in case
            // we have more empty lines.
            if (the_line[0] == '\0') {
               current_stage = last;
               break;
            }

            if ( !process_Adj_line(the_line) ) {
               cerr << "ERROR: Failed to process the Adjacency matrix\n\n";
               return false;
            }
            break;

         case AS_sorting:
            // When we have finished the "AS_sorting" stage, we must
            // process the adjacency matrix by jumping to the
            // "Adj_matrix" stage.
            if ((the_line[0] == '*') && (the_line[1] == 'M')) {
               current_stage = Adj_matrix;
               break;
            }

            if ( !process_ASN_line(the_line, sort_line) ) {
               cerr <<
               "ERROR: Failed to process the ASN list, but process continues\n\n";
               }
               else {
                  sort_line++;
               }
               break;

         case last:
            // In the "last" stage we do nothing, this stage is just
            // to ignore empty lines at the end of the file.
            cout << "last stage \n";
            break;

         default:
            cerr << "ERROR: The input file has reached an invalid stage\n\n";
            return false;
      }
   }
   // Main WHILE loop finishes.
   ////////////////////////////////////////////////////////////////


   // Matlab files do not have ASN name, so we need to create a
   // defalut list of ASN names here.
   // Note: We cannot do this before because we need to know the
   // vertex size (v_size).
   if (build_ASN_List) {
      build_default_ASN_List();
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
//   Function: net_abstract::BFS
//-----------------------------------------------------------------
// This function starts the Breadth First Search (BFS) algorithm.
//    It is almost identical to the standard function, but it only
//    calls the modified_DFS_PROC from the starting vertex.
//    For more info consult any algorithms' book or the Digraphs book
//    by Bang-Jensen and Gutin (Pg. 172, Section 4.1).
//-----------------------------------------------------------------
// Input:
//    a: The starting vertex (s_vertex).
//    b: The terminating vertex (t_vertex).
// Output:
//
////////////////////////////////////////////////////////////////////
inline bool digraph_class::BFS(int a, int b) {
   int   i,
         head;
   queue Q;

   src = a;
   des = b;

   // Initialize arrays: distance and predecesor.
   for (i=0; i<v_size; i++) {
      //prune_vrtx[i] = false;
      distance[i]   = -1;
      predecesor[i] = -1;
   }
   distance[src] = 0;

   // Initialize the queue
   Q.put_tail(src);

   head = Q.get_head();
   //cout << "The head: " << head << "\n";

   // While there are elements in the queue...
   while (head > -1) {
      // For each of the out-neighbours of the head...
      for (i=0; i<Idx_lngth[head]; i++) {

         // If the distance has not been set is because this vertex
         // has been discovered for the 1st time; if not, we have a
         // transitive relationship.
         if (distance[Adj_List[head][i]] == -1) {
            // The following steps are part of the regular BFS
            // algorithm: set the new distance and predecesor and add
            // the discovered node to the queue.
            distance[Adj_List[head][i]]   = distance[head] + 1;
            predecesor[Adj_List[head][i]] = head;
```

```
            Q.put_tail(Adj_List[head][i]);

            // These are the modifications to the BFS algorithm to
            // perform chain routing.
            //-------------------------------------------------------
            // If this arc is the only out-neighbour of the head and
            // the predecesor is not the origin, try to add this arc
            // as part of a Varc. If not, this is just a regular arc.
            if ((Idx_lngth[head] == 1) && (predecesor[head] != -1)) {
               if (!add_arc_to_Varc(head, Adj_List[head][i], predecesor[head])) {
                  cerr << "ERROR: Could not add Arc: " << head
                       << "-" << Adj_List[head][i] << " to Varc.\n\n";
               }
            }
            else {
               if (add_arc_to_Arc(head, Adj_List[head][i]) == -1)
                  cerr << "ERROR: Could not add Arc: " << head
                       << "-" << Adj_List[head][i] << " to Arc.\n\n";
            }
         }
         // This node has been discovered before, there is a
         // transitive relationship that needs to be processed.
         else {
            // Verify if the arc is not a cycle back to the
            // source vertex.
            if (predecesor[Adj_List[head][i]] == -1) {
               ignored++;
               cerr << "WARNING: Ignoring Arc going back to origin: "
                    << head << "-" << Adj_List[head][i] << "\n\n";
            }
            else {
               if (!add_arc_to_Chain(head, Adj_List[head][i], MAX_NUM_VERTICES,
                                     predecesor))
                  cerr << "ERROR: Could not add Arc: " << head
                       << "-" << Adj_List[head][i] << " to Chain.\n\n";
            }
         }

      }

      // Get the head of the queue for the next iteration.
      head = Q.get_head();
   }

   // If the number of vertices is less than 20 display the values on
   // the screen, otherwise use the default output file.
   if (v_size < 20)
      display_all_values(0);
   else
      display_all_values(1);

   return true;
}


//---------------------------------------------------------------------------
```

```
////////////////////////////////////////////////////////////////////////////
//
//   MAIN PROGRAM
//
////////////////////////////////////////////////////////////////////////////
//#pragma argsused
int main() //int argc, char* argv[]) {
   int  a, b;
   char filename[MAX_NUM_CHARS];

   digraph_class     ob_D;


   //ofstream outfile ("TESTING.TXT");
   //outfile << "Where is this file\n";

   do {
      cout << "Please provide the file to be processed:\n";
      cout << "> ";
      cin  >> filename;
   }
   while ( !ob_D.load_Adj_Matrix(filename) );

   ob_D.display_values();

   cout << "Please provide the originating ASN: ";
   cin  >> a;
   //cout << "Please provide the terminating ASN: ";
   //cin  >> b;
   b = 0;

   ob_D.BFS(a, b);
   ob_D.display_values();


   cin >> filename;

   return 1;
}
```

C.2 C++ module: Chainclass.cpp

```
#include <iostream>
#include <fstream>
#include "Arcdbclass.cpp"
using namespace std;



////////////////////////////////////////////////////////////////////////////
//
//
//          IMPLEMENTATION OF DATA STRUCTURE FOR CHAIN ROUTING
//
//
// This class implements the data structure and class needed to implement the
// Chain Routing algorithm.
//
//--------------------------------------------------------------------------
//
// Created by: David Arjona Villicana
// Created on: 20/Oct/2008
// Last modified on:
//
//
////////////////////////////////////////////////////////////////////////////




////////////////////////////////////////////////////////////////////////////
//
//   CONSTANTS
//
////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// ARRAY_BLK_1: Size of a block of data 1.
//------------------------------------------------------------------
const ARRAY_BLK_1 = 6;

//////////////////////////////////////////////////////////////////////
// ARRAY_BLK_2: Size of a block of data 2.
//------------------------------------------------------------------
const ARRAY_BLK_2 = 40;

//////////////////////////////////////////////////////////////////////
// ARRAY_BLK_3: Size of a block of data 3.
//------------------------------------------------------------------
const ARRAY_BLK_3 = 1;




////////////////////////////////////////////////////////////////////////////
//
//   TYPES
//
////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////////
// linktype: The 3 types of links we consider in this data structure.
//------------------------------------------------------------------
enum linktype {Arc, Varc, Chain, invalid};
```

```
////////////////////////////////////////////////////////////////////////////
//
//   STRUCTURES
//
////////////////////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////////
// segment: The attributes of each type of segment are stored here:
//    Arc: They only need tail and head (there is nothing in
//       between).
//    Varc: Since only the tail and head are provided, further info
//       is found at the next level. Use "idx" to find the
//       corresponding path at the next level.
//    Chain: Same as Varc.
//------------------------------------------------------------------
struct segment {
 //  linktype   type;
   int        tail,
              head,
              idx;
};


//////////////////////////////////////////////////////////////////
// path_tuple: The details of each single Path are provided. A Path
//    could be either: a Chain, a Virtual Arc or just a single Arc.
//    The "segment_ptr" is used to find the corresponding segment at
//    a previous level (it will always be nil at level 0). The
//    "Segment_Array" is used to describe each different segment of
//    the path.
//------------------------------------------------------------------
struct path_tuple {
   linktype   type;
   int        chlng,
              length,
              size,
              plidx;
   segment    *Segment_Array;
};


//////////////////////////////////////////////////////////////////
// Path_Level_Struct: This structure contains different Paths
//    between 2 nodes (tail and head). A Path could be either: a
//    Chain, a Virtual Arc or just a single Arc.
//    At level 0, the only tail is the source vertex and the only
//    head is the destination vertex. At other levels, different
//    tail and head nodes should exist.
//------------------------------------------------------------------
struct path_struct {
   int          size,
                Num_of_Paths;
   path_tuple   *Path_Array;
};




////////////////////////////////////////////////////////////////////////////
//
//   CLASS: chain_class
//
////////////////////////////////////////////////////////////////////////////
//
// Private functions and variables:
//    From: The originator AS.
//    To: The terminating AS, also called the destination AS.
//    Max_Level: The current maximum abstraction level. Initialized to 0 by
//       the constructor.
//    Path_Level: This is an array of structures that holds path info for
//       levels from 0 up to Max_Level.
//
//    incr_Path_Level_size(): Increases the size of Path_Level[].
//    incr_Segment_Array_size(): Increases the size of Path_Level[].
//       Path_Array[].Segment_Array[].
//    move_arc_to_next_level(): Moves an exsiting arc to the next level of
//       abstraction.
//
// Public functions and variables:
//    chain_class(): Constructor for this class.
//    ~chain_class(): Destructor for this class.
//    display_all_values(): Displays the values stored in this data
//       structure.
//    add_arc_to_Arc(): Adds an arc as just a single Arc structure.
//    add_arc_to_Varc(): Adds and arc as part of a Varc structure.
//    add_arc_to_Chain(): Adds ...
//
////////////////////////////////////////////////////////////////////////////
class chain_class : arc_db_class {

   int    Nested,
          To,
          Max_Level;

   path_struct   *Path_Level;

   bool incr_Path_Level_size(int level);
   bool decr_Path_Level_size(int level);
   bool incr_Segment_Array_size(int level, int index, int size);
   bool decr_Segment_Array_size(int level, int index);

   bool delete_1_element_from_DS(int a, int b, int c, int d, linktype e);
   bool remove_elements_from_DS(int level, int index);
   int  copy_Varc_or_Chain_to_DS(int lvl, int idx, int New_lvl, int plidx);
   int  move_arc_to_next_level(int a, int b, int c, int d);
   int  move_arc_to_prev_level(int a, int b, int c, int d);

//   bool disassemble_Varc(int level, int index, int a, int b);
//   int  get_segment_from_Varc_provide_Arc_info
//        (int level, int index, int a, int b);
   int  duplicate_Varc_segment
        (int lvl, int idx, int a, int b, int New_lvl);
   int  get_segment_from_Varc_provide_Varc_info
        (int level, int index, int a, int b, int New_level);

   int  create_path_Varc
        (int level, int size, int *vtx_ary, linktype *sgmt_ary);
   int  find_last_index_at_specific_level(int level, int a, int b);
   int  *find_arc_in_data_structure(int a, int b, linktype c);
   int  *find_src_vertex(int level, int index, int a);
   linktype convert_int_to_linktype(int a);

   int  create_simple_3point_Chain
        (int level, int idx_1, int idx_2, int idx_3,
         int tail, int headA, int headB, int plidx);
   int  create_Chain_X_incld_Y
        (int Lvl, int Aidx, int Bidx, int a, int b, bool data);
   int  create_Chain_X_joins_Y
        (int Lvl, int Aidx, int Bidx, int a, int b, bool data);
   bool sanitize_Arc_Vs_Chain(int a, int b, int lvl, int idx);
   bool sanitize_Chain_Vs_Chain(int Alvl, int Aidx, int Blvl, int Bidx);
   bool remove_Chain(int level, int index);
   bool join_Chains
        (int Alvl, int Aidx, int Blvl, int Bidx, int a, int b, bool data);
   bool create_series_of_Chains
        (int Blvl, int Bidx,
         int last_A, int *array_A, int last_B, int *array_B);
   bool process_Chain_with_Varcs
        (int A, int B, int Bbs_lvl, int Bbs_idx,
         int segm_A, int prev_segm_A);
   bool process_invalid_src_vertex
        (int A, int Blvl, int Bidx, int size, int *prev);
   int  *find_alternative_B_path
        (int A, int tA, int B, int tB);

public:
        chain_class();
        ~chain_class();

   void display_all_values(bool print);

   int  add_arc_to_Arc(int a, int b, int c, int d);
   bool add_arc_to_Varc(int a, int b, int c);
   bool add_arc_to_Chain(int A, int B, int c, int *prev);

};


//////////////////////////////////////////////////////////////////
//
// Function: chain_class::chain_class
//------------------------------------------------------------------
// The constructor for this class.
//////////////////////////////////////////////////////////////////
inline chain_class::chain_class() {
   cout << "   Executing constructor for: chain_class\n\n";

   Nested = 0;
```

```
   To     = 0;

   // Initialize Path_Level data structure.
   //
   // NOTE: We still need to implement a procedure to increase the
   // number of Levels when we have used the ones defined by
   // "ARRAY_BLK_1".
   //
   Max_Level  = 0;
   Path_Level = new path_struct[ARRAY_BLK_1];
   if (!Path_Level)
      cerr << "ERROR: Could not allocate memory to create\n"
           << "       Path_Level\n\n";

   Path_Level[0].Num_of_Paths = 0;
   Path_Level[0].Path_Array   = new path_tuple[ARRAY_BLK_2];

   if (!Path_Level[0].Path_Array) {
      cerr << "ERROR: Could not allocate memory to create\n"
           << "       Path_Level[0].Path_Array\n\n";
      Path_Level[0].size = 0;
   }
   else {
      Path_Level[0].size = ARRAY_BLK_2;
   }
}


////////////////////////////////////////////////////////////////////////
//
// Function: chain_class::~chain_class
//--------------------------------------------------------------------
// The destructor for this class.
////////////////////////////////////////////////////////////////////////
inline chain_class::~chain_class() {
   int   i;

   cout << "   Executing destructor for: chain_class\n\n";

   for (i=0; i<=Max_Level; i++) {
      delete [] Path_Level[i].Path_Array;
      Path_Level[i].Path_Array = NULL;
   }

   delete [] Path_Level;
   Path_Level = NULL;
}


////////////////////////////////////////////////////////////////////////
//
// Function: chain_class::display_all_values
//--------------------------------------------------------------------
// Displays the contents of the chain_class data structure.
//--------------------------------------------------------------------
// Input:
//    None.
// Output:
//    None.
////////////////////////////////////////////////////////////////////////
inline void chain_class::display_all_values(bool print=false) {
   int i, j, k;

if (print) {
   ofstream fileout("ChainRtg.out", ios::out);

   if (!fileout) {
      cerr << "ERROR: Output file (" << fileout << ") could not be opened!\n\n";
      return;
   }

   fileout << " ************************************************************\n";
   fileout << " ***               CHAIN DATA STRUCTURE                   ***\n";
   fileout << " ************************************************************\n";

   for (i=0; i<=Max_Level; i++) {
     fileout << "\n   ------------------------------------------------------\n";
     fileout << "   Level: " << i << "\n";
     fileout << "   Number of Paths = " << Path_Level[i].Num_of_Paths << "\n";
     fileout << "   ------------------------------------------------------\n";

     for (j=0; j<Path_Level[i].Num_of_Paths; j++)
     {
       fileout << "   Path " << j << "\n";
       fileout << "   Type = "
            << Path_Level[i].Path_Array[j].type;
       fileout << "   Chain lgth = "
            << Path_Level[i].Path_Array[j].chlng;
       fileout << "   Segmt cnt = "
            << Path_Level[i].Path_Array[j].length;
       fileout << "   Prv lev idx = "
            << Path_Level[i].Path_Array[j].plidx << "\n";

       for (k=0; k<Path_Level[i].Path_Array[j].length; k++)
       {
         fileout << "      Segment " << k << "    Arc = "
           <<Path_Level[i].Path_Array[j].Segment_Array[k].tail << "-"
           <<Path_Level[i].Path_Array[j].Segment_Array[k].head
           << "    Next Lev Idx = "
           <<Path_Level[i].Path_Array[j].Segment_Array[k].idx  << "\n";
       }
     }
   }
   fileout << "\n";

   fileout.close();
}
else {
   cout << " ************************************************\n";
   cout << " ***            CHAIN DATA STRUCTURE              ***\n";
   cout << " ************************************************\n\n";

   for (i=0; i<=Max_Level; i++) {
     cout << "\n   ------------------------------------------------------\n";
     cout << "   Level: " << i << "\n";
     cout << "   Number of Paths = " << Path_Level[i].Num_of_Paths << "\n";
     cout << "   ------------------------------------------------------\n";

     for (j=0; j<Path_Level[i].Num_of_Paths; j++)
     {
       cout << "   Path " << j << "\n";
       cout << "   Type = "
            << Path_Level[i].Path_Array[j].type;
       cout << "   Chain lgth = "
            << Path_Level[i].Path_Array[j].chlng;
       cout << "   Segmt cnt = "
            << Path_Level[i].Path_Array[j].length;
       cout << "   Prv lev idx = "
            << Path_Level[i].Path_Array[j].plidx << "\n";

       for (k=0; k<Path_Level[i].Path_Array[j].length; k++)
       {
         cout << "      Segment " << k << "    Arc = "
           <<Path_Level[i].Path_Array[j].Segment_Array[k].tail << "-"
           <<Path_Level[i].Path_Array[j].Segment_Array[k].head
           << "    Next Lev Idx = "
           <<Path_Level[i].Path_Array[j].Segment_Array[k].idx  << "\n";
       }
     }
   }
   cout << "\n";
}

display_arc_db(print);

}


////////////////////////////////////////////////////////////////////////
//
// Function: chain_class::incr_Path_Level_size
//--------------------------------------------------------------------
// Increases the size of the Path_Level[ ] data structure.
//--------------------------------------------------------------------
// Input:
//    The Level.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////////
inline bool chain_class::incr_Path_Level_size(int level) {
   // Verify if we really need to increment the size of Path_Level.
   // We increment the size of the Path_Array in blocks of size
   // ARRAY_BLK_2 (originally 20).
   if (Path_Level[level].Num_of_Paths >= Path_Level[level].size) {
      int   i,
            k = Path_Level[level].Num_of_Paths/ARRAY_BLK_2;

      // Use a temporary array to store Path_Level before we increase
      // its size. Then we need to copy the values back.

      cout << "   Increasing size of Path_Level[" << level << "]\n";

      path_tuple  * temp_array = new path_tuple[k*ARRAY_BLK_2];
```

```
      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to create\n"
              << "       temp_array\n\n";
         return false;
      }

      temp_array                  = Path_Level[level].Path_Array;
      Path_Level[level].Path_Array = new path_tuple[(k+1)*ARRAY_BLK_2];

      if (!Path_Level[level].Path_Array) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       Path_Level[" << level << "].Path_Array\n\n";
         return false;
      }
      else {
         Path_Level[level].size = (k+1)*ARRAY_BLK_2;

         // Copy each element of temp_array into the new
         // Path_Level[ ].Path_Array
         for (i=0; i<Path_Level[level].Num_of_Paths; i++) {
            Path_Level[level].Path_Array[i] = temp_array[i];
         }
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::decr_Path_Level_size
//----------------------------------------------------------------
// Increases the size of the Path_Level[ ] data structure.
//----------------------------------------------------------------
// Input:
//    The Level.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::decr_Path_Level_size(int level) {
   // Verify if we really need to decrease the size of Path_Level.
   // We decrease the size of the Path_Array in blocks of size
   // ARRAY_BLK_2 (originally 20).
   // Notice that the condition less-than (<) causes that Num_of_Paths
   // decreases 1 more than the necessary to eliminate a complete
   // block of data.
   if ((Path_Level[level].Num_of_Paths + ARRAY_BLK_2) < Path_Level[level].size) {
      int   i,
            k = (Path_Level[level].Num_of_Paths/ARRAY_BLK_2) + 1;

      // Use a temporary array to store Path_Level before we increase
      // its size. Then we need to copy the values back.

      cout << "   Decreasing size of Path_Level[" << level << "]\n\n";

      path_tuple  * temp_array = new path_tuple[Path_Level[level].size];

      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to create\n"
              << "       temp_array\n\n";
         return false;
      }

      temp_array                  = Path_Level[level].Path_Array;
      Path_Level[level].Path_Array = new path_tuple[k*ARRAY_BLK_2];

      if (!Path_Level[level].Path_Array) {
         cerr << "ERROR: Could not allocate memory to decrease size of\n"
              << "       Path_Level[" << level << "].Path_Array\n\n";
         return false;
      }
      else {
         Path_Level[level].size = k*ARRAY_BLK_2;

         // Copy each element of temp_array into the new
         // Path_Level[ ].Path_Array
         for (i=0; i<Path_Level[level].Num_of_Paths; i++) {
            Path_Level[level].Path_Array[i] = temp_array[i];
         }
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::incr_Segment_Array_size
//----------------------------------------------------------------
// Increases the size of the Path_Level[].Path_Array[].Segment_Array
// data structure by ARRAY_BLK_3.
// There are 2 points that are necessary to remember when using this
// function:
// 1. The size of the Segment_Array needs to be correct before
//    calling this function. Hence, new Segment_Array arrays need to
//    have a pre-definded size of zero.
// 2. The length of Segment_Array is ignored by this function,
//    because it is better to mantain an independence between length
//    and size, specially if we want to have a pre-defined length.
//----------------------------------------------------------------
// Input:
//    The Level.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::incr_Segment_Array_size(int level,
                                                 int index,
                                                 int size) {
   int   i,
         k = Path_Level[level].Path_Array[index].size/ARRAY_BLK_3,
         l = size/ARRAY_BLK_3;


   // If the current size is zero, this means that we need to create
   // a new Segment_Array.
   if (Path_Level[level].Path_Array[index].size <= 0) {
      Path_Level[level].Path_Array[index].Segment_Array =
         new segment[l*ARRAY_BLK_3];

      if (!Path_Level[level].Path_Array[index].Segment_Array) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       Path_Level[" << level << "].Path_Array["
              << index << "].Segment_Array\n\n";
         return false;
      }
      else {
         Path_Level[level].Path_Array[index].size = l*ARRAY_BLK_3;
         return true;
      }
   }
   // Verify if we really need to increment the size of Segment_Array.
   // We increment the size of the Segment_Array in blocks of size
   // ARRAY_BLK_3 (originally 1).
   else if (size > Path_Level[level].Path_Array[index].size) {
      // Use a temporary array to store Path_Level before we increase
      // its size. Then we need to copy the values back.

      //cout << "   Increasing size of Path_Level[" << level
      //     << "].Path_Array[" << index << "].Segment_Array\n";

      segment  * temp_array = new segment[k*ARRAY_BLK_3];

      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to create\n"
              << "       temp_array\n\n";
         return false;
      }

      temp_array = Path_Level[level].Path_Array[index].Segment_Array;

      Path_Level[level].Path_Array[index].Segment_Array =
         new segment[l*ARRAY_BLK_3];

      if (!Path_Level[level].Path_Array[index].Segment_Array) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       Path_Level[" << level << "].Path_Array["
              << index << "].Segment_Array\n\n";
         return false;
      }
      else {
         // Copy each element of temp_array into the new
         // Path_Level[ ].Path_Array
         if (temp_array) {
            for (i=0; i<Path_Level[level].Path_Array[index].size; i++) {
               Path_Level[level].Path_Array[index].Segment_Array[i] =
                  temp_array[i];
```

```
            }
         }

         Path_Level[level].Path_Array[index].size = 1*ARRAY_BLK_3;
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::decr_Segment_Array_size
//--------------------------------------------------------------------
// Increases the size of the Path_Level[ ] data structure.
//--------------------------------------------------------------------
// Input:
//    The Level.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::decr_Segment_Array_size(int level,
                                                 int index) {
   // Verify if we really need to decrement the size of Segment_Array.
   // We decrement the size of the Segment_Array in blocks of size
   // ARRAY_BLK_3 (originally 1).
   //if ((Path_Level[level].Path_Array[index].length + ARRAY_BLK_3) <
   //       Path_Level[level].Path_Array[index].size)
   if (Path_Level[level].Path_Array[index].length <
       Path_Level[level].Path_Array[index].size) {
      int   i,
            k = Path_Level[level].Path_Array[index].length/ARRAY_BLK_3;

      // Use a temporary array to store Path_Level before we decrease
      // its size. Then we need to copy the values back.

      //cout << "   Decreasing size of Path_Level[" << level
      //     << "].Path_Array[" << index << "].Segment_Array\n\n";

      segment  * temp_array =
                    new segment[Path_Level[level].Path_Array[index].size];

      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to create\n"
              << "       temp_array\n\n";
         return false;
      }

      temp_array = Path_Level[level].Path_Array[index].Segment_Array;
      Path_Level[level].Path_Array[index].Segment_Array =
         new segment[k*ARRAY_BLK_3];

      if (!Path_Level[level].Path_Array[index].Segment_Array) {
         cerr << "ERROR: Could not allocate memory to decrease size of\n"
              << "       Path_Level[" << level << "].Path_Array["
              << index << "].Segment_Array\n\n";
         return false;
      }
      else {
         Path_Level[level].Path_Array[index].size = k*ARRAY_BLK_3;

         // Copy each element of temp_array into the new
         // Path_Level[ ].Path_Array
         for (i=0; i<Path_Level[level].Path_Array[index].length; i++) {
            Path_Level[level].Path_Array[index].Segment_Array[i] = temp_array[i];
         }
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::delete_1_element_from_DS
//--------------------------------------------------------------------
// Deletes just 1 Arc/Varc/Chain from the Path_Level[].Path_Array[]
// data structure.
// NOTE: If this element is a Varc or Chain and we also need to
// remove the segments that form it, it is necessary to instead use
// procedure "remove_elements_from_DS".
//--------------------------------------------------------------------
// Input:
//    a) The tail of the Arc/Varc/Chain.
//    b) The head of the Arc/Varc/Chain.
//    c) The level from which the element will be deleted.
//    d) The index from which the element will be deleted.
//    e) The type of element we are deleting (the default is an
//       individual Arc, but Varcs and Chains are acceptable).
// Output:
//    Returns TRUE if the operation was succesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::delete_1_element_from_DS
   (int a, int b, int c, int d, linktype e=Arc) {
   int   i, j, nidx, pidx;


   // Delete the Arc location from the Arc database only if we are
   // deleting an individual Arc.
   if (e == Arc)
      if (!del_arc_location(a, b, c, d))
         return false;

   // Now delete the Arc at the current level by shifting all the
   // last elements of Path_Level.
   for (i=d; i<(Path_Level[c].Num_of_Paths-1); i++) {
      Path_Level[c].Path_Array[i].type   = Path_Level[c].Path_Array[i+1].type;
      Path_Level[c].Path_Array[i].chlng  = Path_Level[c].Path_Array[i+1].chlng;
      Path_Level[c].Path_Array[i].length = Path_Level[c].Path_Array[i+1].length;
      Path_Level[c].Path_Array[i].plidx  = Path_Level[c].Path_Array[i+1].plidx;

      // Adjust the size of Segment_Array if necessary.
      if (Path_Level[c].Path_Array[i].size <
          Path_Level[c].Path_Array[i+1].size) {
         if(!incr_Segment_Array_size(c, i, Path_Level[c].Path_Array[i+1].size))
            return false;
      }
      else if (Path_Level[c].Path_Array[i].size >
               Path_Level[c].Path_Array[i+1].size) {
         if(!decr_Segment_Array_size(c, i))
            return false;
      }

      // Copy each element of Segment_Array.
      for (j=0; j<Path_Level[c].Path_Array[i].length; j++) {
         Path_Level[c].Path_Array[i].Segment_Array[j] =
            Path_Level[c].Path_Array[i+1].Segment_Array[j];
//         Path_Level[c].Path_Array[i].Segment_Array[j].head =
//            Path_Level[c].Path_Array[i+1].Segment_Array[j].head;
//         Path_Level[c].Path_Array[i].Segment_Array[j].idx  =
//            Path_Level[c].Path_Array[i+1].Segment_Array[j].idx;
      }


      // Next Level Operations:
      if (Path_Level[c].Path_Array[i].type == Arc) {
         // If this is an individual Arc, move its location at the
         // Arc Database from i+1 to i.
         mod_arc_location(Path_Level[c].Path_Array[i].Segment_Array[0].tail,
                          Path_Level[c].Path_Array[i].Segment_Array[0].head,
                          c, i+1, c, i);
      }
      else {
         // This is NOT an individual Arc, move the indexes at the
         // next level of abstraction.
         for (j=0; j<Path_Level[c].Path_Array[i].length; j++) {
            nidx = Path_Level[c].Path_Array[i].Segment_Array[j].idx;

            Path_Level[c+1].Path_Array[nidx].plidx = i;
         }
      }


      // Previous Level Operations:
      // If the previous level (plidx) is not -1, move the
      //corresponding index at the previous level of abstraction.
      pidx = Path_Level[c].Path_Array[i].plidx;

      if (-1 != pidx) {
         for (j=0; j<Path_Level[c-1].Path_Array[pidx].length; j++) {
            if ((i+1) == Path_Level[c-1].Path_Array[pidx].Segment_Array[j].idx) {
               Path_Level[c-1].Path_Array[pidx].Segment_Array[j].idx = i;
               break;
            }
         }
      }
```

```
   }


   delete [] Path_Level[c].Path_Array[i].Segment_Array;
   Path_Level[c].Path_Array[i].Segment_Array = NULL ;

   Path_Level[c].Num_of_Paths--;

   // Decrease the number of Path_Level elements at this level.
   return decr_Path_Level_size(c);
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::remove_elements_from_DS
//----------------------------------------------------------------
// Deletes an Arc/Varc/Chain from the Path_Level[].Path_Array[] data
// structure and all the other segments that form such element.
//----------------------------------------------------------------
// Input:
//    a) The level from which the element will be deleted.
//    b) The index from which the element will be deleted.
// Output:
//    Returns TRUE if the operation was succesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::remove_elements_from_DS
   (int level, int index) {
   int   i, tmp,
         tail, head, last;
   linktype   type;


   // Find more information about the element we are about to remove.
   type = Path_Level[level].Path_Array[index].type;

   if (Arc == type)
      last = 0;
   else
      last = Path_Level[level].Path_Array[index].length-1;

   tail = Path_Level[level].Path_Array[index].Segment_Array[0].tail;
   head = Path_Level[level].Path_Array[index].Segment_Array[last].head;


   // Remove process starts here.
   switch (type) {
      case Varc:
      case Chain: {
         // Chains and Varcs have segments that need to be removed
         // before the current element.
         for (i=0; i<=last; i++) {
            tmp = Path_Level[level].Path_Array[index].Segment_Array[i].idx;

            // Verify that the segment has not been assigned to other
            // structure.
            if (index == Path_Level[level+1].Path_Array[tmp].plidx)
               if(!remove_elements_from_DS(level+1, tmp)) {
                  cerr << "ERROR: Could not succesfully remove an element\n"
                       << "       from the data structure.\n\n";
               }
         }
      }
      case Arc: {
         // Here we can remove the current element from the DS.
         return delete_1_element_from_DS(tail, head, level, index, type);
      }
      default: {
         cerr << "ERROR: Invalid element type at procedure:\n"
              << "       remove_elements_from_DS.\n\n";
         return false;
      }
   }
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::add_arc_to_Arc
//----------------------------------------------------------------
// Add a new arc into the Path_Level[].Path_Array[] data structure.
//
// WARNING: It is possible (and sometimes desirable) that more than
//    one instance of an Arc exist at the same level. This may happen
//    if an Arc forms part of more than one Chain/Varc.
//----------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The level to which the arc will be added (the default level
//       is 0).
//    d) The index of the Varc or Chain to which this arc belongs (at
//       the previous level and only if it does exist)
// Output:
//    Returns the index of Path_Level[c].Path_Array where the arc now
//    resides, or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
inline int chain_class::add_arc_to_Arc(int a, int b, int c=0, int d=-1) {
   int   idx = Path_Level[c].Num_of_Paths;


   // Increase the Max_Level if necessary.
   if (Max_Level < c) {
      Max_Level = c;
      Path_Level[c].Num_of_Paths = 0;
   }

   if (!incr_Path_Level_size(c))
      return -1;

   // Add new Arc to the Arc database or a new location if the arc
   // has already been created.
   if (find_arc_in_db(a, b) == -1) {
      if (!add_new_arc_to_db(a, b, c, idx))
         return -1;
   }
   else {
      if (!add_arc_location(a, b, c, idx))
         return -1;
   }

   // Let's add the new Arc.
   Path_Level[c].Path_Array[idx].type   = Arc;
   Path_Level[c].Path_Array[idx].chlng  = 0;
   Path_Level[c].Path_Array[idx].length = 1;
   Path_Level[c].Path_Array[idx].plidx  = d;

   Path_Level[c].Path_Array[idx].size   = 0;
   if (!incr_Segment_Array_size(c, idx, 1))
      return -1;

   Path_Level[c].Path_Array[idx].Segment_Array[0].tail = a;
   Path_Level[c].Path_Array[idx].Segment_Array[0].head = b;
   Path_Level[c].Path_Array[idx].Segment_Array[0].idx  = -1;
   Path_Level[c].Num_of_Paths++;

   return idx;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::copy_Varc_or_Chain_to_DS
//----------------------------------------------------------------
// Deletes the "remainder" parts of a Chain that has been used to
// form a Combined Chain.
// NOTE: This procedure verifies, before removing any arc, that the
// segments of an unused Chain are not being used by other Chains.
//----------------------------------------------------------------
// Input:
//    a) The level where the Chain is.
//    b) The index where the Chain is.
//    d) The level where we want the new copy to reside (the default
//       value is level zero).
// Output:
//    Returns TRUE if the operation was succesful.
////////////////////////////////////////////////////////////////////
inline int chain_class::copy_Varc_or_Chain_to_DS
   (int lvl, int idx, int New_lvl=0, int plidx=-1) {
   int   i,
         New_idx,
         tmp_idx,
         NL_idx, NL_tail, NL_head;


   // Increase the Max_Level if necessary.
   if (Max_Level < New_lvl) {
      Max_Level = New_lvl;
      Path_Level[New_lvl].Num_of_Paths = 0;
   }

   // Verify if we need to modify the size of the structure at the
   // current level.
   if (!incr_Path_Level_size(New_lvl))
```

```
      return -1;

   // Obtain the "New_idx" at the current "New_lvl".
   New_idx = Path_Level[New_lvl].Num_of_Paths;

   // Set the correct size for this structure.
   Path_Level[New_lvl].Path_Array[New_idx].size   = 0;

   if (!incr_Segment_Array_size(New_lvl,
                                New_idx,
                                Path_Level[lvl].Path_Array[idx].size))
      return -1;

   // Set the preliminary data of the new structure.
   Path_Level[New_lvl].Path_Array[New_idx].type   =
       Path_Level[lvl].Path_Array[idx].type;
   Path_Level[New_lvl].Path_Array[New_idx].chlng  =
       Path_Level[lvl].Path_Array[idx].chlng;
   Path_Level[New_lvl].Path_Array[New_idx].length =
       Path_Level[lvl].Path_Array[idx].length;
   Path_Level[New_lvl].Path_Array[New_idx].plidx  = plidx;


   // Add each of the segments to this structure.
   for (i=0; i<Path_Level[lvl].Path_Array[idx].length; i++) {
      // What is the index of this segment?
      tmp_idx = Path_Level[lvl].Path_Array[idx].Segment_Array[i].idx;
      NL_tail = Path_Level[lvl].Path_Array[idx].Segment_Array[i].tail;
      NL_head = Path_Level[lvl].Path_Array[idx].Segment_Array[i].head;

      // Is this segment an Arc or a Varc/Chain?
      if (Path_Level[lvl+1].Path_Array[tmp_idx].type == Arc)
         NL_idx = add_arc_to_Arc(NL_tail, NL_head, New_lvl+1, New_idx);
      else
         NL_idx = copy_Varc_or_Chain_to_DS(lvl+1, tmp_idx, New_lvl+1, New_idx);

      if (NL_idx == -1)
         return -1;

      // Add all the details of this segment.
      Path_Level[New_lvl].Path_Array[New_idx].Segment_Array[i].tail = NL_tail;
      Path_Level[New_lvl].Path_Array[New_idx].Segment_Array[i].head = NL_head;
      Path_Level[New_lvl].Path_Array[New_idx].Segment_Array[i].idx  = NL_idx;
   }

   // Finally, increase the number of valid paths at the "New_lvl".
   Path_Level[New_lvl].Num_of_Paths++;

   return New_idx;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::move_arc_to_next_level
//--------------------------------------------------------------------
// Moves an arc to the next level (c+1) of the Path_Level data
// structure.
// In order to make a clean move, it is necessary to shift all the
// elements of Path_Level at the current level and then delete the
// last one, because this will create a cleaner structure at the
// current level (although this means more operations per move).
//
// WARNING: It has been observed that when this procedure calls
//    itself, it creates a nesting effect that may loose track of
//    some indexes. Becaue of this, we need to check the indexes
//    after the nesting has finished. The global "Nested" variable
//    helps to keep track of how many nestings have occurred. This
//    variable sould be reseted before other procedures call this
//    one and we are sure no nesting can exist.
//--------------------------------------------------------------------
// Input:
//    a) The tail of the Arc that is moved.
//    b) The head of the Arc that is moved.
//    c) The current level of the Arc.
//    d) The current index of the Arc in Path_Level[].Path_Array.
// Output:
//    Returns the index where the arc resides after being moved to
//    the next level, or -1 if the operation was unsuccesful.
////////////////////////////////////////////////////////////////////
inline int chain_class::move_arc_to_next_level
   (int a, int b, int c, int d) {
   int    i, idx, lst, parent;


   // Increase the Max_Level if necessary.
   if (Max_Level < (c+1)) {
      Max_Level = c+1;
      Path_Level[c+1].Num_of_Paths = 0;
   }


   // If this is a Varc or Chain, we need to move the arcs in
   // Segment_Array to the next level.
   // The "parent" variable keeps track of how many levels of Nesting
   // have been aplied while moving the arcs. When the nesting is 2
   // or more it is possible that some of the indexes have moved.
   // The Nested variable should be reset to zero by procedures that
   // directly call this procedure without any nesting effect.
   if (Arc != Path_Level[c].Path_Array[d].type) {
      parent = Nested++;

      for (i=0; i<Path_Level[c].Path_Array[d].length; i++) {
         idx = move_arc_to_next_level(
               Path_Level[c].Path_Array[d].Segment_Array[i].tail,
               Path_Level[c].Path_Array[d].Segment_Array[i].head,
               c+1,
               Path_Level[c].Path_Array[d].Segment_Array[i].idx);

         Path_Level[c].Path_Array[d].Segment_Array[i].idx = idx;
      }

      // If the Nesting is more than one, we need to verify that no
      // index has lost track of the Segments that have been moved to
      // the next level. If a discrepancy in the indexes is found, we
      // will try to fix it by calling function "find_last_index_at_
      // specific_level.
      if ((Nested - parent) > 1) {
         Nested--;

         for (i=0; i<Path_Level[c].Path_Array[d].length; i++) {
            idx = Path_Level[c].Path_Array[d].Segment_Array[i].idx;
            lst = Path_Level[c+2].Path_Array[idx].length-1;

            if ((Path_Level[c].Path_Array[d].Segment_Array[i].tail !=
                 Path_Level[c+2].Path_Array[idx].Segment_Array[0].tail)
                 ||
                (Path_Level[c].Path_Array[d].Segment_Array[i].head !=
                 Path_Level[c+2].Path_Array[idx].Segment_Array[lst].head)) {
               idx = find_last_index_at_specific_level(c+2,
                     Path_Level[c].Path_Array[d].Segment_Array[i].tail,
                     Path_Level[c].Path_Array[d].Segment_Array[i].head);

               Path_Level[c].Path_Array[d].Segment_Array[i].idx = idx;
            }
         }
      }
   }


   // Start by defining the index of the current arc.
   idx = Path_Level[c+1].Num_of_Paths;

   if (!incr_Path_Level_size(c+1))
      return -1;

   // Let's move the new Arc to Level a+1.
   Path_Level[c+1].Path_Array[idx].type   = Path_Level[c].Path_Array[d].type;
   Path_Level[c+1].Path_Array[idx].chlng  = Path_Level[c].Path_Array[d].chlng;
   Path_Level[c+1].Path_Array[idx].length = Path_Level[c].Path_Array[d].length;
   Path_Level[c+1].Path_Array[idx].plidx  = d;

   Path_Level[c+1].Path_Array[idx].size = 0;
   if (!incr_Segment_Array_size(c+1, idx, Path_Level[c].Path_Array[d].size))
      return -1;

   // Copy each element of Segment_Array.
   for (i=0; i<Path_Level[c+1].Path_Array[idx].length; i++) {
      Path_Level[c+1].Path_Array[idx].Segment_Array[i] =
         Path_Level[c].Path_Array[d].Segment_Array[i];

      if (Arc != Path_Level[c+1].Path_Array[idx].type) {
         Path_Level[c+2].Path_Array
         [Path_Level[c+1].Path_Array[idx].Segment_Array[i].idx]
         .plidx = idx;
      }
   }
   Path_Level[c+1].Num_of_Paths++;

   if (Arc == Path_Level[c].Path_Array[d].type)
      // Add the new Arc location in the Arc Database.
      if (!add_arc_location(a, b, c+1, idx))
         return -1;
```

```
   // Delete this arc at the current level.
   if (!delete_1_element_from_DS(a, b, c, d, Path_Level[c].Path_Array[d].type))
      return -1;

   return idx;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::move_arc_to_prev_level
//------------------------------------------------------------------
// Moves an arc to the previous level (c-1) of the Path_Level data
// structure.
// In order to make a clean move, it is necessary to shift all the
// elements of Path_Level at the current level and then delete the
// last one, because this will create a cleaner structure at the
// current level (although this means more operations per move).
//------------------------------------------------------------------
// Input:
//    a) The tail of the Arc that is moved.
//    b) The head of the Arc that is moved.
//    c) The current level of the Arc.
//    d) The current index of the Arc in Path_Level[].Path_Array.
// Output:
//    Returns the index where the arc resides after being moved to
//    the next level, or -1 if the operation was unsuccesful.
////////////////////////////////////////////////////////////////////
inline int chain_class::move_arc_to_prev_level
(int a, int b, int c, int d) {
   int   j, idx, lst, parent;


   // If this is a Varc or Chain, we need to move the arcs in
   // Segment_Array to the next level.
   // The "parent" variable keeps track of how many levels of Nesting
   // have been aplied while moving the arcs. When the nesting is 2
   // or more it is possible that some of the indexes have moved.
   // The Nested variable should be reset to zero by procedures that
   // directly call this procedure without any nesting effect.
   if (Arc != Path_Level[c].Path_Array[d].type) {
      parent = Nested++;

      for (j=0; j<Path_Level[c].Path_Array[d].length; j++) {
         idx = move_arc_to_prev_level(
               Path_Level[c].Path_Array[d].Segment_Array[j].tail,
               Path_Level[c].Path_Array[d].Segment_Array[j].head,
               c+1,
               Path_Level[c].Path_Array[d].Segment_Array[j].idx);

         Path_Level[c].Path_Array[d].Segment_Array[j].idx = idx;
      }

      // If the Nesting is more than one, we need to verify that no
      // index has lost track of the Segments that have been moved to
      // the next level. If a discrepancy in the indexes is found, we
      // will try to fix it by calling function "find_last_index_at_
      // specific_level.
      if ((Nested - parent) > 1) {
         Nested--;

         for (j=0; j<Path_Level[c].Path_Array[d].length; j++) {
            idx = Path_Level[c].Path_Array[d].Segment_Array[j].idx;
            lst = Path_Level[c].Path_Array[idx].length-1;

            if ((Path_Level[c].Path_Array[d].Segment_Array[j].tail !=
                 Path_Level[c].Path_Array[idx].Segment_Array[0].tail)
                 ||
                (Path_Level[c].Path_Array[d].Segment_Array[j].head !=
                 Path_Level[c].Path_Array[idx].Segment_Array[lst].head)) {
               idx = find_last_index_at_specific_level(c,
                     Path_Level[c].Path_Array[d].Segment_Array[j].tail,
                     Path_Level[c].Path_Array[d].Segment_Array[j].head);

               Path_Level[c].Path_Array[d].Segment_Array[j].idx = idx;
            }
         }
      }
   }


   // Start by defining the index of the current arc.
   idx = Path_Level[c-1].Num_of_Paths;

   if (!incr_Path_Level_size(c-1))
      return -1;

   // Let's move the new Arc to Level a-1.
   Path_Level[c-1].Path_Array[idx].type   = Path_Level[c].Path_Array[d].type;
   Path_Level[c-1].Path_Array[idx].chlng  = Path_Level[c].Path_Array[d].chlng;
   Path_Level[c-1].Path_Array[idx].length = Path_Level[c].Path_Array[d].length;
   Path_Level[c-1].Path_Array[idx].plidx  = -1;

   Path_Level[c-1].Path_Array[idx].size = 0;
   if (!incr_Segment_Array_size(c-1, idx, Path_Level[c].Path_Array[d].size))
      return -1;

   // Copy each element of Segment_Array.
   for (j=0; j<Path_Level[c-1].Path_Array[idx].length; j++) {
      Path_Level[c-1].Path_Array[idx].Segment_Array[j] =
         Path_Level[c].Path_Array[d].Segment_Array[j];

      if (Arc != Path_Level[c-1].Path_Array[idx].type) {
         Path_Level[c].Path_Array
         [Path_Level[c-1].Path_Array[idx].Segment_Array[j].idx]
         .plidx = idx;
      }
   }
   Path_Level[c-1].Num_of_Paths++;

   if (Arc == Path_Level[c].Path_Array[d].type)
      // Add the new arc location in the Arc Database.
      if (!add_arc_location(a, b, c-1, idx))
         return -1;

   // Now delete the Arc at the current level.
   if (!delete_1_element_from_DS(a, b, c, d, Path_Level[c].Path_Array[d].type))
      return -1;

   return idx;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::add_arc_to_Varc
//------------------------------------------------------------------
// If possible, add new arc into a suitable Varc.
//------------------------------------------------------------------
// Input:
//    a) The tail of the New Arc (also the head of the previous arc).
//    b) The head of the New Arc.
//    c) The tail of the Previous Arc.
// Output:
//    Returns true if it has been possible to add the Arc into a
//    Varc, and false otherwise.
////////////////////////////////////////////////////////////////////
inline bool chain_class::add_arc_to_Varc(int a, int b, int c) {
   bool  option2=false, option4=false;
   int   i, j, k, cnt,
         PVarc_level,   // The level of the Previous Arc
         PVarc_index,   // The index of the Previous Arc
         index1,
         index2,
         Abst_lvl,      // Abstracted arc level (stores the Prev Arc)
         Abst_idx,      // Abstracted arc index (stores the Prev Arc)
         Abst_last,     // Abstracted arc's last element index.
         New_idx;       // Index of the new Varc.


   // 1st Step:
   // Find the "Previous Arc" in the Arc Database. Variable i is the
   // index into Arc_Array, and cnt is the number of locations that
   // the Previous Arc has recorded in the database.
   i   = find_arc_in_db(c, a);
   cnt = query_arc_db_for_loc_count(i);

   if (!cnt)
      return false;


   // 2nd Step:
   // Process each location found for the "Previous Arc".
   for (j=0; j<cnt; j++) {
      // Find the level and index of the Previous Arc's current
      // location.
      PVarc_level = query_arc_db_for_arc_level(i, j);
      PVarc_index = query_arc_db_for_arc_index(i, j);

      //-------------------------------------------------------------
      // We have 4 options when including the "New Arc" into the
      // chain_class data structure:
      // 1. If the Previous Arc is stored individually, create a new
      //    Varc = Previous Arc + New Arc.
```

```
// 2. If the Previous Arc is part of a Chain or a Varc that is
//    also part of a Chain, the New Arc needs to be stored
//    individually.
// 3. If the Previous Arc was the last segment of a Varc, add
//    the New Arc to this Varc = Varc + New Arc.
// 4. If the Previous Arc was part of a Varc (but not the last
//    segment), create a new Varc that contains the 1st part of
//    the original Varc and the New Arc.  (DISABLED!!!)
//-------------------------------------------------------------

// Option 1:
//-------------------------------------------------------------
// If the level of the Previous Arc is zero, then it has been
// stored as an individual arc. Convert both arcs into a Varc.
if (PVarc_level == 0) {
   // Move the Previous Arc from Level 0 to Level 1.
   Nested = 0;
   index1 = move_arc_to_next_level(c, a, 0, PVarc_index);

   // Add the New Arc to Level 1.
   index2 = add_arc_to_Arc(a, b, 1);

   // Return false if any of the previous functions has failed.
   if ((index1 == -1) || (index2 == -1))
      return false;

   // At Level 0, convert the Previous Arc into a Varc.
   if (!incr_Path_Level_size(0))
      return false;

   New_idx = Path_Level[0].Num_of_Paths;
   Path_Level[1].Path_Array[index1].plidx = New_idx;
   Path_Level[1].Path_Array[index2].plidx = New_idx;

   Path_Level[0].Path_Array[New_idx].type   = Varc;
   Path_Level[0].Path_Array[New_idx].chlng  = 0;
   Path_Level[0].Path_Array[New_idx].length = 2;
   Path_Level[0].Path_Array[New_idx].plidx  = -1;

   Path_Level[0].Path_Array[New_idx].size   = 0;
   if (!incr_Segment_Array_size(0, New_idx, 2))
      return false;

   // Add the Previous Arc and the New Arc as the 2 new
   // segments of this Varc.
   Path_Level[0].Path_Array[New_idx].Segment_Array[0].tail = c;
   Path_Level[0].Path_Array[New_idx].Segment_Array[0].head = a;
   Path_Level[0].Path_Array[New_idx].Segment_Array[0].idx  = index1;
   Path_Level[0].Path_Array[New_idx].Segment_Array[1].tail = a;
   Path_Level[0].Path_Array[New_idx].Segment_Array[1].head = b;
   Path_Level[0].Path_Array[New_idx].Segment_Array[1].idx  = index2;

   Path_Level[0].Num_of_Paths++;
}
else {
   // The level of the Previous Arc is greater than zero, hence
   // this arc has been abstracted before and it must be part
   // of either a Chain or a Varc (Options 2, 3 or 4).

   // Find more info at the previous abstracted level.
   Abst_lvl = PVarc_level-1;
   Abst_idx = Path_Level[PVarc_level].Path_Array[PVarc_index].plidx;

   // Option 2:
   //----------------------------------------------------------
   // Verify if the abstracted structure is either part of
   // another abstracted structure (Abst_lvl is not 0) or a
   // Chain.
   if ((Abst_lvl > 0) ||
       (Path_Level[Abst_lvl].Path_Array[Abst_idx].type == Chain)) {
      // Since a Chain is complete, there is no possible Arc
      // that could be added to it, so the New Arc has to be
      // stored as an independent arc.

      // But only once.
      if (option2)
         continue;
      else
         option2 = true;

      index1 = add_arc_to_Arc(a, b);

      if (index1 == -1)
         return false;
   }
   else {
      // The abstracted structure is not a Chain, so it has to
      // be a Varc (Options 3 or 4).

      // The index of the last segment of the abstracted
      // structure.
      Abst_last = Path_Level[Abst_lvl].Path_Array[Abst_idx].length-1;

      // Option 3:
      //----------------------------------------------------------
      // If the Previous Arc is the last segment of this Varc,
      // append the New Arc at the end of this Varc.
      if (Path_Level[Abst_lvl].Path_Array[Abst_idx].
          Segment_Array[Abst_last].idx == PVarc_index) {
         index1 = add_arc_to_Arc(a, b, PVarc_level, Abst_idx);

         if (index1 == -1)
            return false;

         // Modify the Varc to include the New Arc as the last
         // segment.
         Abst_last++;
         Path_Level[Abst_lvl].Path_Array[Abst_idx].length++;

         if (!incr_Segment_Array_size(Abst_lvl, Abst_idx,
              Path_Level[Abst_lvl].Path_Array[Abst_idx].length))
            return false;

         Path_Level[Abst_lvl].Path_Array[Abst_idx].
         Segment_Array[Abst_last].tail = a;
         Path_Level[Abst_lvl].Path_Array[Abst_idx].
         Segment_Array[Abst_last].head = b;
         Path_Level[Abst_lvl].Path_Array[Abst_idx].
         Segment_Array[Abst_last].idx  = index1;
      }
      else {
         // Option 4:
         //----------------------------------------------------------
         // The Previous Arc is not the last one in the Varc,
         // this means that the New Arc is a new branch of a
         // tree. This means that we need to create a new Varc
         // that includes the previous arcs of the 1st Varc and
         // the New Arc.

         // But only once.
         if (option4)
            continue;
         else
            option4 = true;

         cout << "How did we get in this mess??????\n\n";
         index1 = add_arc_to_Arc(a, b);

         if (index1 == -1)
            return false;
/*         New_idx = Path_Level[Abst_lvl].Num_of_Paths;

         if (!incr_Path_Level_size(Abst_lvl))
            return false;

         Path_Level[Abst_lvl].Path_Array[New_idx].type   = Varc;
         Path_Level[Abst_lvl].Path_Array[New_idx].chlng  = 0;
         Path_Level[Abst_lvl].Path_Array[New_idx].length = 0;
         Path_Level[Abst_lvl].Path_Array[New_idx].plidx  = -1;
         Path_Level[Abst_lvl].Path_Array[New_idx].size   = 0;

         // Copy all the segments stored up to the Previous
         // Arc. Notice that although a while loop could be
         // more suitable, it is safer to use a for loop.
         for (k=0; k<Abst_last; k++) {
            // Add the new location of each copied arc.
            index1 = add_arc_to_Arc(
Path_Level[Abst_lvl].Path_Array[Abst_idx].Segment_Array[k].tail,
Path_Level[Abst_lvl].Path_Array[Abst_idx].Segment_Array[k].head,
            PVarc_level,
            New_idx);

            if (index1 == -1)
               return false;

            if (!incr_Segment_Array_size(Abst_lvl, New_idx, k+1))
               return false;

            // Add each copied arc as a new segment of this
            // Varc.
Path_Level[Abst_lvl].Path_Array[New_idx].Segment_Array[k].tail =
Path_Level[Abst_lvl].Path_Array[Abst_idx].Segment_Array[k].tail;

Path_Level[Abst_lvl].Path_Array[New_idx].Segment_Array[k].head =
```

```
            Path_Level[Abst_lvl].Path_Array[Abst_idx].Segment_Array[k].head;

            Path_Level[Abst_lvl].Path_Array[New_idx].Segment_Array[k].idx  =
            index1;

                  Path_Level[Abst_lvl].Path_Array[New_idx].length++;

                  // Check if we have just copied the Previous Arc
                  if (Path_Level[Abst_lvl].Path_Array[Abst_idx].
                      Segment_Array[k].idx == PVarc_index)
                     break;
               }

               // Add the New Arc at the same level as the Previous Arc.
               index1 = add_arc_to_Arc(a, b, PVarc_level, New_idx);

               if (index1 == -1)
                  return false;

               k++;
               if (!incr_Segment_Array_size(Abst_lvl, New_idx, k+1))
                  return false;

               // Add the New Arc.
               Path_Level[Abst_lvl].Path_Array[New_idx].Segment_Array[k].tail =
                a;
               Path_Level[Abst_lvl].Path_Array[New_idx].Segment_Array[k].head =
                b;
               Path_Level[Abst_lvl].Path_Array[New_idx].Segment_Array[k].idx  =
                index1;

               Path_Level[Abst_lvl].Path_Array[New_idx].length++;
               Path_Level[Abst_lvl].Num_of_Paths++;
*/          }
         }
      }
   }

   // When we have finished processing each location of the Previous
   // Arc we can return true.
   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::disassemble_Varc
//-----------------------------------------------------------------
// This function has not been completelly tested!!!!
// Some definitions of the arcs used need to be provided:
// If we originally have arcs V1V2 and V1V3 and the New Arc is V2V3,
// We will try to process the Chain (V1, V2, V3). The arcs will be
// labeled as follows:
//-----------------------------------------------------------------
// Input:
//    a) The tail of the New Arc (also the head of the A arc).
//    b) The head of the New Arc (also the head of the B Arc).
//    c) The tail of the A Arc.
//    d) The tail of the B Arc.
// Output:
//    Returns the index of Path_Level[c].Path_Array where the arc now
//    resides, or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
// Obsoleted function
// Replaced by: get_segment_from_Varc_provide_Varc_info
// Date: 12/2/09
/*inline bool chain_class::disassemble_Varc(int level, int index, int a, int b) {
   bool  found_B;
   int   i, j, k,
         state=0, max_state=3,
         tail, head, idx,
         Last_idx, New_idx;
//   static  int   ret_array[2];

   Last_idx = Path_Level[level].Path_Array[index].length-1;

   if ((b-a) == Last_idx) {
      // Nothing to do.
      return true;
   }

   for (i=0; i<max_state; i++) {
      switch (state) {
         case 0: {
            // Verify that we have a Beginning Section.
            if (a != 0) {
               // Do we need to move just 1 Arc, or do we need to
               // create a new Varc.
               if (a == 1) {
                  // Only need to move the first Arc.
                  tail = Path_Level[level].Path_Array[index].Segment_Array[0].
                         tail;
                  head = Path_Level[level].Path_Array[index].Segment_Array[0].
                         head;
                  idx  = Path_Level[level].Path_Array[index].Segment_Array[0].
                         idx;

                  New_idx = move_arc_to_prev_level(tail, head, level+1, idx);

                  Path_Level[level].Path_Array[New_idx].plidx =
                      Path_Level[level].Path_Array[idx].plidx;
               }
               else {
                  // Need to create a new Varc for the Beginning
                  // Section.
                  New_idx = Path_Level[level].Num_of_Paths;

                  if (!incr_Path_Level_size(level))
                     return false;

                  Path_Level[level].Path_Array[New_idx].type   = Varc;
                  Path_Level[level].Path_Array[New_idx].chlng  = 0;
                  Path_Level[level].Path_Array[New_idx].length = 0;
                  Path_Level[level].Path_Array[New_idx].plidx  =
                     Path_Level[level].Path_Array[index].plidx;

                  for (j=0; j<a; j++) {
                     // Modify the location of each copied arc.
                     if (!mod_arc_location(Path_Level[level].Path_Array[index].
                                           Segment_Array[j].tail,
                                         Path_Level[level].Path_Array[index].
                                           Segment_Array[j].head,
                                         level, index, level, New_idx))
                        return false;

                     if (!incr_Segment_Array_size(level, New_idx, j+1))
                       return false;

                     // Add each copied arc as a new segment of this
                     // Varc and modify the previous level index of
                     // each Arc.
                     Path_Level[level].Path_Array[New_idx].
                     Segment_Array[j].tail = Path_Level[level].
                     Path_Array[index].Segment_Array[j].tail;

                     Path_Level[level].Path_Array[New_idx].
                     Segment_Array[j].head = Path_Level[level].
                     Path_Array[index].Segment_Array[j].head;

                     Path_Level[level].Path_Array[New_idx].
                     Segment_Array[j].idx  = Path_Level[level].
                     Path_Array[index].Segment_Array[j].idx;

                     Path_Level[level+1].
                     Path_Array[Path_Level[level].Path_Array[index].
                                Segment_Array[j].idx].
                     plidx = New_idx;

                     Path_Level[level].Path_Array[New_idx].length++;
                  }

                  Path_Level[level].Num_of_Paths++;
               }
            }

            // When we reach this line it is time to move to the next
            // state.
            state++;
            break;
         }

         case 1: {
            // Verify that we have and Ending Section.
            if (b != Last_idx) {
               // Do we need to move just 1 Arc, or do we need to
               // create a new Varc.
               if (b == (Last_idx-1)) {
                  // Only need to move the last Arc.
                  tail = Path_Level[level].Path_Array[index].
                         Segment_Array[Last_idx].tail;
                  head = Path_Level[level].Path_Array[index].
                         Segment_Array[Last_idx].head;
                  idx  = Path_Level[level].Path_Array[index].
                         Segment_Array[Last_idx].idx;
```

```
         New_idx = move_arc_to_prev_level(tail, head, level+1, idx);

         Path_Level[level].Path_Array[New_idx].plidx =
            Path_Level[level].Path_Array[index].plidx;
      }
      else {
         // Need to create a new Varc for the Ending
         //Section.
         New_idx = Path_Level[level].Num_of_Paths;

         if (!incr_Path_Level_size(level))
            return false;

         Path_Level[level].Path_Array[New_idx].type   = Varc;
         Path_Level[level].Path_Array[New_idx].chlng  = 0;
         Path_Level[level].Path_Array[New_idx].length = 0;
         Path_Level[level].Path_Array[New_idx].plidx  =
            Path_Level[level].Path_Array[index].plidx;

         k = 0;
         for (j=b+1; j<=Last_idx; j++) {
            // Modify the location of each copied arc.
            if (!mod_arc_location(Path_Level[level].Path_Array[index].
                                  Segment_Array[j].tail,
                                 Path_Level[level].Path_Array[index].
                                  Segment_Array[j].head,
                                 level, index, level, New_idx))
               return false;

            if (!incr_Segment_Array_size(level, New_idx, k+1))
              return false;

            // Add each copied arc as a new segment of this
            // Varc and modify the previous level index of
            // each Arc.
            Path_Level[level].Path_Array[New_idx].
            Segment_Array[k].tail = Path_Level[level].
            Path_Array[index].Segment_Array[j].tail;

            Path_Level[level].Path_Array[New_idx].
            Segment_Array[k].head = Path_Level[level].
            Path_Array[index].Segment_Array[j].head;

            Path_Level[level].Path_Array[New_idx].
            Segment_Array[k].idx  = Path_Level[level].
            Path_Array[index].Segment_Array[j].idx;

            Path_Level[level+1].
            Path_Array[Path_Level[level].Path_Array[index].
                       Segment_Array[j].idx].
            plidx = New_idx;

            Path_Level[level].Path_Array[New_idx].length++;
            k++;
         }

         Path_Level[level].Num_of_Paths++;
      }
   }

   // When we reach this line it is time to move to the next
   // state.
   state++;
   break;
}

case 2: {
   // Process the Middle Section.
   if (a == b) {
      // Only need to disassemble an Arc. So we convert the
      // current Varc into this Arc.
      Path_Level[level].Path_Array[index].Segment_Array[0].tail =
      Path_Level[level].Path_Array[index].Segment_Array[a].tail;

      Path_Level[level].Path_Array[index].Segment_Array[0].head =
      Path_Level[level].Path_Array[index].Segment_Array[a].head;

      Path_Level[level].Path_Array[index].Segment_Array[0].idx = -1;

      Path_Level[level].Path_Array[index].type   = Arc;
      Path_Level[level].Path_Array[index].length = 1;

      //if (!decr_Segment_Array_size(level, index))
      //   return false;

      if (!delete_l_element_from_DS(
            Path_Level[level].Path_Array[index].Segment_Array[a].tail,
            Path_Level[level].Path_Array[index].Segment_Array[a].head,
            level+1,
            Path_Level[level].Path_Array[index].Segment_Array[a].idx))
         return false;
      }
      else {
         // Need to disassemble a Varc. So we just modify this
         // Varc.
         k = 0;
         for (j=a; j<b; j++) {
            if (!delete_l_element_from_DS(
               Path_Level[level].Path_Array[index].Segment_Array[k].tail,
               Path_Level[level].Path_Array[index].Segment_Array[k].head,
               level+1,
               Path_Level[level].Path_Array[index].Segment_Array[k].idx))
               return false;

            // Move each arc to the correct segment of this
            // Varc. The loaction of each Arc stays.
            Path_Level[level].Path_Array[index].Segment_Array[k].tail =
            Path_Level[level].Path_Array[index].Segment_Array[j].tail;

            Path_Level[level].Path_Array[index].Segment_Array[k].head =
            Path_Level[level].Path_Array[index].Segment_Array[j].head;

            Path_Level[level].Path_Array[index].Segment_Array[k].idx  =
            Path_Level[level].Path_Array[index].Segment_Array[j].idx;

            Path_Level[level].Path_Array[index].length--;
            k++;
         }
            if (!decr_Segment_Array_size(level, index))
               return false;

         }

         // When we reach this line it is time to terminate this
         // function.
         state++;
         break;
      }

      default:
         cerr << "ERROR: Could not disassemble Varc at\n";
         cerr << "       Level " << level << "   Index " << index
              << "   From " << a << "   To " << b << "\n";
         return false;
      }
   }

   return true;
}*/


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::duplicate_Varc_segment
//-----------------------------------------------------------------
// This function duplicates either:
// 1. A complete Varc with all its segments.
// 2. A section of a Varc with its corresponding segments.
// 3. A single segment of a Varc.
// In order to use this function it is necessary to provide a
// starting and ending point of the segments we need to duplicate,
// as well as the Level where we want the duplicated part to reside.
//-----------------------------------------------------------------
// Input:
//    a) The level of the Varc.
//    b) The index of the Varc.
//    c) The starting point of the segment we need to duplicate.
//    d) The ending point of the segment we need to duplicate.
//    e) The level where we want the duplicate to reside
// Output:
//    Returns the index where the duplicated part now resides,
//    or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
inline int chain_class::duplicate_Varc_segment
   (int lvl, int idx, int a, int b, int New_lvl) {
   int   i, j,
         tail,
         head,
         Last_idx;


   Last_idx = Path_Level[lvl].Path_Array[idx].length-1;
```

```
   // Do we need to duplicate the whole Varc?
   if ((b-a) == Last_idx) {
      // If the Level of this Varc is greater than zero it means that
      // is being used by another structure (a Chain). If not then it
      // is just a lonely Varc hanging by itself, let's use it.
      if (lvl > 0)
         return copy_Varc_or_Chain_to_DS(lvl, idx, New_lvl);
      else {
         if (lvl == New_lvl)
            return idx;
         else {
            tail = Path_Level[lvl].Path_Array[idx].Segment_Array[a].tail;
            head = Path_Level[lvl].Path_Array[idx].Segment_Array[Last_idx].head;

            copy_Varc_or_Chain_to_DS(lvl, idx, New_lvl);

            remove_elements_from_DS(lvl, idx);

            return find_last_index_at_specific_level(New_lvl, tail, head);
         }
      }
   }



   // Process the segments we need to duplicate.
   if (a == b) {
      Last_idx = Path_Level[lvl].Path_Array[idx].Segment_Array[a].idx;

      if (Arc == Path_Level[lvl+1].Path_Array[Last_idx].type) {
         // Only need to duplicate a single Arc.
         tail = Path_Level[lvl].Path_Array[idx].Segment_Array[a].tail;
         head = Path_Level[lvl].Path_Array[idx].Segment_Array[a].head;

         return add_arc_to_Arc(tail, head, New_lvl);
      }
      else
         return copy_Varc_or_Chain_to_DS(lvl+1, Last_idx, New_lvl);
   }
   else {
      // Need to create the path of the new Varc.
      j = 0;

      int * path = new int[Last_idx];

      for (i=a; i<b; i++)
         path[j++] = Path_Level[lvl].Path_Array[idx].Segment_Array[i].tail;

      path[j] = Path_Level[lvl].Path_Array[idx].Segment_Array[i].tail;

      return create_path_Varc(New_lvl, j, path, NULL);
   }
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::get_segment_from_Varc_provide_Arc_info
//----------------------------------------------------------------
// Obtain a segment of a Varc delimited by two vertices: a and b.
// This function requires that the level and index of one of the
// individual arcs that form the Varc are provided.
//----------------------------------------------------------------
// Input:
//    a) The level of an Arc that form part of the Varc.
//    b) The index of an Arc that form part of the Varc.
//    c) The vertex where the desired segment starts.
//    d) The vertex where the desired segment finishes.
// Output:
//    Returns the index where the segment now resides,
//    or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
// Obsoleted function
// Replaced by: get_segment_from_Varc_provide_Varc_info
// Date: 12/2/09
/*inline int chain_class::get_segment_from_Varc_provide_Arc_info
   (int level, int index, int a, int b) {
   int   k,
         Start=-1,
         End=-1,
         Abstr_level,
         Abstr_index,
         Abstr_last;

   // Find info at the previous abstracted level (Varc).
   Abstr_level = level-1;
   Abstr_index = Path_Level[level].Path_Array[index].plidx;
   Abstr_last  = Path_Level[Abstr_level].Path_Array[Abstr_index].length-1;

   // Find the segment starting from the last vertex.
   for (k=Abstr_last; k>=0; k--) {
      // Find if we have found the last vertex.
      if (b == Path_Level[Abstr_level].Path_Array[Abstr_index].
               Segment_Array[k].head)
         End = k;

      // Find if we have found the 1st vertex.
      if (a == Path_Level[Abstr_level].Path_Array[Abstr_index].
               Segment_Array[k].tail) {
         Start = k;

         if (disassemble_Varc(Abstr_level, Abstr_index, Start, End))
            return Abstr_index;
      }
   }

   // If we hit this line we could not obtain the segment.
   return -1;
}*/


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::get_segment_from_Varc_provide_Varc_info
//----------------------------------------------------------------
// Obtain a segment of a Varc delimited by two vertices: a and b.
// This function requires that the level and index of the Varc
// are provided.
//----------------------------------------------------------------
// Input:
//    a) The level of the Varc.
//    b) The index of the Varc.
//    c) The vertex where the desired segment starts.
//    d) The vertex where the desired segment finishes.
//    e) The level where the desired Varc will reside.
// Output:
//    Returns the index where the segment now resides,
//    or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
inline int chain_class::get_segment_from_Varc_provide_Varc_info
   (int level, int index, int a, int b, int New_level) {
   int   k,
         Start,
         End=-1;


   // Find the segment starting from the last vertex.
   for (k=(Path_Level[level].Path_Array[index].length-1); k>=0; k--) {
      // Find if we have found the last vertex.
      if (b == Path_Level[level].Path_Array[index].Segment_Array[k].head)
         End = k;


      // Find if we have found the 1st vertex.
      if ((a == Path_Level[level].Path_Array[index].Segment_Array[k].tail)
          &&
          (End != -1)) {
         Start = k;

         return duplicate_Varc_segment(level, index, Start, End, New_level);
      }
   }

   // If we hit this line we could not obtain the segment.
   return -1;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::create_path_Varc
//----------------------------------------------------------------
// Creates a new Varc, which is defined by an array of vertices.
//
// WARNING: This procedure does not verify that the vertices provided
//    really form a sensible Varc. It is responsibility of the parent
//    procedure to check the sanity of this data.
//----------------------------------------------------------------
// Input:
//    a) The level where the Varc will be created.
//    b) The number of vertices that will form the Varc.
//    c) The array that contains the vertices that form the Varc.
//    d) Optional array that indicates the type of each segment of
```

```
//       the Varc. By default, all the segments of the Varc are of
//       type Arc. The length of this array is (size-1).
// Output:
//    Returns the index of the Varc created, or -1 if the operation
//    failed.
////////////////////////////////////////////////////////////////////
inline int chain_class::create_path_Varc
   (int level, int size, int *vtx_ary, linktype *sgmt_ary=NULL) {
   int   i,
         index,
         sgmt_idx,
         sgmt_var;


   // The index of the new Varc.
   index = Path_Level[level].Num_of_Paths;

   if (!incr_Path_Level_size(level))
      return -1;

   Path_Level[level].Path_Array[index].type   = Varc;
   Path_Level[level].Path_Array[index].chlng  = 0;
   Path_Level[level].Path_Array[index].length = size-1;
   Path_Level[level].Path_Array[index].plidx  = -1;

   Path_Level[level].Path_Array[index].size   = 0;
   if (!incr_Segment_Array_size(level, index, (size-1)))
      return -1;

   // Because we may need to move arcs at the current level, it is
   // necessary to update "Num_of_Paths" and initialize indexes here.
   Path_Level[level].Num_of_Paths++;
   for (i=0; i<(size-1); i++)
      Path_Level[level].Path_Array[index].Segment_Array[i].idx = 0;

   // Add each segment of the new Varc, and either each new instance
   // of each Arc, or move the corresponding Varc/Chain.
   for (i=0; i<(size-1); i++) {
      if ((sgmt_ary) && (sgmt_ary[i] != Arc)) {
         // Here "sgmt_var" is the index where the Varc resides at
         // the specified level.
         sgmt_var = find_last_index_at_specific_level(level, vtx_ary[i],
                                                      vtx_ary[i+1]);
         if (sgmt_var == -1) {
            Path_Level[level].Path_Array[index].length = 0;
            decr_Segment_Array_size(level, index);
            Path_Level[level].Num_of_Paths--;
            return -1;
         }

         Nested   = 0;
         sgmt_idx = move_arc_to_next_level(vtx_ary[i], vtx_ary[i+1],
                                           level, sgmt_var);
         index--;
      }
      else {
         // Here, "sgmt_var" is the level where this Arc will be
         // added to the DS.
         sgmt_var = level + 1;

         // Increase the Max_Level if necessary.
         if (Max_Level < sgmt_var) {
            Max_Level = sgmt_var;
            Path_Level[sgmt_var].Num_of_Paths = 0;
         }

         sgmt_idx = add_arc_to_Arc(vtx_ary[i], vtx_ary[i+1], sgmt_var, index);
      }

      if (sgmt_idx == -1) {
         Path_Level[level].Path_Array[index].length = 0;
         decr_Segment_Array_size(level, index);
         Path_Level[level].Num_of_Paths--;
         return -1;
      }

      Path_Level[level].Path_Array[index].Segment_Array[i].tail = vtx_ary[i];
      Path_Level[level].Path_Array[index].Segment_Array[i].head = vtx_ary[i+1];
      Path_Level[level].Path_Array[index].Segment_Array[i].idx  = sgmt_idx;
   }

   // When we have finished adding the Varc return the index.
   return index;
}

////////////////////////////////////////////////////////////////////
//
// Function: chain_class::find_last_index_at_specific_level
//-----------------------------------------------------------------
// Finds the last index of a segment (Arc, Varc or Chain) at a
// specified level.
// This procedure is useful when the index to a segment has been lost
// or is currently unknown.
//-----------------------------------------------------------------
// Input:
//    a) The level where we expect to find the segment.
//    b) The tail of the segment.
//    c) The head of the segment.
// Output:
//    Returns the index where the segment currently resides, or -1 if
//    the segment could not be found at this level.
////////////////////////////////////////////////////////////////////
inline int chain_class::find_last_index_at_specific_level
   (int level, int a, int b) {
   int   i, last;


   for (i=(Path_Level[level].Num_of_Paths-1); i>=0; i--) {
      last  = Path_Level[level].Path_Array[i].length-1;

      if ((a == Path_Level[level].Path_Array[i].Segment_Array[0].tail) &&
          (b == Path_Level[level].Path_Array[i].Segment_Array[last].head))
         return i;
   }

   // If we hit this line we could not find a match.
   return -1;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::find_arc_in_data_structure
//-----------------------------------------------------------------
// Finds a Varc or Chain in the data structure by specifying their
// 1st and last vertices.
// This procedure is useful when function "find_arc_in_db" cannot
// find a desired individual Arc.
//
// WARNING: Remember that this procedure is not intended to replace
//    "find_arc_in_db". It is just intended to use in cases where it
//    is necessary to enhance the search provided in the Arc DB.
//-----------------------------------------------------------------
// Input:
//    a) The tail of the Varc/Chain.
//    b) The head of the Varc/Chain.
//    c) Search options:
//       invalid: Look for either Chain or Varc (default option).
//       Varc: Look only for Varcs.
//       Chain: Look only for Chains.
// Output:
//    Returns a pointer to an array of length 2 which contains the
//    level (index 0) and index (index 1) where the Chain/Varc has
//    been found, or NULL if the desired segment could not be found.
////////////////////////////////////////////////////////////////////
inline int *chain_class::find_arc_in_data_structure
   (int a, int b, linktype c=invalid) {
   int   i, j, last;

   static int   ret_array[2];


   for (i=0; i<Max_Level; i++) {
      for (j=0; j<Path_Level[i].Num_of_Paths; j++) {
         // Remember that this procedure does not look for individual
         // Arcs.
         if (Path_Level[i].Path_Array[j].type == Arc)
            continue;

         switch (c) {
            case Varc: {
               // Look only for Varcs.
               if (Path_Level[i].Path_Array[j].type == Chain)
                  continue;
            }
            case Chain: {
               // Look only for Chains.
               if (Path_Level[i].Path_Array[j].type == Varc)
                  continue;
            }
         }

         last = Path_Level[i].Path_Array[j].length - 1;
```

```
         if ((a == Path_Level[i].Path_Array[j].Segment_Array[0].tail) &&
             (b == Path_Level[i].Path_Array[j].Segment_Array[last].head)) {
            // We have found a match.
            ret_array[0] = i;
            ret_array[1] = j;
            return ret_array;
         }
      }
   }

   // If we hit this line we could not find a match.
   return NULL;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::find_src_vertex_type
//--------------------------------------------------------------------
// Finds if Arc B and Arc A have a common source vertex at Level 0
// or, if Arc B belongs to another structure, finds if there is
// another segment that has a common source vertex (also at Level 0).
// In the lst case, this function will return a value of type "Arc",
// and on the later case, it will return either a value of "Chain" or
// "Varc" depending on the type of the structure at Level 0.
//    Remember...
//    Previous Arc: The arc that comes BEFORE the New Arc (V1V2), it
//       will be also labeled as "A arc".
//    Predecesor Arc: The arc that shares the FINAL vertex of the
//       chain (V1V3), it will be also labeled as "B arc".
//--------------------------------------------------------------------
// Input:
//    a) The level of B Arc (Predecesor).
//    b) The index of B Arc.
//    c) The tail of A arc (the source vertex), it should match the
//       tail of Arc B or the tail of another arc that belongs to the
//       same structure than B at Level 0.
// Output:
//    If a valid source vertex is found it will return the type of
//    structure where Arc B resides at Level 0; otherwise it will
//    return an "invalid" type value.
////////////////////////////////////////////////////////////////////
inline int *chain_class::find_src_vertex
   (int level, int index, int a) {
   int   i,
         Arc_tail,
         Varc_tail,
         This_tail,
         Abs_lvl,
         Abs_idx;

   static int   ret_array[3];


   // Initialize the return array values.
   ret_array[0] = 3;    // invalid type
   ret_array[1] = -1;   // invalid level
   ret_array[2] = -1;   // invalid index

   Arc_tail  = Path_Level[level].Path_Array[index].Segment_Array[0].tail;
   Varc_tail = Arc_tail;


   // First verify that B is an Arc.
   if (Arc != Path_Level[level].Path_Array[index].type) {
      cerr << "ERROR: Parameters \"Level\" and \"Index\" provided data\n"
           << "       for a structure that is not an Arc at:\n"
           << "       find_src_vertex\n\n";
      return ret_array;
   }


   // If the level of B is zero, it is stored currently only as an
   // Arc.
   if (level == 0) {
      // Verify that the tail of both Arcs match.
      if (a == Path_Level[level].Path_Array[index].Segment_Array[0].tail) {
         ret_array[0] = 0;      // Arc type
         ret_array[1] = level;
         ret_array[2] = index;
      }

      return ret_array;
   }

   // If the level of the Arc is NOT zero, this arc has been
   // abstracted before. Find the source vertex at a suitable
   // abstraction level.
   else {
      Abs_lvl = level;

      // Process the abstractions at previous levels until level zero
      // has been reached.
      while (Abs_lvl) {
         if (Abs_lvl == level)
            // Initialize the lst Abs_idx.
            Abs_idx = Path_Level[level].Path_Array[index].plidx;
         else
            // Use the Abs_idx provided by the data structure.
            Abs_idx = Path_Level[Abs_lvl].Path_Array[Abs_idx].plidx;

         Abs_lvl--;
         ret_array[1] = Abs_lvl;
         ret_array[2] = Abs_idx;


         // What type of structure do we have? Chain/Varc
         switch (Path_Level[Abs_lvl].Path_Array[Abs_idx].type) {
            case Chain: {
               // Find if there is a Chain segment in which the tails
               // match.
               // Note: Once we have found that the segment's tail
               // matches either "Arc_tail" of "Varc_tail" we must
               // stop the search to avoid looking on other segments
               // of this Chain.
               for (i=0; i<Path_Level[Abs_lvl].Path_Array[Abs_idx].length;
                    i++) {
                  This_tail =
                  Path_Level[Abs_lvl].Path_Array[Abs_idx].Segment_Array[i].tail;

                  // Have we found a match between "a" and any of the
                  // segments in the Chain?
                  if (a == This_tail) {
                     ret_array[0] = 2;   // Chain type
                     return ret_array;
                  }

                  // If this vertex's tail is the same as B's or
                  // Vtail at the previous level, continue processing
                  // at the next level (if any).
                  if ((Arc_tail == This_tail) || (Varc_tail == This_tail))
                     break;
               }

               // If we could not find a match set the type to
               // invalid and continue to the next level (if any).
               ret_array[0] = 3;   // invalid type
               Varc_tail    = Arc_tail;
               break;
            }
            case Varc: {
               // Find if there is a Varc segment in which the tails
               // match.
               for (i=0; i<Path_Level[Abs_lvl].Path_Array[Abs_idx].length;
                    i++) {
                  This_tail =
                  Path_Level[Abs_lvl].Path_Array[Abs_idx].Segment_Array[i].tail;

                  // Have we found a match between "a" and any of the
                  // segments in the Varc?
                  if (a == This_tail) {
                     ret_array[0] = 1;   // Varc type
                     return ret_array;
                  }

                  // Record the tail of the Varc to compare it with
                  // a Chain at the next level.
                  if (i == 0)
                     Varc_tail = This_tail;
               }

               // If we could not find a match set the type to
               // invalid and continue to the next level (if any).
               ret_array[0] = 3;   // invalid type
               break;
            }
            default: {
               cerr << "ERROR: Incorrect type of structure at:\n"
                    << "       find_src_vertex\n\n";
               ret_array[0] = 3;   // invalid type
               return ret_array;
            }
         }
```

```
      }
   }

   // If we reach this line, the most probable solution is that we
   // could not find a valid match.
   return ret_array;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::find_src_vertex_type
//--------------------------------------------------------------------
// Finds if Arc B and Arc A have a common source vertex at Level 0
// or, if Arc B belongs to another structure, finds if there is
// another segment that has a common source vertex (also at Level 0).
// In the 1st case, this function will return a value of type "Arc",
// and on the later case, it will return either a value of "Chain" or
// "Varc" depending on the type of the structure at Level 0.
//    Remember...
//    Previous Arc: The arc that comes BEFORE the New Arc (V1V2), it
//       will be also labeled as "A arc".
//    Predecesor Arc: The arc that shares the FINAL vertex of the
//       chain (V1V3), it will be also labeled as "B arc".
//--------------------------------------------------------------------
// Input:
//    a) The level of B Arc (Predecesor).
//    b) The index of B Arc.
//    c) The tail of A arc (the source vertex), it should match the
//       tail of Arc B or the tail of another arc that belongs to the
//       same structure than B at Level 0.
// Output:
//    If a valid source vertex is found it will return the type of
//    structure where Arc B resides at Level 0; otherwise it will
//    return an "invalid" type value.
////////////////////////////////////////////////////////////////////
inline linktype chain_class::convert_int_to_linktype
   (int a) {
   switch (a) {
      case 0:
         return Arc;
      case 1:
         return Varc;
      case 2:
         return Chain;
      case 3:
         return invalid;
      default: {
         cerr << "ERROR: Invalid integer value, which cannot\n"
              << "       convert to linktype.\n\n";
         return invalid;
      }
   }
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::create_simple_3point_Chain
//--------------------------------------------------------------------
// This function creates a very simple 3 vertex Chain using the
// following structure:
//    Chain: X-Y-Z
//    Segment 1: X-Y
//    Segment 2: X-Z
//    Segment 3: Y-Z
//--------------------------------------------------------------------
// Input:
//    a) The level where the Chain needs to be created.
//    b) The index (at the next level) of the 1st segment.
//    c) The index (at the next level) of the 2nd segment.
//    d) The index (at the next level) of the 3rd segment.
//    c) The tail of segment 1.
//    d) The head of segment 1.
//    e) The tail of segment 2 (should be the same as (c)).
//    f) The head of segment 2.
// Output:
//    Returns the index of the Chain added to the data structure. If
//    the operation fails, a -1 is returned.
////////////////////////////////////////////////////////////////////
inline int chain_class::create_simple_3point_Chain
   (int level, int idx_1, int idx_2, int idx_3,
    int tail,  int headA, int headB, int plidx=-1) {
   int    New_index;

   // Return false if an index has an invalid value.
   if ((idx_1 == -1) || (idx_2 == -1) || (idx_3 == -1)) {
      cerr << "ERROR: Invalid index values at procedure:\n"
           << "       create_simple_3point_Chain\n\n";
      return -1;
   }

   // Verify if we need to modify the size of the structure at the
   // current level.
   if (!incr_Path_Level_size(level))
      return -1;

   // Obtain the "New_index" at the current level.
   New_index = Path_Level[level].Num_of_Paths;

   // Set the correct size for this Chain (3 segments).
   Path_Level[level].Path_Array[New_index].size   = 0;
   if (!incr_Segment_Array_size(level, New_index, 3))
      return -1;
   // From this point, this procedure should not fail.

   // Set the index at the next level for each of the segments.
   Path_Level[level+1].Path_Array[idx_1].plidx = New_index;
   Path_Level[level+1].Path_Array[idx_2].plidx = New_index;
   Path_Level[level+1].Path_Array[idx_3].plidx = New_index;

   // Set the preliminary data of the new Chain.
   Path_Level[level].Path_Array[New_index].type   = Chain;
   Path_Level[level].Path_Array[New_index].chlng  = 3;
   Path_Level[level].Path_Array[New_index].length = 3;
   Path_Level[level].Path_Array[New_index].plidx  = plidx;

   // Add each of the 3 segments to this Chain.
   Path_Level[level].Path_Array[New_index].Segment_Array[0].tail = tail;
   Path_Level[level].Path_Array[New_index].Segment_Array[0].head = headA;
   Path_Level[level].Path_Array[New_index].Segment_Array[0].idx  = idx_1;

   Path_Level[level].Path_Array[New_index].Segment_Array[1].tail = tail;
   Path_Level[level].Path_Array[New_index].Segment_Array[1].head = headB;
   Path_Level[level].Path_Array[New_index].Segment_Array[1].idx  = idx_2;

   Path_Level[level].Path_Array[New_index].Segment_Array[2].tail = headA;
   Path_Level[level].Path_Array[New_index].Segment_Array[2].head = headB;
   Path_Level[level].Path_Array[New_index].Segment_Array[2].idx  = idx_3;

   // Finally, increase the number of valid paths at the current
   // level.
   Path_Level[level].Num_of_Paths++;

   return New_index;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::Chain_X_joins_Y
//--------------------------------------------------------------------
// This function processes 2 Chains and an Arc in order to find if it
// is possible to combine them into a new Chain.
// It is assumed that both chains share the 1st and the last vertex.
// We call this a joint relationship in which one Chain (X) joins
// another one (Y) using a joining arc AB (there could be more than
// one joining arc) which starts in X and finishes in Y.
// A later modifictation to this function allows the arc to be a
// Varc or Chain that has been formed before, and that should reside
// in 1 level more than the final combined Chain level (New_lvl+1).
//--------------------------------------------------------------------
// Input:
//    a) The basic level where all Chains reside.
//    b) The index of the A Chain.
//    c) The index of the B Chain.
//    d) The level of abstracted arc ab, or the tail of the new Arc.
//    e) The index of abstracted arc ab, or the head of the new Arc.
//    f) Type of argument that has been passed in (c) and (d). If
//       this argument is true, arc ab is a Varc/Chain and the
//       arguments represent the level and index of the abstracted
//       structure. But if this argument is false, arc ab is a new
//       individual Arc that also needs to be added to the data
//       structure.
// NOTE: It is necessary that the level of the existing abstraced arc
//    is 1 more than the final level after the 2 Chains have been
//    joined (New_lvl+1). If not, this procedure will fail to
//    correctly include arc ab into the combined Chain.
//
// Output:
//    Returns TRUE if it combined the 2 Chains and the Arc
//    successfully, otherwise it returns FALSE.
////////////////////////////////////////////////////////////////////
inline int chain_class::create_Chain_X_joins_Y
```

```
(int Lvl, int Aidx, int Bidx, int a, int b, bool data) {
bool   A_incl_B=false,
       B_incl_A=false;

int    i, j, x, y, Xz, Yz, idx,
       A, B,
       varA, varB,
       Alast, Blast,
       Yintsc,
       Xidx, Xlast,
       Yidx, Ylast,
       common_vtcs,
       banned_vtcs,
       NWidx,
       MP_vertics,
       MP_arc_idx,
       del0, del1, del2, del3;


// 1st Part:
// Find which Chain may be included into the other.
//-----------------------------------------------------------------
// We will label X as the 1st Chain that will join to the 2nd
// Chain Y, e.g.:
//    If   X = s-a-b-e   Y = s-c-e   and Arcs ac, bc exist
//    =>   X joins Y
//    Combined Chain = s-a-b-c-e
//-----------------------------------------------------------------
// Alast: The index to the last vertex of Chain A.
// Blast: The index to the last vertex of Chain B.
// Yintsc: The index to the segment where the head of the new arc
//    (AB) intersects Chain Y.
// Xidx(Yidx): The index of Chain X(Y).
// Xlast(Ylast): The index to the last segment of Chain X(Y).

// Verify the type of data that has been passed as argument.
if (data) {
   varA = Path_Level[a].Path_Array[b].length-1;
   A    = Path_Level[a].Path_Array[b].Segment_Array[0].tail;
   B    = Path_Level[a].Path_Array[b].Segment_Array[varA].head;
}
else {
   A   = a;
   B   = b;
}

Alast = Path_Level[Lvl].Path_Array[Aidx].chlng-2;
Blast = Path_Level[Lvl].Path_Array[Bidx].chlng-2;

for (i=(Alast-1); i>=0; i--) {
   if (Path_Level[Lvl].Path_Array[Aidx].Segment_Array[i].head == B) {
      B_incl_A = true;
      Yintsc = i;
      Xidx   = Bidx;
      Xlast  = Blast;
      Yidx   = Aidx;
      Ylast  = Alast;
      break;
   }
}

if (!B_incl_A) {
   for (i=(Blast-1); i>=0; i--) {
      if (Path_Level[Lvl].Path_Array[Bidx].Segment_Array[i].head == B) {
         A_incl_B = true;
         Yintsc = i;
         Xidx   = Aidx;
         Xlast  = Alast;
         Yidx   = Bidx;
         Ylast  = Blast;
         break;
      }
   }
}

// Do we need to continue?
if (!(B_incl_A || A_incl_B))
   return 0;


// 2nd Part:
// Find common and banned vertices.
//-----------------------------------------------------------------
// Common vertices are vertices (and arcs) that are common between
// X and Y. The vertex at Yjnct_idx is a "necessary" common vertex
// and doesn't need to be counted again, e.g.:
//    If   X = s-a-b-c-d   Y = s-b-c-d-e   =>   b and c are the
//    common vertices and d is the necessary common vertex (it is
//    counted by Yjnct_idx). Combined Chain = s-a-b-c-d-e
//
// common_vtcs: The number of common vertices between Chains X and
//    Y, except for the necessary vertex.
//-----------------------------------------------------------------

common_vtcs = 0;

for (i=(Yintsc-1); i>=0; i--) {
   j = Xlast-1;

   if (Path_Level[Lvl].Path_Array[Yidx].Segment_Array[i].head ==
       Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].head) {
      common_vtcs++;
      j--;
   }
}

//-----------------------------------------------------------------
// Banned vertices are any vertex of the Combined Chain that we
// know we cannot include in the final chain because there are
// segments that either do not exist or have not been found, e.g.:
//    If X = s-a-b-e, Y = s-x-c-e and we cannot include "b" in the
//    Combined Chain (because arc bx does not exist), b becomes a
//    banned vertex in the combined Chain = s-a-x-c-e:
//                  s a b x c e
//      banned[]= [0 0 1 0 0 0]
//
// Note 1: Banned vertices allow us to preprocess the Chain and
//   avoid having to look for missing arcs once the Chain is being
//   formed in the 3rd Part.
// Note 2: When banned vertices exist, it is necessary to create a
//   copy of the Chain's Arcs for which the banned vertices exist.
// Note 3: In this procedure, the banned vertices only belong to
//   Chain X.
//
// banned_vtcs: The number of banned vertices.
// banned[]: Array of boolean that tells if the current index is
//    banned or not.
//-----------------------------------------------------------------

bool  * banned = new bool[1+Xlast+Ylast-common_vtcs];

// The first and last vertices cannot be banned in this
// procedure.
banned_vtcs = 0;
banned[0]   = false;
banned[1+Xlast+Ylast-common_vtcs] = false;

for (i=0; i<Xlast; i++) {
   for (j=0; j<=Ylast; j++) {
      // All vertices are not banned by default.
      banned[i+j+1] = false;

      // Elements greater than Yintsc already exist in Chain Y.
      if (j > Yintsc)
         continue;

      // Arc AB should not be banned.
      if ((A == Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i].head) &&
          (B == Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j].head)) {
         continue;
      }
      else {
          // If the following arc does not exist we need to ban
          // the corresponding vertex.
          if (-1 == find_arc_in_db(
                    Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i].head,
                    Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j].head))
          {
             banned[i+j+1] = true;
             banned_vtcs++;
          }
      }
   }
}


// 3rd Part:
// Start creating the Combined Chain.
//-----------------------------------------------------------------
// MP_arc_idx: The index to the last Arc of the Main Part of the
//    Combined Chain.
// MP_vertics: Number of vertices of the Combined Chain.
// NWidx: The index for the Combined Chain.
```

```
MP_arc_idx = Xlast + Ylast - common_vtcs - banned_vtcs;
MP_vertics = MP_arc_idx + 2;
NWidx      = Path_Level[Lvl].Num_of_Paths;

if (!incr_Path_Level_size(Lvl))
   return 0;

Path_Level[Lvl].Path_Array[NWidx].type   = Chain;
Path_Level[Lvl].Path_Array[NWidx].chlng  = MP_vertics;
Path_Level[Lvl].Path_Array[NWidx].length = MP_vertics*(MP_vertics-1)/2;
Path_Level[Lvl].Path_Array[NWidx].plidx  = -1;

Path_Level[Lvl].Path_Array[NWidx].size   = 0;
if (!incr_Segment_Array_size(Lvl, NWidx,
                          Path_Level[Lvl].Path_Array[NWidx].length))
   return 0;


// 4th Part:
// Combine X and Y.
//-------------------------------------------------------------------
//    If   X = s-a-b-c-e   and   Y = s-d-m-e   and   AB = bm
//         Xlast = 3     (0 for a, 1 for b, 2 for c, 3 for "e")
//         Ylast = 2     (0 for d, 1 for m, 2 for "e")
//
//               0  1  2  3  4  5
//                  j
//            s  a  b  c  d  m  e
//      0     s -  X  X  X  Y  Y  Y
//      1     a    -  X  X  ?  ?  X
//      2 i   b       -  X  ?  ?  X
//      3     c          -  ?  ?  X
//      4     d             -  Y  Y
//      5     m                -  Y
//            e                   -
//                          ^  ^---------- This row is:
//                          ^              j < Xlast + Ylast
//                          ^----------- This row is:
//                                       j >= Xlast
//
// There are 3 kind of entries depending of the row:
//    1. The first row (i == 0).
//    2. The rows that belong to X [1, (Xlast-common_vtcs)].
//    3. The rows that belong to Y (not 1 or 2).
//-------------------------------------------------------------------
// x: Index count of the Combined Chain.
// Xz: Index count of Chain X.
// Yz: Index count of Chain Y.

x  = 0;
Xz = Xlast + 1;
Yz = Ylast + 1;
del0 = 0;

// y represents the value when we shift from Chain X to Chain Y.
y = Xlast;

for (i=0; i<=(Xlast+Ylast-common_vtcs); i++) {
   for (j=i; j<=(Xlast+Ylast-common_vtcs); j++) {
      // The first row is processed here.
      if (i == 0) {
         if (banned[j+1])
            continue;

         // Elements of Chain X are copied here.
         if (j < y) {
            if (banned_vtcs) {
               // Create a new instance ot this Arc.
               idx = add_arc_to_Arc(
                     Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].tail,
                     Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].head,
                     Lvl+1, NWidx);
            }
            else {
               // Just adjust the index of the exisitng Arc.
               idx = Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].idx;

               Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;
            }

            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].tail;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].head;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  =
            idx;
            x++;
         }
         // Elements of Chain Y are copied here.
         else {
            idx = Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].idx;

            Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;

            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
            Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].tail;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
            Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].head;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  =
            idx;
            x++;
         }

      }
      // Rows that belong to X are processed here.
      else if (i <= (Xlast - common_vtcs)) {
         // The following arcs do not belong to X. They must
         // either be new (Arc ab) or be individual arc (consult
         // the Arc Database).
         if ((j >= Xlast) && (j < (Xlast + Ylast))) {
            if (banned[i] || banned[j+1])
               continue;

            varA =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i-1].head;
            varB =
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[i+j-y].head;

            if ((A == varA) && (B == varB)) {
               if (data)
                  // We assume that the level of the exsiting
                  // Varc/Chain is already equal to "Lvl+1".
                  idx = b;
               else
                  idx = add_arc_to_Arc(A, B, Lvl+1, NWidx);

               if (idx == -1) {
                  cerr << "ERROR: failed to create combined Chain at\n"
                       << "       create_Chain_X_joins_Y\n"
                       << "       Number: 1\n\n";
                  return false;
               }

               Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;

               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].
               tail = A;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].
               head = B;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].
               idx  = idx;
               x++;
            }
            else {
               idx = find_arc_in_db(varA, varB);

               if (idx == -1) {
                  cerr << "ERROR: failed to create combined Chain at\n"
                       << "       create_Chain_X_joins_Y\n"
                       << "       Number: 2\n\n";
                  return false;
               }

               // If this Arc only exists at Lvl 0 it has been
               // stored as an individual arc, let's delete it
               // but after we have created the Combined Chain.
               if (0 == query_arc_db_for_arc_level(idx, 0)) {
                  del0++;
                  del1 = varA;
                  del2 = varB;
                  del3 = query_arc_db_for_arc_index(idx, 0);
               }

               // Let's create a new instance at the appropiate
               // level.
               idx = add_arc_to_Arc(varA, varB, Lvl+1, NWidx);

               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].
               tail = varA;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].
               head = varB;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].
               idx  = idx;
```

```
               x++;
            }
         }
         // The following Arc belongs to X.
         else {
            if (banned[i] || banned[j+1]) {
               Xz++;
               continue;
            }

            if (banned_vtcs) {
               // Create a new instance ot this Arc so we do not
               // disturb the Chain to which the banned vertices
               // belong.
               idx = add_arc_to_Arc(
                     Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].tail,
                     Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].head,
                     Lvl+1, NWidx);
            }
            else {
               // Just adjust the index of the exisitng Arc.
               idx = Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].idx;

               Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;
            }

            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].tail;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].head;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  =
            idx;
            Xz++;
            x++;
         }
      }
      // Rows that belong to Y are processed here.
      else {
         if (banned[i] || banned[j+1]) {
            Yz++;
            continue;
         }

         idx = Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].idx;

         Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;

         Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
         Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].tail;
         Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
         Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].head;
         Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  =
         idx;
         Yz++;
         x++;
      }
   }
}


// 5th Part:
// Complete Combined Chain and return value.
//-------------------------------------------------------------------
// Return value:
//    1. Delete both Chains (A and B).
//    2. Delete Chain A.
//    3. Delete Chain B.

Path_Level[Lvl].Num_of_Paths++;
delete [] banned;

if (del0 > 0) {
   delete_1_element_from_DS(del1, del2, 0, del3, Arc);

   if (del0 > 1) {
      cerr << "ERROR: There is more than one arc that needs\n"
           << "       to be deleted at procedure:\n"
           << "       create_Chain_X_incld_Y";
   }
}


if (banned_vtcs) {
   if (B_incl_A)
      return 2;
   else
      return 3;
   }
   else
      return 1;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::Chain_X_incld_Y
//-----------------------------------------------------------------
// This function processes 2 Chains and an Arc in order to find if it
// is possible to combine them into a new Chain.
// It is assumed that both chains share the lst vertex and other(s)
// vertex (vertices) that is not (doesn't include) the last vertex.
// We call this an inclusion relationship in which one Chain may be
// included (X) as part of another one (Y), because the last vertex
// of X is an intermediate vertex of Y.
// A later modifictation to this function allows the arc to be a
// Varc or Chain that has been formed before, and that should reside
// in 1 level more than the final combined Chain level (New_lvl+1).
//-----------------------------------------------------------------
// Input:
//    a) The basic level where all Chains reside.
//    b) The index of the A Chain.
//    c) The index of the B Chain.
//    d) The level of abstracted arc ab, or the tail of the new Arc.
//    e) The index of abstracted arc ab, or the head of the new Arc.
//    f) Type of argument that has been passed in (c) and (d). If
//       this argument is true, arc ab is a Varc/Chain and the
//       arguments represent the level and index of the abstracted
//       structure. But if this argument is false, arc ab is a new
//       individual Arc that also needs to be added to the data
//       structure.
// NOTE: It is necessary that the level of the existing abstraced arc
//    is 1 more than the final level after the 2 Chains have been
//    joined (New_lvl+1). If not, this procedure will fail to
//    correctly include arc ab into the combined Chain.
//
// Output:
//    Returns TRUE if it combined the 2 Chains and the Arc
//    successfully, otherwise it returns FALSE.
////////////////////////////////////////////////////////////////////
inline int chain_class::create_Chain_X_incld_Y
   (int Lvl, int Aidx, int Bidx, int a, int b, bool data) {
   bool   A_incl_B=false,
          B_incl_A=false;

   int    i, j, x, y, idx, Xz, Yz,
          A, B,
          varA, varB,
          Alast, Blast,
          Xidx, Xlast,
          Yidx, Ylast,
          Yjnct_idx,
          common_vtcs,
          banned_vtcs,
          NWidx,
          MP_vertics,
          MP_arc_idx,
          del0, del1, del2, del3;


   // 1st Part:
   // Find which Chain may be included into the other.
   //-----------------------------------------------------------------
   // We will label X as the Chain that will be included into Chain
   // Y, e.g.:
   //    If   X = s-a-b-c   Y = s-c-d-e   =>   X incl Y
   //    Combined Chain = s-a-b-c-d-e
   //-----------------------------------------------------------------
   // Alast: The index to the last vertex of Chain A.
   // Blast: The index to the last vertex of Chain B.
   // Yjnct_idx: The index to the segment of Chain Y where it is
   //    joined by Chain X.
   // Xidx(Yidx): The index of Chain X(Y).
   // Xlast(Ylast): The index to the last segment of Chain X(Y).

   // Verify the type of data that has been passed as argument.
   if (data) {
      varA = Path_Level[a].Path_Array[b].length-1;
      A    = Path_Level[a].Path_Array[b].Segment_Array[0].tail;
      B    = Path_Level[a].Path_Array[b].Segment_Array[varA].head;
   }
   else {
      A   = a;
      B   = b;
   }
```

```
   Alast = Path_Level[Lvl].Path_Array[Aidx].chlng-2;
   Blast = Path_Level[Lvl].Path_Array[Bidx].chlng-2;

   for (i=Alast; i>=0; i--) {
      if (Path_Level[Lvl].Path_Array[Aidx].Segment_Array[i].head ==
          Path_Level[Lvl].Path_Array[Bidx].Segment_Array[Blast].head) {
         B_incl_A  = true;
         Yjnct_idx = i;
         Xidx  = Bidx;
         Xlast = Blast;
         Yidx  = Aidx;
         Ylast = Alast;
         break;
      }
   }

   if (!B_incl_A) {
      for (i=Blast; i>=0; i--) {
         if (Path_Level[Lvl].Path_Array[Bidx].Segment_Array[i].head ==
             Path_Level[Lvl].Path_Array[Aidx].Segment_Array[Alast].head) {
            A_incl_B  = true;
            Yjnct_idx = i;
            Xidx  = Aidx;
            Xlast = Alast;
            Yidx  = Bidx;
            Ylast = Blast;
            break;
         }
      }
   }

   // Do we need to continue?
   if (!(B_incl_A || A_incl_B))
      return 0;

/* PoA_lgth = Path_Level[Alvl].Path_Array[Aidx].chlng - (2+common_vtcs);
   PoA_arcs = PoA_lgth * (PoA_lgth-1) / 2;
   PoB_lgth = Path_Level[Blvl].Path_Array[Bidx].chlng - (2+common_vtcs);
   PoB_arcs = PoB_lgth * (PoB_lgth-1) / 2;
   missing_arcs = ((PoA_lgth+PoB_lgth) * (PoA_lgth+PoB_lgth-1) / 2) -
                  (PoA_arcs+PoB_arcs);
*/

   // 2nd Part:
   // Find common and banned vertices.
   //-----------------------------------------------------------------
   // Common vertices are vertices (and arcs) that are common between
   // X and Y. The vertex at Yjnct_idx is a "necessary" common vertex
   // and doesn't need to be counted again, e.g.:
   //    If   X = s-a-b-c-d   Y = s-b-c-d-e   =>   b and c are the
   //    common vertices and d is the necessary common vertex (it is
   //    counted by Yjnct_idx). Combined Chain = s-a-b-c-d-e
   //
   // common_vtcs: The number of common vertices between Chains X and
   //    Y, except for the necessary vertex.
   //-----------------------------------------------------------------

   common_vtcs = 0;

   for (i=(Yjnct_idx-1); i>=0; i--) {
      j = Xlast-1;

      if (Path_Level[Lvl].Path_Array[Yidx].Segment_Array[i].head ==
          Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].head) {
         common_vtcs++;
         j--;
      }
   }

   // If all intermediate vertices are common in Chain X, there is no
   // point to combine this two Chains.
   if (common_vtcs == Xlast)
      return 0;

   //-----------------------------------------------------------------
   // Banned vertices are any vertex of the Combined Chain that we
   // know we cannot include in the final chain, e.g.:
   //    If X = s-a-b-c, Y = s-x-c-d-e and we cannot include x in the
   //    Combined Chain (because arc ax does not exist), x becomes a
   //    banned vertex in the Combined Chain = s-a-b-c-d-e.
   //                  s a b x c d e
   //      banned[]= [0 0 0 1 0 0 0]
   //
   // Note 1: Banned vertices allow us to preprocess the Chain and
   //   avoid having to look for missing arcs once the Chain is being
   //   formed in the 3rd Part.
   // Note 2: When banned vertices exist, it is necessary to create a
   //   copy of the Chain's Arcs for which the banned vertices exist.
   // Note 3: In this procedure, the banned vertices only belong to
   //   Chain Y.
   //
   // banned_vtcs: The number of banned vertices.
   // banned[]: Array of boolean that tells if the current index is
   //    banned or not.
   //-----------------------------------------------------------------

   bool  * banned = new bool[2+Xlast+Ylast-common_vtcs];

   // The first and second vertices cannot be banned in this
   // procedure.
   varA        = 0;
   banned_vtcs = 0;
   banned[0]   = false;
   banned[1]   = false;

   for (i=0; i<Xlast; i++) {
      for (j=0; j<=Ylast; j++) {
         banned[i+j+2] = false;

         if (j == Yjnct_idx)
            continue;

         if ((A == Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i].head) &&
             (B == Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j].head)) {
            continue;
         }
         else {
             if (-1 == find_arc_in_db(
                       Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i].head,
                       Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j].head))
             {
                banned[i+j+2] = true;
                banned_vtcs++;

                // Here "varA" is used to find if the last vertices
                // in Chain Y are just being ignored. In which case
                // there is no point on building a Combined Chain.
                if ((i+j+2) >= (Xlast+Ylast))
                   varA++;
             }
         }
      }
   }

   // If the last vertices in Chain Y are being ignored just exit.
   if (varA >= (Ylast-Yjnct_idx))
      return 0;



   // 3rd Part:
   // Start creating the Combined Chain.
   //-----------------------------------------------------------------
   // MP_arc_idx: The index to the last Arc of the Main Part of the
   //    Combined Chain.
   // MP_vertics: Number of vertices of the Combined Chain.
   // NWidx: The index for the Combined Chain.

   MP_arc_idx = Xlast + Ylast - common_vtcs - banned_vtcs;
   MP_vertics = MP_arc_idx + 2;
   NWidx      = Path_Level[Lvl].Num_of_Paths;

   if (!incr_Path_Level_size(Lvl))
      return 0;

   Path_Level[Lvl].Path_Array[NWidx].type   = Chain;
   Path_Level[Lvl].Path_Array[NWidx].chlng  = MP_vertics;
   Path_Level[Lvl].Path_Array[NWidx].length = MP_vertics*(MP_vertics-1)/2;
   Path_Level[Lvl].Path_Array[NWidx].plidx  = -1;

   Path_Level[Lvl].Path_Array[NWidx].size   = 0;
   if (!incr_Segment_Array_size(Lvl, NWidx,
                                Path_Level[Lvl].Path_Array[NWidx].length))
      return 0;



   // 4th Part:
   // Combine X and Y.
   //-----------------------------------------------------------------
   //    If   X = s-a-b-c-e   and   Y = s-d-e-f
   //         Xlast = 3        (0 for a, 1 for b, 2 for c, 3 for "e")
   //         Yjnct_idx = 1    (0 for d, 1 for "e", 2 for f)
   //
```

```
//                0  1  2  3  4  5
//                   j
//             s  a  b  c  d  e  f
//      0    s -  X  X  X  Y  Y  Y
//      1    a    -  X  X  ?  X  ?
//      2 i  b       -  X  ?  X  ?
//      3    c          -  ?  X  ?
//      4    d             -  Y  Y
//      5    e                -  Y
//           f                   -
//                         ^     ^--------- This row is:
//                         ^              j > Xlast + Yjnct_idx
//                         ^----------- This row is:
//                                      1 + Xlast - Yjnct_idx
//
// There are 3 kind of entries depending of the row:
//    1. The first row (i == 0).
//    2. The rows that belong to X [1, (Xlast-common_vtcs)].
//    3. The rows that belong to Y (not 1 or 2).
//-----------------------------------------------------------------
// x: Index count of the Combined Chain.
// Xz: Index count of Chain X.
// Yz: Index count of Chain Y.

x  = 0;
Xz = Xlast + 1;
Yz = Ylast + 1;
del0 = 0;

for (i=0; i<=(Xlast+Ylast-common_vtcs); i++) {
   for (j=i; j<=(Xlast+Ylast-common_vtcs); j++) {
      // The first row is processed here.
      if (i == 0) {
         if (banned[j+1])
            continue;

         // y represents the value when we shift from Chain X to
         // Chain Y.
         y = Xlast;

         // Elements of Chain X are copied here.
         if (j < y) {
            Path_Level[Lvl+1].Path_Array
               [Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i+j].idx].
               plidx = NWidx;

            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].tail;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].head;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[j].idx;
            x++;
         }
         // Elements of Chain Y are copied here.
         else {
            if (banned_vtcs) {
               // Create a new instance ot this Arc.
               idx = add_arc_to_Arc(
                     Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].tail,
                     Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].head,
                     Lvl+1, NWidx);
            }
            else {
               // Just adjust the index of the exisitng Arc.
               idx = Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].idx;

               Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;
            }

            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
            Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].tail;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
            Path_Level[Lvl].Path_Array[Yidx].Segment_Array[j-y].head;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  = idx;
            x++;
         }

      }
      // Rows that belong to X are processed here.
      else if (i <= (Xlast - common_vtcs)) {
         y = Xlast + Yjnct_idx;

         // The following arcs do not belong to X. They must
         // either be new (Arc ab) or be individual arc (consult
         // the Arc Database).
         if ((j == (1 + Xlast - Yjnct_idx)) || (j > y)) {
            if (banned[i] || banned[j+1])
               continue;

varA = Path_Level[Lvl].Path_Array[Xidx].Segment_Array[i-1].head;
varB = Path_Level[Lvl].Path_Array[Yidx].Segment_Array[i+j-y].head;

            if ((A == varA) && (B == varB)) {
               if (data)
                  // We assume that the level of the exsiting
                  // Varc/Chain is already equal to "Lvl+1".
                  idx = b;
               else
                  idx = add_arc_to_Arc(A, B, Lvl+1, NWidx);

               if (idx == -1) {
                  cerr << "ERROR: failed to create combined Chain at\n"
                       << "       create_Chain_X_incld_Y\n"
                       << "       Number: 1\n\n";
                  return 0;
               }

               Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;

               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail = A;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head = B;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  = idx;
               x++;
            }
            else {
               idx = find_arc_in_db(varA, varB);

               if (idx == -1) {
                  cerr << "ERROR: failed to create combined Chain at\n"
                       << "       create_Chain_X_incld_Y\n"
                       << "       Number: 2\n\n";
                  return false;
               }

               // If this Arc only exists at Lvl 0 it has been
               // stored as an individual arc, let's delete it
               // but after we have create the Combined Chain.
               if (0 == query_arc_db_for_arc_level(idx, 0)) {
                  del0++;
                  del1 = varA;
                  del2 = varB;
                  del3 = query_arc_db_for_arc_index(idx, 0);
               }

               // Let's create a new instance at the appropiate
               // level.
               idx = add_arc_to_Arc(varA, varB, Lvl+1, NWidx);

               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail = varA;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head = varB;
               Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  = idx;
               x++;
            }
         }
         // The following Arc belongs to X.
         else {
            if (banned[i] || banned[j+1]) {
               Xz++;
               continue;
            }

            idx = Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].idx;

            Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;

            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].tail;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].head;
            Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  =
            Path_Level[Lvl].Path_Array[Xidx].Segment_Array[Xz].idx;
            Xz++;
            x++;
         }
      }
      // Rows that belong to Y are processed here.
      else {
         if (banned[i] || banned[j+1]) {
            Yz++;
            continue;
         }

         if (banned_vtcs) {
```

```
            // Create a new instance ot this Arc so we do not
            // disturb the Chain to which the banned vertices
            // belong.
            idx = add_arc_to_Arc(
                  Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].tail,
                  Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].head,
                  Lvl+1, NWidx);
         }
         else {
            // Just adjust the index of the exisitng Arc.
            idx = Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].idx;

            Path_Level[Lvl+1].Path_Array[idx].plidx = NWidx;
         }

         Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].tail =
         Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].tail;
         Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].head =
         Path_Level[Lvl].Path_Array[Yidx].Segment_Array[Yz].head;
         Path_Level[Lvl].Path_Array[NWidx].Segment_Array[x].idx  = idx;
         Yz++;
         x++;
      }
   }
}


// 5th Part:
// Complete Combined Chain and return value.
//-----------------------------------------------------------------
// Return value:
//    1. Delete both Chains (A and B).
//    2. Delete Chain A.
//    3. Delete Chain B.

Path_Level[Lvl].Num_of_Paths++;
delete [] banned;

if (del0 > 0) {
   delete_1_element_from_DS(del1, del2, 0, del3, Arc);

   if (del0 > 1) {
      cerr << "ERROR: There is more than one arc that needs\n"
           << "       to be deleted at procedure:\n"
           << "       create_Chain_X_incld_Y";
   }
}


if (banned_vtcs) {
   if (B_incl_A)
      return 3;
   else
      return 2;
}
else
   return 1;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::sanitize_Arc_Vs_Chain
//-----------------------------------------------------------------
// This function performs the following operations to verify the
// sanity between a new Arc and a Chain:
// 1. Verify that the Arc does not create a 2-Arc cycle when added to
//    the Chain.
//
//-----------------------------------------------------------------
// Input:
//    a) The tail of the new Arc.
//    b) The head of the new Arc.
//    c) The level of the Chain.
//    d) The index of the Chain.
// Output:
//    Returns true if the Chain and the Arc can be combined
//    successfully, otherwise it returns false.
////////////////////////////////////////////////////////////////////
inline bool chain_class::sanitize_Arc_Vs_Chain
   (int a, int b, int lvl, int idx) {
   int    i;

   // Find 2-Arc cycles between the Chain and the Arc ab.
   for (i=0; i<Path_Level[lvl].Path_Array[idx].length; i++) {
      if ((a == Path_Level[lvl].Path_Array[idx].Segment_Array[i].head) &&
          (b == Path_Level[lvl].Path_Array[idx].Segment_Array[i].tail)) {
         return false;
      }
   }

   // We didn't find anything suspicious, the Arc and the Chain are
   // fine.
   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::sanitize_Chain_Vs_Chain
//-----------------------------------------------------------------
// This function performs the following operations to verify the
// sanity between 2 combined Chains:
// 1. Verify that any segment of one of the Chains, which is a Varc
//    or a Chain at the next level, does not contain one of the
//    vertices of the other Chain. If we ignore this rule we may
//    obtain the following scenario:
//       Chain A: 1-2-3
//       Chain B: 1-4-5-3
//       segments 4-2 and 5-2 exist, but segment 1-4 is
//       Varc: 1-2-4
//       Combined Chain: 1-4-5-2-3 with segment 1-4 = Varc(1-2-4)!!!!
//
//-----------------------------------------------------------------
// Input:
//    a) The level of the 1st Chain (A).
//    b) The index of the 1st Chain (A).
//    c) The level of the 2nd Chain (B).
//    d) The index of the 2nd Chain (B).
// Output:
//    Returns true if the Chain and the Arc can be combined
//    successfully, otherwise it returns false.
////////////////////////////////////////////////////////////////////
inline bool chain_class::sanitize_Chain_Vs_Chain
   (int Alvl, int Aidx, int Blvl, int Bidx) {
   int    i, j, k,
          Xidx;

   // Find if any of the segments of Chain A contain a Varc/Chain
   // that already hides any of the vertices of Chain B.
   for (i=0; i<Path_Level[Alvl].Path_Array[Aidx].length; i++) {
      Xidx = Path_Level[Alvl].Path_Array[Aidx].Segment_Array[i].idx;

      // Is this segment a Chain or a Varc?
      switch (Path_Level[Alvl+1].Path_Array[Xidx].type) {
         case Chain: {
         // If A is a Chain, compare the intermediate vertices of
         // this Chain with the intermediate vertices of Chain B.
         for (j=0; j<(Path_Level[Alvl+1].Path_Array[Xidx].chlng-2); j++) {
           for (k=0; k<(Path_Level[Blvl].Path_Array[Bidx].chlng-2); k++) {
               if (Path_Level[Alvl+1].Path_Array[Xidx].Segment_Array[j].head
                  == Path_Level[Blvl].Path_Array[Bidx].Segment_Array[k].head)
                  return false;
               }
            }
            break;
         }
         case Varc: {
         // If A is a Varc, compare the intermediate vertices of
         // this Varc with the intermediate vertcies of Chain B.
         for (j=1; j<Path_Level[Alvl+1].Path_Array[Xidx].length; j++) {
            for (k=0; k<(Path_Level[Blvl].Path_Array[Bidx].chlng-2); k++) {
               if (Path_Level[Alvl+1].Path_Array[Xidx].Segment_Array[j].tail
                  == Path_Level[Blvl].Path_Array[Bidx].Segment_Array[k].head)
                  return false;
               }
            }
         }
      }
   }

   // Find if any of the segments of Chain B contain a Varc/Chain
   // that already hides any of the vertices of Chain A.
   for (i=0; i<Path_Level[Blvl].Path_Array[Bidx].length; i++) {
      Xidx = Path_Level[Blvl].Path_Array[Bidx].Segment_Array[i].idx;

      // Is this segment a Chain or a Varc?
      switch (Path_Level[Blvl+1].Path_Array[Xidx].type) {
         case Chain: {
         // If B is a Chain, compare the intermediate vertices of
         // this Chain with the intermediate vertices of Chain A.
         for (j=0; j<(Path_Level[Blvl+1].Path_Array[Xidx].chlng-2); j++) {
            for (k=0; k<(Path_Level[Alvl].Path_Array[Aidx].chlng-2); k++) {
               if (Path_Level[Blvl+1].Path_Array[Xidx].Segment_Array[j].head
```

```
                  == Path_Level[Alvl].Path_Array[Aidx].Segment_Array[k].head)
                  return false;
               }
            }
            break;
         }
         case Varc: {
         // If B is a Varc, compare the intermediate vertices of
         // this Varc with the intermediate vertcies of Chain A.
         for (j=1; j<Path_Level[Blvl+1].Path_Array[Xidx].length; j++) {
            for (k=0; k<(Path_Level[Alvl].Path_Array[Aidx].chlng-2); k++) {
               if (Path_Level[Blvl+1].Path_Array[Xidx].Segment_Array[j].tail
                  == Path_Level[Alvl].Path_Array[Aidx].Segment_Array[k].head)
                  return false;
               }
            }
         }
      }
   }

   // We didn't find anything suspicious, the Chains are fine.
   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::remove_Chain
//--------------------------------------------------------------------
// Deletes the "remainder" parts of a Chain that has been used to
// form a Combined Chain.
// NOTE: This procedure verifies, before removing any arc, that the
// segments of an unused Chain are not being used by other Chains.
//--------------------------------------------------------------------
// Input:
//    a) The level where the Chain is.
//    b) The index where the Chain is.
// Output:
//    Returns TRUE if the operation was succesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::remove_Chain(int lvl, int idx) {
   int   i, idx_l;
         //tail, head;
   //linktype   type;

   // Delete the arcs that are just hanging at the next level of
   // abstraction.
   for (i=(Path_Level[lvl].Path_Array[idx].length-1); i>=0 ; i--) {
      idx_l = Path_Level[lvl].Path_Array[idx].Segment_Array[i].idx;

      if (idx == Path_Level[lvl+1].Path_Array[idx_l].plidx) {
         if(!remove_elements_from_DS(lvl+1, idx_l))
            cerr << " ERROR: Could not completelly delete segment at\n"
                 << "        level: " << (lvl+1) << ", " << idx_l << "\n"
                 << "        from Chain: " << lvl << ", " << idx << ".\n\n";
      }
   }

   // Now delete the Chain at the current level.
   return delete_1_element_from_DS(0, 0, lvl, idx, Chain);
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::join_Chains
//--------------------------------------------------------------------
// This function processes 2 Chains and an Arc in order to find if it
// is possible to combine them into a new Chain. If a combination is
// possible, it calls the appropriate function to perform this
// operation.
// This function also removes the Chains that need to be discarded
// after performing a combination.
//--------------------------------------------------------------------
// Input:
//    a) The level of the A Chain.
//    b) The index of the A Chain.
//    c) The level of the B Chain.
//    d) The index of the B Chain.
//    c) The level of abstracted arc ab, or the tail of the new Arc.
//    d) The index of abstracted arc ab, or the head of the new Arc.
//    e) Type of argument that has been passed in (c) and (d). If
//       this argument is true, arc ab is a Varc/Chain and the
//       arguments represent the level and index of the abstracted
//       structure. But if this argument is false, arc ab is a new
//       individual Arc that also needs to be added to the data
//       structure.
// Output:
//    Returns TRUE if it combined the 2 Chains and the Arc
//    successfully, otherwise it returns FALSE.
////////////////////////////////////////////////////////////////////
inline bool chain_class::join_Chains
   (int Alvl, int Aidx, int Blvl, int Bidx, int a, int b, bool data) {
   bool   tst2, tst3;
   int    tst1,
          last,
          copy,
          Level,
          Vtx_A, Vtx_B,
          Alast, Blast;


   // For now, we only process Chains that have a common tail.
   if (Path_Level[Alvl].Path_Array[Aidx].Segment_Array[0].tail ==
       Path_Level[Blvl].Path_Array[Bidx].Segment_Array[0].tail) {
      if (data) {
         last  = Path_Level[a].Path_Array[b].length-1;
         Vtx_A = Path_Level[a].Path_Array[b].Segment_Array[0].tail;
         Vtx_B = Path_Level[a].Path_Array[b].Segment_Array[last].head;
      }
      else {
         Vtx_A = a;
         Vtx_B = b;
      }

      // Verify sanity between Chain A and arc ab.
      if (!sanitize_Arc_Vs_Chain(Vtx_A, Vtx_B, Alvl, Aidx))
         return false;

      // Verify sanity between Chain B and arc ab.
      if (!sanitize_Arc_Vs_Chain(Vtx_A, Vtx_B, Blvl, Bidx))
         return false;

      // Verify sanity between Chains.
      if (!sanitize_Chain_Vs_Chain(Alvl, Aidx, Blvl, Bidx))
         return false;

      // If either one of the Chains is at a higher level than the
      // other we need to create a copy so both chains are at the
      // same lower level.
      if (Alvl > Blvl) {
         Aidx  = copy_Varc_or_Chain_to_DS(Alvl, Aidx, Blvl);
         Level = Blvl;
         copy  = 1;
      }
      else if (Blvl > Alvl) {
         Bidx  = copy_Varc_or_Chain_to_DS(Blvl, Bidx, Alvl);
         Level = Alvl;
         copy  = 2;
      }
      else {
         // Since the level of both Chains is the same we can use any
         // as the default level value.
         Level = Alvl;
         copy  = 0;
      }

      // Alast & Blast: The index to the last Arc of the Main Part of
      //    either the A Chain or the B Chain = vertex number - 2.
      Alast = Path_Level[Level].Path_Array[Aidx].length-1;
      Blast = Path_Level[Level].Path_Array[Bidx].length-1;

      // Find if the head of the Chains are the same, and use the
      // correct type of function to create the Combined Chain.
      if (Path_Level[Level].Path_Array[Aidx].Segment_Array[Alast].head ==
          Path_Level[Level].Path_Array[Bidx].Segment_Array[Blast].head) {
         // We have 2 Chains that share tail and head and are joined
         // by an Arc. An example of these Chains is: A-B-C and A-D-C
         // joined by an Arc BD.
         tst1 = create_Chain_X_joins_Y(Level, Aidx, Bidx, a, b, data);
      }
      else {
         // We have 2 Chains where one may be included into another
         // one. An example of these Chains is: A-B-C and A-D-B
         // plus an Arc DC.
         tst1 = create_Chain_X_incld_Y(Level, Aidx, Bidx, a, b, data);
      }

      // Once we have created the Combined Chain, it is time to check
      // if we need to delete either of the old Chains from the data
      // structure.
      switch (tst1) {
         case 0: {
            switch (copy) {
```

```
               case 1:
                  remove_elements_from_DS(Level, Aidx);
                  break;
               case 2:
                  remove_elements_from_DS(Level, Bidx);
                  break;
            }

            return false;
         }
         case 1: {
            // Assuming both Chains are at the same level, we need to
            // remove first the one with the highest index.
            if (Aidx > Bidx) {
               tst2 = remove_Chain(Level, Aidx);
               tst3 = remove_Chain(Level, Bidx);
            }
            else {
               tst3 = remove_Chain(Level, Bidx);
               tst2 = remove_Chain(Level, Aidx);
            }

            if (tst2 && tst3)
               return true;
            else
               return false;
         }
         case 2: {
            if (remove_Chain(Level, Aidx))
               return true;
            else
               return false;
         }
         case 3: {
            if (remove_Chain(Level, Bidx))
               return true;
            else
               return false;
         }
         defalut: {
            cerr << " ERROR: Invalid return value at\n"
                 << "        join_Chains\n\n";
            return false;
         }
      }
   }
   else {
      // If we hit this line, the tails are NOT equal.
      return false;
   }

   // Bogus return (not used) to eliminate warning message from
   // compiler.
   return false;
}


////////////////////////////////////////////////////////////////////////
//
// Function: chain_class::create_series_of_Chains
//----------------------------------------------------------------------
// This function is used when it is necessary to create a "series of
// Chains". This condition happens when there is more than posibility
// of grouping Arcs into the Varcs that will form the segments of a
// Chain.
// This procedure may create a series of similar Chains that could be
// used as different solutions to create longer Chains later in the
// execution of this program.
// An example of what a series of Chains and how it is formed is
// shown bellow:
//
//    array_A:  3,2,1,0  (reverse order)
//    array_B:  4,5,6,0  (reverse order)
//    implicit arc AB: 3-4
//
//    Series of Chains     Segment 1      Segment 2      Segment 3
//       series 1          0-1            0-6-5-4        1-2-3-4
//       series 2          0-1-2          0-6-5-4        2-3-4
//       series 3          0-1-2-3        0-6-5-4        3-4
//
// Note that the path described by "array_B" stays the same. The only
// variations are experienced by "array_A". This means that "array_B"
// needs to be carefully determined before calling this procedure.
//----------------------------------------------------------------------
// Input:
//    a) The level of the B Chain.
//    b) The index of the B Chain.
//    c) The last index to (e).
//    d) An array that contains all the vertices of "path_A" in
//       reverse order.
//    e) The last index to (g).
//    f) An array that contains all the vertices of "path_B" in
//       reverse order.
// Output:
//    Returns TRUE if the series of Chains or a Chain was created
//    successfully, otherwise it returns FALSE.
////////////////////////////////////////////////////////////////////////
inline bool chain_class::create_series_of_Chains
   (int Blvl, int Bidx, int last_A, int *array_A, int last_B, int *array_B) {
   bool   start, end;
   int    i, j,
          D_lvl,              // The level where the Varcs should be.
          T_lvl,
          Atail, Ahead, Asize,
          Btail, Bhead, Bsize,
          var_1, var_2, var_3,
          index1, index2, index3,
          plidx=-1;


   // path_A: Array that will be used to temporarly store vertices
   //    that form the Varcs (segments) of Chains. The vertices in
   //    this array will change according on how they are grouped for
   //    each variation of a Chain.
   // path_B: Array that will be used to store the vertices of
   //    "array_B". In opositon to "path_A", this array will not
   //    change once it has been setup in Part I.
   int *path_A = new int[last_A+1];
   int *path_B = new int[last_B+2];

   if ((!path_A) || (!path_B)) {
      cerr << "ERROR: Could not allocate memory to create a\n"
           << "       dynamic array at procedure:\n "
           << "       create_series_of_Chains\n\n";
      return false;
   }


   // Part I: Array B
   // ---------------------------------------------------------------
   // Sort out the arcs in "array_B". This array is prepared using
   // "path_B" and is created later inside the loop started at
   // Part II.
   // ---------------------------------------------------------------
   Btail = array_A[last_A];
   Bhead = array_B[0];
   Bsize = 0;

   if (Blvl == 0) {
      D_lvl = 1;

      // If "Blvl" is zero, the last segment of the Varc is an
      // individual Arc. Prepare "path_array" to create a Varc.
      for (j=last_B; j>=0; j--)
         path_B[j] = array_B[Bsize++];

      // Delete the Arc at Level 0, it will be added later by the
      // Varc.
      delete_1_element_from_DS(array_B[1], Bhead, Blvl, Bidx, Arc);
   }
   else {
      // If the level of the Arc is greater than zero, then this
      // segment may have been abstracted before. Prepare
      // "path_array" to create a Varc.
      var_1 = Blvl-1;
      var_2 = Path_Level[Blvl].Path_Array[Bidx].plidx;


      switch (Path_Level[var_1].Path_Array[var_2].type) {
         case Varc: {
            // This should be a Varc that already forms part of a
            // Chain. Var_1 should be greater than zero.
            if (var_1 > 0)
               D_lvl = var_1;
            else
               D_lvl = 1;

            // The last segment of the Varc is an individual Arc that
            // has been included in a Varc. Prepare "path_array" to
            // create a Varc.
            for (j=last_B; j>=0; j--)
               path_B[j] = array_B[Bsize++];

/*            start = false;
```

```
   end   = false;

   // Try to duplicate the segments of Varc that are
   // included between "Btail" and "Bhead".
   for (j=0; j<Path_Level[var_1].Path_Array[var_2].length; j++) {
      // Find "Btail".
      if (Btail ==
          Path_Level[var_1].Path_Array[var_2].Segment_Array[j].tail)
         start = true;

      // If the tail has been found copy each vertex of the
      // Varc into "path_array".
      if (start)
         path_B[Bsize++] = Path_Level[var_1].Path_Array[var_2].
                           Segment_Array[j].tail;

      // Find "Bhead".
      if (Bhead ==
          Path_Level[var_1].Path_Array[var_2].Segment_Array[j].head)
         end = true;

      // If the head has been found copy the last vertex of
      // the Varc into "path_array" and exit this loop.
      if (end) {
         path_B[Bsize++] = Path_Level[var_1].Path_Array[var_2].
                           Segment_Array[j].head;
         break;
      }
   }

   // Verify that the Varc was constructed.
   if (!(start && end)) {
      cerr << "ERROR: Could not succesfully construct Varc: "
           << Btail << "-" << Bhead << "\n"
           << "       at procedure:\n"
           << "       create_series_of_Chains\n\n";
   }   */
   break;
}

case Chain: {
   // The last segment of the Varc is an individual Arc that
   // has been included in a Chain. Prepare "path_array" to
   // create a Varc.
   for (j=last_B; j>=0; j--)
      path_B[j] = array_B[Bsize++];

   // If "array_A" only contains one single arc, it is
   // possible to substitute an arc of this Chain for the
   // new Chain that we are building at the end of this
   // procedure. If not, it is better to build individual
   // Chains at Level 0.
   if (last_A <= 1) {
      // The following series of Chains will replace an
      // existing Arc. Hence D_lvl equals to Blvl+1.
      D_lvl = Blvl+1;

      // Remove the last segment of the Chain at the current
      // level.
      // NOTE: the value of "index1" in the following lines
      // is temporary.
      var_1 = path_B[Bsize-2];
      var_2 = find_arc_in_db(var_1, Bhead);

      if (var_2 != -1) {
         var_3 = query_arc_db_for_loc_count(var_2);

         for (j=0; j<var_3; j++) {
            if ((D_lvl-1) == query_arc_db_for_arc_level(var_2, j)) {
               index1 = query_arc_db_for_arc_index(var_2, j);

               plidx  = Path_Level[D_lvl-1].Path_Array[index1].plidx;

               delete_1_element_from_DS(var_1, Bhead, D_lvl-1, index1,
                                        Arc);
               break;
            }
         }
      }
   }
   else
      D_lvl = 1;

   break;
}

case invalid: {
      cerr << "ERROR: Unexpected structure type at procedure:\n"
           << "       create_series_of_Chains\n\n";
      return false;
   }
}



// Part II: Array A
// ----------------------------------------------------------------
// Sort out the arcs in "array_A". The segments in this array are
// divided in 2 sections. The 1st section always contains the 1st
// segment and will sequentially contain the other segments as the
// for loop progresses. The 2nd section always contains the last
// segment and will sequentially pass the other segments as the
// for loop progresses.
// This is an example of how does the segments move from the 1st
// section to the 2nd section.
//    1st execution:     1st section         2nd section
//                       [n-m]               [m-o][o-p][p-q]
//    2nd execution:     1st section         2nd section
//                       [n-m][m-o]          [o-p][p-q]
//    Last execution:    1st section         2nd section
//                       [n-m][m-o][o-p]     [p-q]
// ----------------------------------------------------------------
for (i=0; i<last_A; i++) {
   // 1st section:
   // ------------------------------------------------------------
   // "index1" is always set at this section.
   if (i == 0) {
      // This is the 1st execution of the 1st section. Define the
      // 1st segment.
      Ahead = array_A[last_A-1];
      Atail = array_A[last_A];

      // Find its location at the current level.
      var_1  = find_arc_in_db(Atail, Ahead);
      var_2  = query_arc_db_for_loc_count(var_1);
      index1 = -1;

      for (j=0; j<var_2; j++) {
         var_3 = query_arc_db_for_arc_level(var_1, j);

         if ((D_lvl-1) == var_3) {
            index1 = query_arc_db_for_arc_index(var_1, j);
            Nested = 0;
            index1 = move_arc_to_next_level(Atail, Ahead, D_lvl-1, index1);
            break;
         }

         // If we could not find an appropriate location at this
         // level, it is possible that arc A only exists at other
         // levels. Create a new instance for arc A.
         if (j == (var_2-1)) {
            // If the ONLY instance of an arc is at level 0, it
            // is only a single Arc. Remove it before adding the
            // new instance at "D_lvl" level.
            if ((var_2 == 1) && (var_3 == 0)) {
               index1 = query_arc_db_for_arc_index(var_1, j);

               delete_1_element_from_DS(Atail, Ahead, var_3, index1, Arc);
            }

            index1 = add_arc_to_Arc(Atail, Ahead, D_lvl);
         }
      }
   }
   else {
      // This is NOT the 1st execution of the 1st section. Define
      // a Varc as the 1st segment.
      // The order in which we add data to "path_array" is the
      // inverse of "array_A".
      for (Asize=0; Asize<=(1+i); Asize++) {
         path_A[Asize] = array_A[last_A-Asize];
      }

      Ahead = path_A[Asize-1];
      Atail = path_A[0];

      // Remove the last segment of the Varc at the current level.
      // NOTE: the value of "index1" in the following lines is
      // temporary.
      var_3 = path_A[Asize-2];
      var_1 = find_arc_in_db(var_3, Ahead);

      if (var_1 != -1) {
```

```
      var_2 = query_arc_db_for_loc_count(var_1);

      for (j=0; j<var_2; j++) {
         T_lvl = query_arc_db_for_arc_level(var_1, j);
         if ((D_lvl-1) >= T_lvl) {
            index1 = query_arc_db_for_arc_index(var_1, j);

            delete_1_element_from_DS(var_3, Ahead, T_lvl, index1, Arc);
            break;
         }
      }
   }


   // Create the Varc that will be added to the data structure
   // This is the correct value for "index1".
   index1 = create_path_Varc(D_lvl, Asize, path_A);
}

// 2nd section:
//-----------------------------------------------------------------
// "index3" is always set at this section.
if (i == (last_A-1)) {
   // This is the last execution of the 2nd section. Add the
   // last segment to the data structure.
   index3 = add_arc_to_Arc(array_A[0], array_B[0], D_lvl);
}
else {
   // This is NOT the last execution of the 2nd section. Define
   // a Varc as the 2nd segment.
   Asize = 0;

   // The order in which we add data to "path_array" is the
   // inverse of "array_A".
   for (j=last_A-(1+i); j>=0; j--) {
      path_A[Asize++] = array_A[j];
   }

   // The last vertex in this Varc is Bhead.
   path_A[Asize++] = Bhead;

   index3 = create_path_Varc(D_lvl, Asize, path_A);
}



// Part III: Create each Chain
// ----------------------------------------------------------------
// Now it is time to create a Chain that combines the 3 Varcs/
// Arcs created in pats I and II.
// This section needs to be executed for each combination
// found in section II.
// ----------------------------------------------------------------

// Create the Varc defined by Array B, but first consider that
// if the Varc has length only 2, it really is just one Arc.
//This Varc was prepared in Part I.
if (Bsize == 2)
   index2 = add_arc_to_Arc(path_B[0], path_B[1],  D_lvl);
else
   index2 = create_path_Varc(D_lvl, Bsize, path_B);


if (Atail != Btail) {
   cerr << "ERROR: Invalid Chain parameters at procedure:\n"
        << "       create_series_of_Chains\n\n";
   return false;
}

var_1 = create_simple_3point_Chain(D_lvl-1, index1, index2, index3,
                                   Atail, Ahead, Bhead, plidx);
if (var_1 == -1)
   return false;


// If a "plidx" was provided, we need to set the index a the
// previous level of abstraction (D_lvl-2).
if (plidx != -1) {
   var_2 = Path_Level[D_lvl-2].Path_Array[plidx].length;

   // Find the segment that needs to be modified.
   for (j=0; j<var_2; j++) {
      if ((Atail ==
           Path_Level[D_lvl-2].Path_Array[plidx].Segment_Array[j].tail)
         &&
          (Bhead ==
           Path_Level[D_lvl-2].Path_Array[plidx].Segment_Array[j].head))
         Path_Level[D_lvl-2].Path_Array[plidx].Segment_Array[j].idx = var_1;
      }
   }

}

delete [] path_A;
delete [] path_B;
return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::process_Chain_with_Varcs
//-----------------------------------------------------------------
// This function processes 2 Chains and an Arc in order to find if it
// is possible to combine them into a new Chain. If a combination is
// possible, it calls the appropriate function to perform this
// operation.
// This function also removes the Chains that need to be discarded
// after performing a combination.
//-----------------------------------------------------------------
// Input:
//    a) The level of the A Chain.
//    b) The index of the A Chain.
//    c) The level of the B Chain.
//    d) The index of the B Chain.
//    c) The tail of the new Arc.
//    d) The head of the new Arc.
// Output:
//    Returns TRUE if it combined the 2 Chains and the Arc
//    successfully, otherwise it returns FALSE.
////////////////////////////////////////////////////////////////////
inline bool chain_class::process_Chain_with_Varcs
   (int A, int B, int Bbs_lvl, int Bbs_idx, int segm_A, int prev_segm_A) {
   bool   delete_arc=false;
   int    i,
          A_db, A_cnt,
          A_level, A_index,
          Abs_lvl, Abs_idx,
          A_tail,
          X_db,
          X_lvl,
          X_idx,
          *Xdata,
          Xpath[3] = {0, A, B};
   linktype   Xsgmt[2] = {Arc, Arc};


   // Find preliminary information of path A.
   A_db  = find_arc_in_db(prev_segm_A, segm_A);
   A_cnt = query_arc_db_for_loc_count(A_db);


   // For each one of the instances of the Arc currently processed at
   // path A, try to build a Chain with Varcs as segments.
   for (i=0; i<A_cnt; i++) {
      // Find the level and index of the current location.
      A_level = query_arc_db_for_arc_level(A_db, i);
      A_index = query_arc_db_for_arc_index(A_db, i);

      // If A's level is zero, it means that the ONLY instance of
      // this segment is a single Arc. We will later return false
      // and try to build a series of Chains.
      if (A_level == 0) {
         if (A_cnt != 1) {
            cerr << "ERROR: Unexpected number of instances of an\n"
                 << "       individual arc at procedure:\n"
                 << "       process_Chain_with_Varcs\n\n";
         }
         continue;
      }
      else {
         // Find to which Chain does Arc A belongs. If Arc B does not
         // belong to a Chain, find to which structure it belongs at
         // Level 0.
         Abs_lvl = A_level-1;
         Abs_idx = Path_Level[A_level].Path_Array[A_index].plidx;

         while ((Chain != Path_Level[Abs_lvl].Path_Array[Abs_idx].type)
                && (Abs_lvl != 0)) {
            Abs_idx = Path_Level[Abs_lvl].Path_Array[Abs_idx].plidx;
            Abs_lvl--;
         }

         // Verify that the obtained structure is not the same as
```

```
      // Path B's structure, which was passed as argument.
      if ((Abs_lvl == Bbs_lvl) && (Abs_idx == Bbs_idx))
         continue;
   }

   // Let's see if it is possible to create a longer Chain using
   // path B:
   //-------------------------------------------------------------
   // First try to find the missing arc of this Chain. The origin
   // should be the same (A_tail) and we have an arc from A to B,
   // so let's find how to create a path (A_tail - A - B).

   A_tail = Path_Level[Abs_lvl].Path_Array[Abs_idx].Segment_Array[0].head;

   // Verify that path (A_tail - A - B) is not circular.
   if (B == A_tail)
      continue;

   // Find if the missing arc exisits as an individual arc.
   X_db = find_arc_in_db(A_tail, A);

   if (X_db == -1) {
      // There is no individual arc that joins Chain A and vertex
      // A, how about a Chain or a Varc.
      Xdata = find_arc_in_data_structure(A_tail, A);

      if (!Xdata)
         continue;
      else {
        // Let's use the Varc/Chain and arc found.
        Xsgmt[0] = Varc;
        //Xsgmt[1] = Arc;   <- This value has been initialized.
      }
   }
   else {
      // Let's use the individual arcs found.
      Xsgmt[0] = Arc;
      //Xsgmt[1] = Arc;   <- This value has been initialized.

      // If the individual arc resides at Level 0, we will need to
      // delete it later so we do not leave lonely arcs at this
      // level.
      if (0 == query_arc_db_for_arc_level(X_db, 0))
         delete_arc = true;
   }

   // Define tha path Varc vertices.
   Xpath[0] = A_tail;
   //Xpath[1] = A;   <- This value has been initialized.
   //Xpath[2] = B;   <- This value has been initialized.

   // Create the path Varc at 1 more than the lowest level.
   if (Bbs_lvl < Abs_lvl)
      X_lvl = Bbs_lvl + 1;
   else
      X_lvl = Abs_lvl + 1;

   X_idx = create_path_Varc(X_lvl, 3, Xpath, Xsgmt);

   if (X_idx == -1)
      continue;


   // Time to see if we can create the Chain.
   if (join_Chains(Abs_lvl, Abs_idx, Bbs_lvl, Bbs_idx, X_lvl, X_idx, true)) {
      // Time to delete the lonely arc at Level 0.
      if (delete_arc) {
         X_idx = query_arc_db_for_arc_index(X_db, 0);

         delete_1_element_from_DS(Xpath[0], Xpath[1], 0, X_idx, Arc);
      }

      return true;
   }
   else {
      delete_arc = false;

      // Undo changes to the data structure. First delete all the
      // Varc segments and then delete the Varc.
      if (!remove_elements_from_DS(X_lvl, X_idx)) {
         cerr << "ERROR: Could not restore the data structure\n"
              << "       to its original state at:\n"
              << "       process_Chain_with_Varcs\n\n";
      }
   }
}
```

```
   // If we reach this line it means that we could not build a
   // suitable Chain with Varcs, try to build a series of Chains by
   // returning false.
   return false;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::process_invalid_src_vertex
//-----------------------------------------------------------------
// When there is not a valid source vertex to build a Chain, this
// procedure looks back at the "previous" vertices in the path array
// that has been passed as an argument and tries to build a Chain
// that may include Varcs or series of Chains.
// This function builds more complex structures than functions:
// - add_arc_to_Chain
//-----------------------------------------------------------------
// Input:
//    a) Vertex A.
//    b) Vertex B.
//    c) The level of the B Chain.
//    d) The index of the B Chain.
//    c) The size of the array provided in (d).
//    d) Pointer to the array that contains the previous vertices.
// Output:
//    Returns TRUE if it successfully created a new Chain,
//    otherwise it returns FALSE.
////////////////////////////////////////////////////////////////////
inline bool chain_class::process_invalid_src_vertex
   (int A, int Blvl, int Bidx, int size, int *prev) {
   bool   start,
          retrn_val=false;
   int    i, B,
          tmpA, tmpB,
          tmpB_lvl, tmpB_idx,
          revrse_idx_A,
          revrse_idx_B,
          B_Abs_lvl, B_Abs_idx,
          B_db,
          *B_src;
   linktype   B_type;


   // The reversed paths store each path that will be used to build
   // a series of chains. The vertices are placed in reversed order,
   // hence the following properties are valid for each reversed
   // path:
   //    array            1st element     last element
   //    revrse_path_A       A            source vertex
   //    revrse_path_B       B            1 vertex before the source vertex
   int  *revrse_path_A = new int[size];
   int  *revrse_path_B = new int[size];
   //int revrse_path_A[7];
   //int revrse_path_B[7];

   if ((!revrse_path_A) || (!revrse_path_B)) {
      cerr << "ERROR: Could not allocate memory to create\n"
           << "       revrse_path_A or revrse_path_B at procedure:\n"
           << "       process_invalid_src_vertex\n\n";

      delete [] revrse_path_A;
      delete [] revrse_path_B;
      return retrn_val;
   }


   // revrse_idx_A and revrse_idx_B: Index to each one of the
   // reversed paths.
   revrse_idx_A = 0;
   revrse_idx_B = 0;

   // Vertex B:
   B = Path_Level[Blvl].Path_Array[Bidx].Segment_Array[0].head;

   // Initialize each one of the reversed paths.
   revrse_path_A[revrse_idx_A]   = A;
   revrse_path_B[revrse_idx_B]   = B;

   // tmpB: Temporary value of vertex B.
   // tmpB_lvl: Level of arc currently under examination at path B.
   // tmpB_idx: Index of arc currently under examination at path B.
   tmpB     = revrse_path_B[revrse_idx_B];
   tmpB_lvl = Blvl;
   tmpB_idx = Bidx;
```

```
// Find to which Chain does Arc B belongs. If Arc B does not
// belong to a Chain, find to which structure it belongs at level
// 0. This values are used if we need to build a series of Chains.
if (Blvl != 0) {
   B_Abs_lvl = Blvl - 1;
   B_Abs_idx = Path_Level[Blvl].Path_Array[Bidx].plidx;

   while ((Chain != Path_Level[B_Abs_lvl].Path_Array[B_Abs_idx].type)
          && (B_Abs_lvl != 0)) {
      B_Abs_idx = Path_Level[B_Abs_lvl].Path_Array[B_Abs_idx].plidx;
      B_Abs_lvl--;
   }
}
else {
   B_Abs_lvl = Blvl;
   B_Abs_idx = Bidx;
}



// While loop used to process vertices at path B.
//-------------------------------------------------------------
while (prev[tmpB] > -1) {
   // Set the previous vertex at path B. But in the 1st iteration
   // use the data provided by "Blvl" and "Bidx".
   if (tmpB == B)
      revrse_path_B[++revrse_idx_B] =
      Path_Level[Blvl].Path_Array[Bidx].Segment_Array[0].tail;
   else
      revrse_path_B[++revrse_idx_B] = prev[tmpB];

   // Initialize values for path A.
   tmpA = A;
   revrse_idx_A = 0;


   // While loop used to process vertices at path A.
   //----------------------------------------------------------
   while (prev[tmpA] > -1) {
      // Set the previous vertex at path A.
      revrse_path_A[++revrse_idx_A] = prev[tmpA];

      // Verify if we have found a valid source type.
      B_src  = find_src_vertex(tmpB_lvl, tmpB_idx, prev[tmpA]);
      B_type = convert_int_to_linktype(B_src[0]);

      switch (B_type) {
         case Arc: {
            // This is the simplest case in which we just need to
            // create a series of Chains to solve this structure.
            retrn_val = create_series_of_Chains(Blvl, Bidx,
                                                revrse_idx_A, revrse_path_A,
                                                revrse_idx_B, revrse_path_B);

            // Clean before exiting this procedure.
            delete [] revrse_path_A;
            delete [] revrse_path_B;
            return retrn_val;
         }

         case Chain: {
            // Is it possible to create a complete Chain with
            // Varcs?
            retrn_val = process_Chain_with_Varcs(A, B,
                                                 B_Abs_lvl, B_Abs_idx,
                                                 tmpA, prev[tmpA]);
            // If not, build a series of Chains.
            if (!retrn_val) {
               // Verify that "revrse_path_B" is complete.
               if (prev[tmpA] != revrse_path_B[revrse_idx_B]) {
                  if ((tmpB_lvl-B_src[1]) > 1) {
                     cerr << "WARNING: Cannot create Varc using structures\n"
                          << "         at different levels:\n"
                          << "         process_invalid_src_vertex\n\n";
                     // Clean before exiting this procedure.
                     delete [] revrse_path_A;
                     delete [] revrse_path_B;
                     return retrn_val;
                  }
                  else
                  // Since "revrse_path_B" is not complete we need
                  // to add the only missing vertex before calling
                  // function "create_series_of_Chains". Here we
                  // take advantage of the transitive relation of
                  // all the vertices in a Chain.
                  revrse_path_B[++revrse_idx_B] = prev[tmpA];
               }

               retrn_val = create_series_of_Chains(Blvl, Bidx,
                                                   revrse_idx_A, revrse_path_A,
                                                   revrse_idx_B, revrse_path_B);
            }

            // Clean before exiting this procedure.
            delete [] revrse_path_A;
            delete [] revrse_path_B;
            return retrn_val;
         }

         case Varc: {
            // Is it possible to create a complete Chain with
            // Varcs?
            retrn_val = process_Chain_with_Varcs(A, B,
                                                 B_Abs_lvl, B_Abs_idx,
                                                 tmpA, prev[tmpA]);
            // If not, build a series of Chains.
            if (!retrn_val) {
               // Verify that "revrse_path_B" is complete.
               if (prev[tmpA] != revrse_path_B[revrse_idx_B]) {
                  // Since "revrse_path_B" is not complete we need
                  // to add the Varc's missing vertices before
                  // calling function "create_series_of_Chains".
                  start = false;

               for (i=(Path_Level[B_src[1]].Path_Array[B_src[2]].length-1);
                    i>=0; i--) {
                     // When the last vertex in "revrse_path_B"
                     // match the Varc/Chain we may start to add
                     // vertices to "revrse_path_B".
                     if (revrse_path_B[revrse_idx_B] ==
                         Path_Level[B_src[1]].Path_Array[B_src[2]].
                         Segment_Array[i].tail) {
                        start = true;
                        continue;
                     }

                     // Add this vertex to "revrse_path_B".
                     if (start)
                        revrse_path_B[++revrse_idx_B] =
                          Path_Level[B_src[1]].Path_Array[B_src[2]].
                          Segment_Array[i].tail;

                     // If we have found the last vertex, exit
                     // this loop.
                     if (prev[tmpA] ==
                         Path_Level[B_src[1]].Path_Array[B_src[2]].
                         Segment_Array[i].tail)
                        break;
                  }
               }

               retrn_val = create_series_of_Chains(Blvl, Bidx,
                                                   revrse_idx_A, revrse_path_A,
                                                   revrse_idx_B, revrse_path_B);
            }

            // Clean before exiting this procedure.
            delete [] revrse_path_A;
            delete [] revrse_path_B;
            return retrn_val;
         }
      }

      // Move to the previous vertex of path A.
      tmpA = prev[tmpA];
   }
   // Ends while loop for path A.
   //-------------------------------------------------------------


   // Move to the previous vertex of path B.
   tmpB = prev[tmpB];

   // Find the level and index for the previous arc.
   B_db     = find_arc_in_db(prev[tmpB], tmpB);
   tmpB_lvl = query_arc_db_for_arc_level(B_db, 0);
   tmpB_idx = query_arc_db_for_arc_index(B_db, 0);

   // This piece of code serves to remind us that ther is more
   // than one possible solution to this problem.
   //if (query_arc_db_for_loc_count(B_db)>1)
   //{
   //   cerr << "WARNING: More than 1 valid arc: " << prev[tmpB]
   //        << "-" << tmpB << "\n\n";
```

```
      //}
   }
   // Ends while loop for path B.
   //-----------------------------------------------------------------


   // Clean before exiting this procedure.
   delete [] revrse_path_A;
   delete [] revrse_path_B;
   return retrn_val;
}



////////////////////////////////////////////////////////////////////
//
// Function: chain_class::add_arc_to_Chain
//-----------------------------------------------------------------
// Add a new arc into the Path_Level[].Path_Array[] data structure.
// Some definitions of the arcs used need to be provided:
// If we originally have arcs V1V2 and V1V3 and the New Arc is V2V3,
// We will try to process the Chain (V1, V2, V3). The arcs will be
// labeled as follows:
//    New Arc: The arc that completes the chain (V2V3 or "ab").
//    Previous Arc: The arc that comes BEFORE the New Arc (V1V2), it
//       will be also labeled as "A arc" or "ca".
//    Predecesor Arc: The arc that shares the FINAL vertex of the
//       chain (V1V3), it will be also labeled as "B arc" or "db".
//-----------------------------------------------------------------
// Input:
//    a) The tail of the New Arc (also the head of the A arc).
//    b) The head of the New Arc (also the head of the B Arc).
//    c) The tail of the A Arc.
//    d) The tail of the B Arc.
// Output:
//    Returns the index of Path_Level[c].Path_Array where the arc now
//    resides, or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
inline int *chain_class::find_alternative_B_path
   (int A, int tA, int B, int tB) {
   int    i, j,
          cnt,
          B_db, B_cnt,
          B_level, B_index,
          *list;

   static int   ret_array[2];


   // Find other Arcs that have B as the last vertex (head).
   list = find_last_vtx_in_db(B);
   cnt  = list[0];

   // Process each one of the Arcs found.
   for (i=1; i<=cnt; i++) {
      // Skip "tB" (the tail provided by default).
      if (list[i] != tB) {

         if (list[i] == tA) {
            // This B tail is equal to A's tail, let's use it.
            B_db  = find_arc_in_db(list[i], B);
            B_cnt = query_arc_db_for_loc_count(B_db);

            ret_array[0] = query_arc_db_for_arc_level(B_db, 0);
            ret_array[1] = query_arc_db_for_arc_index(B_db, 0);

            return ret_array;
         }
/*         else {
            // Although this tail is not equal to A's, we still might
            // be able to use it.

            A_db  = find_arc_in_db(tA, A);
            A_cnt = query_arc_db_for_loc_count(A_db);

            for (j=0; j<A_cnt; j++) {
               // Find the level and index of B arc's current location.
               A_level = query_arc_db_for_arc_level(A_db, j);
               A_index = query_arc_db_for_arc_index(A_db, j);

               A_src  = find_src_vertex(A_level, A_index, list[i]);
               A_type = convert_int_to_linktype(A_src[0]);

               if (A_type != invalid)
                  return list[i];
            }

         } */

      }
   }

   // If we hit this line we could not find an alternative path.
   return NULL;
}


////////////////////////////////////////////////////////////////////
//
// Function: chain_class::add_arc_to_Chain
//-----------------------------------------------------------------
// Add a new arc into the Path_Level[].Path_Array[] data structure.
// Some definitions of the arcs used need to be provided:
// If we originally have arcs V1V2 and V1V3 and the New Arc is V2V3,
// We will try to process the Chain (V1, V2, V3). The arcs will be
// labeled as follows:
//    New Arc: The arc that completes the chain (V2V3 or "ab").
//    Previous Arc: The arc that comes BEFORE the New Arc (V1V2), it
//       will be also labeled as "A arc" or "ca".
//    Predecesor Arc: The arc that shares the FINAL vertex of the
//       chain (V1V3), it will be also labeled as "B arc" or "db".
//-----------------------------------------------------------------
// Input:
//    a) The tail of the New Arc (also the head of the A arc).
//    b) The head of the New Arc (also the head of the B Arc).
//    c) The tail of the A Arc.
//    d) The tail of the B Arc.
// Output:
//    Returns the index of Path_Level[c].Path_Array where the arc now
//    resides, or -1 if the operation has been unsuccesful.
////////////////////////////////////////////////////////////////////
inline bool chain_class::add_arc_to_Chain
   (int A, int B, int size, int *previous) {
   int     i, j, k,
           tA, tB,
           A_db,  B_db,
           A_cnt, B_cnt,
           A_level, A_index,
           B_level, B_index,
           Apr_level, Apr_index,
           Bpr_level, Bpr_index,
           index1, index2, index3,
           tail,
           *B_src;
   linktype   B_type;


//   if ((A==1) && (B==10)) {
//      display_all_values(1);
//   }

   // 1st Step:
   // Find all the occurrences of "A arc" and "B arc" in the Arc
   // Database.
   tB    = previous[B];
   tA    = previous[A];

   if ((tA==B) || (tB==A)) {
      cerr << "WARNING: The following circular arc has been found: "
           << A << "-" << B << "\n\n";

      if (-1 == add_arc_to_Arc(A, B))
         return false;
      else
         return true;
   }

   A_db  = find_arc_in_db(tA, A);
   A_cnt = query_arc_db_for_loc_count(A_db);
   B_db  = find_arc_in_db(tB, B);
   B_cnt = query_arc_db_for_loc_count(B_db);

   if ((!A_cnt) || (!B_cnt)) {
      cerr << "ERROR: Using invalid Arcs at procedure\n"
           << "       add_arc_to_Chain\n\n";
      return false;
   }


   // 2nd Step:
   // Find the originating vertex of this triplet.
   for (j=0; j<B_cnt; j++) {
      // Find the level and index of B arc's current location.
      B_level = query_arc_db_for_arc_level(B_db, j);
```

```
B_index = query_arc_db_for_arc_index(B_db, j);

for (i=0; i<A_cnt; i++) {
   // Find the level and index of A arc's current location.
   A_level = query_arc_db_for_arc_level(A_db, i);
   A_index = query_arc_db_for_arc_index(A_db, i);


   // A is an Arc:
   //-------------------------------------------------------
   // If the level of the A Arc is zero, then it has been
   // stored as an individual arc. This also means that the
   // Reference Level is zero.
   if (A_level == 0) {
      B_src  = find_src_vertex(B_level, B_index, tA);
      B_type = convert_int_to_linktype(B_src[0]);

      switch (B_type) {
         case invalid: {
            B_src = find_alternative_B_path(A, tA, B, tB);

            if (B_src) {
               B_level = B_src[0];
               B_index = B_src[1];
            }

            // A more complex structure may need to be
            // constructed.
            if (process_invalid_src_vertex(A, B_level, B_index,
                                           size, previous))
               return true;
            else {
         cerr << "WARNING: Could not build a complex structure\n"
              << "         using an Arc at procedure:\n"
              << "         add_arc_to_Chain\n\n";

               // If we could not process the invalid source
               // vertex, we must at least add the New Arc to
               // the data structure.
               if (-1 == add_arc_to_Arc(A, B))
                  return false;
               else
                  return true;
            }
         }
         case Arc: {
            // 3 individual Arcs, a match made in heaven!
            // Remember that Arc B must be at Level 0 if it is
            // still only stored as an individual Arc.

            // Reset nesting before calling procedure:
            // move_arc_to_next_level()
            Nested = 0;
            index1 = move_arc_to_next_level(tB, B, B_level, B_index);
            break;
         }
         case Chain: {
            // Arc B belongs to a Chain. Create a new instance
            // of Arc B at Level 1.
            // Remember that the level at which the Chain
            // resides could be different to zero.
            index1 = add_arc_to_Arc(tB, B, A_level+1);
            break;
         }
         case Varc: {
            B_level = B_src[1];
            B_index = B_src[2];
            // Arc B belongs to a Varc. Because Varcs are now
            // either used at specific Chains or do not have
            // extra arcs, we only need to create a new
            // instance of Arc B.
            index1 = get_segment_from_Varc_provide_Varc_info(
                     B_level, B_index, tA, B, A_level+1);
            break;
         }
         default: {
            // We should never meet this condition.
            cerr << "ERROR: Unknown source type at procedure\n"
                 << "       add_arc_to_Chain\n\n";
            return false;
         }
      }


      // Move Arc A from Level 0 to Level 1:
      // Note that the index might have changed when we moved
      // Arc B to the next level.
      A_index = query_arc_db_for_arc_index(A_db, i);

      // Reset nesting before calling procedure:
      // move_arc_to_next_level()
      Nested  = 0;
      index2  = move_arc_to_next_level(tA, A, A_level, A_index);


      // Add the New Arc to Level 1:
      index3 = add_arc_to_Arc(A, B, A_level+1);


      // Create the 3 point Chain and exit this procedure.
      return create_simple_3point_Chain(A_level, index2, index1, index3,
                                        tA, A, B);

   }
   // The level of the A Arc is greater than zero, hence
   // this arc has been abstracted before and it must be part
   // of either a Chain or a Varc.
   else {
      // Find more info at the previous abstracted level.
      Apr_level = A_level-1;
      Apr_index = Path_Level[A_level].Path_Array[A_index].plidx;


      // Switch for A Arc starts here.
      //-------------------------------------------------------
      switch (Path_Level[Apr_level].Path_Array[Apr_index].type) {

      // A is a Chain:
      //-------------------------------------------------------
      // If the abstracted structure is a Chain, we will try to
      // create another Chain.
      case Chain: {
         if (tA != Path_Level[A_level].Path_Array[A_index].
                   Segment_Array[0].tail) {
      cerr << "ERROR: Arc's tail information mismatch at procedure\n"
           << "       add_arc_to_Chain (2)\n\n";
            return false;
         }

         B_src  = find_src_vertex(B_level, B_index, tA);
         B_type = convert_int_to_linktype(B_src[0]);

         switch (B_type) {
         case invalid: {
            B_src = find_alternative_B_path(A, tA, B, tB);

            if (B_src) {
               B_level = B_src[0];
               B_index = B_src[1];
            }

            // A more complex structure may need to be
            // constructed.
            if (process_invalid_src_vertex(A, B_level, B_index,
                                           size, previous))
               return true;
            else {
               cerr << "WARNING: Could not build a complex structure\n"
                    << "         using a Chain at procedure:\n"
                    << "         add_arc_to_Chain\n\n";

               // If we could not process the invalid source
               // vertex, we must at least add the New Arc to
               // the data structure.
               if (-1 == add_arc_to_Arc(A, B))
                  return false;
               else
                  return true;
            }
         }
         case Chain: {
            // Arc B also belongs to a Chain.
            Bpr_level = B_level-1;
            Bpr_index = Path_Level[B_level].Path_Array[B_index].plidx;

            // Is it possible to join these Chains?
            if (join_Chains(Apr_level, Apr_index,
                            Bpr_level, Bpr_index,
                            A, B, false)) {
               return true;
            }
            else {
               // Is this the last chance to match the Chains?
               if (((i+1) == A_cnt) && ((j+1) == B_cnt)) {
```

```
            // Create a new instance of B Arc and add
            // this Chain to the data structure.
            index1 = add_arc_to_Arc(tB, B, A_level);
            break;
         }
         else
            continue;
      }
   }
   case Arc: {
      // Arc B is an individual Arc.
      // Is this the last chance to create a Chain?
      if ((i+1) == A_cnt) {
         // Move Arc B to the next level.
         Nested = 0;
         index1 = move_arc_to_next_level(tB, B, B_level, B_index);
         break;
      }
      else
         continue;
   }
   case Varc: {
      // Arc B belongs to a Varc. Because Varcs are now
      // either used at specific Chains or do not have
      // extra arcs, we only need to create a new
      // instance of Arc B.
      if (((i+1) == A_cnt) && ((j+1) == B_cnt)) {
         B_level = B_src[1];
         B_index = B_src[2];

         index1 = get_segment_from_Varc_provide_Varc_info(
                  B_level, B_index, tA, B, A_level);
         break;
      }
      else
         continue;
   }
   }

   // Add the new location of A Arc.
   index2 = add_arc_to_Arc(tA, A, A_level);

   // Add the New Arc.
   index3 = add_arc_to_Arc(A, B, A_level);



   // Create the Chain and exit this procedure.
   create_simple_3point_Chain(Apr_level, index2, index1,
                                    index3, tA, A, B);
   break;
}

// A is a Varc:
//-------------------------------------------------------
// If the abstracted structure is a Varc, we will try to
// create a new Chain.
case Varc: {
   index1 = -1;

   // We need to find the source vertex of the Varc!!!!
   for (k=(Path_Level[Apr_level].Path_Array[Apr_index].length-1);
        k>=0; k--) {
      tail =
      Path_Level[Apr_level].Path_Array[Apr_index].Segment_Array[k].tail;

      B_src  = find_src_vertex(B_level, B_index, tail);
      B_type = convert_int_to_linktype(B_src[0]);

      switch (B_type) {
         case invalid: {
            // Is this the last opportunity to create a
            // Chain? If it is, continue like a regular
            // Chain.
            if (k != 0)
               break;
         }
         case Chain: {
            B_src = find_alternative_B_path(A, tA, B, tB);

            if (B_src) {
               B_level = B_src[0];
               B_index = B_src[1];
            }

            // A more complex structure may need to be
            // constructed.
            if (process_invalid_src_vertex(A, B_level, B_index,
                                           size, previous))
               return true;
            else {
               cerr << "WARNING: Could not build a complex structure\n"
                    << "         using a Varc at procedure:\n"
                    << "         add_arc_to_Chain\n\n";

               // There is no common source vertex, but
               // still the New Arc needs to be added.
               if (-1 == add_arc_to_Arc(A, B))
                  return false;
               else
                  return true;
            }
         }

         case Arc: {
            // B is an Arc, move it to the same level as
            // A_level.
            switch (B_level-Apr_level) {
               case 0: {
                  // B_level == Apr_level
                  Nested = 0;
                  index1 =
                  move_arc_to_next_level(tB, B, B_level, B_index);
                  break;
               }
               case 1: {
                  // B_level == A_level
                  index1 = B_index;
                  break;
               }
               default: {
                  // B_level > A_level or B_level < Apr_level
                  // Add a new instance of arc B.
                  index1 = add_arc_to_Arc(tB, B, A_level);

                  if (B_level == 0)
                     delete_1_element_from_DS
                     (tB, B, B_level, B_index, Arc);
               }

            }
            break;
         }

         case Varc: {
            B_level = B_src[1];
            B_index = B_src[2];

            index1 = get_segment_from_Varc_provide_Varc_info(
                     B_level, B_index, tail, B, A_level);
            break;
         }
      }

      if (index1 != -1)
         break;
   }


   // The previous move of segment B may have caused part
   // A to shift places.
   index2 = find_last_index_at_specific_level(Apr_level, tail, A);


   // When "Apr_level" is zero we have a Varc standing on
   // its own. Let's create a 3 point Chain at Level 0
   // (Apr_level). Segments reside at "A_level".
   if (Apr_level == 0) {
      // Add the new location of A Arc.
      if (index2 == -1)
         index2 = get_segment_from_Varc_provide_Varc_info(
                  Apr_level, Apr_index, tail, A, A_level);
      else
         index2 = get_segment_from_Varc_provide_Varc_info(
                  Apr_level, index2, tail, A, A_level);

      // The previous move of segment A may have caused part
      // B to shift places.
      index1 = find_last_index_at_specific_level(A_level, tail, B);

      // Add the New Arc.
      index3 = add_arc_to_Arc(A, B, A_level);
```

```
                if (create_simple_3point_Chain(Apr_level, index2, index1,
                                               index3, tail, A, B))
                   return true;
                else
                   return false;
             }
             else
             // When "Apr_level" is greater than zero we have a
             // Varc that is part of another structure (a Chain),
             // let's create a 3 point Chain at "Apr_level-1".
             // Segments reside at "Apr_level". {
                // Add the new location of A Arc.
                if (index2 == -1)
                   index2 = get_segment_from_Varc_provide_Varc_info(
                            Apr_level, Apr_index, tail, A, Apr_level);
                else
                   index2 = get_segment_from_Varc_provide_Varc_info(
                            Apr_level, index2, tail, A, Apr_level);

                // The previous move of segment A may have caused part
                // B to shift places.
                //index1 = find_last_index_at_specific_level(Apr_level, tail, B);
                Nested = 0;
                index1 = move_arc_to_prev_level(tail, B, A_level, index1);

                // Add the New Arc.
                index3 = add_arc_to_Arc(A, B, Apr_level);


                if (create_simple_3point_Chain(Apr_level-1, index2, index1,
                                               index3, tail, A, B))
                   return true;
                else
                   return false;
             }
          }

          // A is invalid:
          //-------------------------------------------------------
          // Generate Error message.
          default: {
             cerr << "ERROR: Invalid path type at procedure\n"
                  << "       add_arc_to_Chain\n\n";
             return false;
          }
          }
          // Switch for Arc A finishes here.
       }
    }
 }

 return true;
}
```

C.3 C++ module: Arcdbclass.cpp

```
#include <iostream>
#include <fstream>
using namespace std;



////////////////////////////////////////////////////////////////////////////
//
//
//          IMPLEMENTATION OF DATA STRUCTURE FOR CHAIN ROUTING
//
//
// This class implements the data structure and class needed to implement the
// Chain Routing algorithm.
//
//--------------------------------------------------------------------------
//
// Created by: David Arjona Villicana
// Created on: 23/Oct/2008
// Last modified on: 10/Nov/2008
//
//--------------------------------------------------------------------------
//
// Comments:
// 1. We later may need to add a function that deletes arcs or records from
//    the database.
//
////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////
//
//   CONSTANTS
//
////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// ARRAY_BLK_1: Size of a block of data for the Arc Database.
//---------------------------------------------------------------
const DB_BLOCK = 100;

/////////////////////////////////////////////////////////////////
// ARRAY_BLK_3: Size of a block for the location array.
// Because we need to add and later delete 1 location every time the
// arc moves, it is convenient to have at least 2 locations per arc.
//---------------------------------------------------------------
const ARC_BLOCK = 2;



////////////////////////////////////////////////////////////////////////////
//
//   STRUCTURES
//
////////////////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////
// loc_tuple: Each one of the recorded locations of an arc.
// ----------------------------------------------------------------
struct loc_tuple {
   int   level,
         index;
};


/////////////////////////////////////////////////////////////////
// arc_tuple: The real individual arcs in the digraph (a record).
// ----------------------------------------------------------------
struct arc_tuple {
   int   tail,
         head,
         loc_size,
         loc_count;
   loc_tuple   *location;
};



////////////////////////////////////////////////////////////////////////////
//
//   CLASS: arc_db_class
//
////////////////////////////////////////////////////////////////////////////
//
// Private functions and variables:
//    db_size: The current size of the block of data. This number should be a
//       multiple of DB_BLOCK.
//    Arc_Count: The number of arcs or records currently stored in this
//       database (db).
//    Arc_Array: Array with information of each arc stored in the db.
//
//    increase_db_size(): Increases the size of the block of data if
//       necessary.
//    increase_loc_size(): Increases the size of the location array for a
//       specific record in the db.
//    decrease_loc_size(): Decreases the size of the location array for a
//       specific record in the db.
//
// Public functions and variables:
//    arc_db_class(): Constructor for this class.
//    ~arc_db_class(): Destructor for this class.
//    display_arc_db(): Displays all the records stored in the db structure.
//    find_arc_in_db(): Overloaded function that allows to find a record in
//       the db.
//    query_arc_db_for_loc_count(): Finds how many locations exist for a
//       specific record.
//    query_arc_db_for_arc_level(): Finds the level of an arc in the db.
```

```
//    query_arc_db_for_arc_index(): Finds the index of an arc in the db.
//    add_new_arc_to_db(): Adds a new arc or record to the db.
//    mod_arc_location(): Modifies a specific loaction entry of 1 arc.
//    add_arc_location(): Adds a new location entry of an existing arc.
//    del_arc_location(): Deletes a specific location entry of 1 arc.
//
////////////////////////////////////////////////////////////////////////////
class arc_db_class {

   int    db_size,
          Arc_Count;

   arc_tuple   *Arc_Array;

   bool increase_db_size();
   bool increase_loc_size(int idx);
   bool decrease_loc_size(int idx);

public:
       arc_db_class();
       ~arc_db_class();

   void display_arc_db(bool print);

   int  find_arc_in_db(int a, int b);
   int  *find_arc_in_db(int a, int b, int c, int d);
   int  *find_last_vtx_in_db(int b);

   int  query_arc_db_for_loc_count(int idx);
   int  query_arc_db_for_arc_level(int i, int j);
   int  query_arc_db_for_arc_index(int i, int j);

   bool add_new_arc_to_db(int a, int b, int c, int d);
   bool mod_arc_location(int a, int b, int c, int d, int e, int f);
   bool add_arc_location(int a, int b, int c, int d);
   bool del_arc_location(int a, int b, int c, int d);

};


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::arc_db_class
//------------------------------------------------------------------
// The constructor for this class.
//////////////////////////////////////////////////////////////////////
inline arc_db_class::arc_db_class() {
   cout << "   Executing constructor for: arc_db_class\n\n";

   // Initialize Ind_Arcs data structure.
   Arc_Count = 0;
   Arc_Array = new arc_tuple[DB_BLOCK];

   if (!Arc_Array) {
      cerr << "ERROR: Could not allocate memory to create\n"
           << "       Arc_Array\n\n";
      db_size = 0;
   }
   else {
      db_size = DB_BLOCK;
   }

}


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::~arc_db_class
//------------------------------------------------------------------
// The destructor for this class.
//////////////////////////////////////////////////////////////////////
inline arc_db_class::~arc_db_class() {
   int   i;

   cout << "   Executing destructor for: arc_db_class\n\n";

   for (i=0; i<Arc_Count; i++) {
      delete [] Arc_Array[i].location;
      Arc_Array[i].location = NULL;
   }

   delete [] Arc_Array;
   Arc_Array = NULL;
}


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::display_arc_db
//------------------------------------------------------------------
// Displays the contents of the Arc database.
//------------------------------------------------------------------
// Input:
//    None.
// Output:
//    None.
//////////////////////////////////////////////////////////////////////
inline void arc_db_class::display_arc_db(bool print=false) {
   int i, j;

if (print) {
   ofstream fileout("ChainRtg.out", ios::app);

   if (!fileout) {
      cerr << "ERROR: Output file (" << fileout << ") could not be opened!\n\n";
      return;
   }

   fileout << " ***********************************************\n";
   fileout << " ***           THE ARC DATABASE              ***\n";
   fileout << " ***********************************************\n\n";
   fileout << "    Number of Arcs: " << Arc_Count << "\n\n";

   for (i=0; i<Arc_Count; i++) {
     fileout << "    Arc " << i << ": "
          << Arc_Array[i].tail << "-"
          << Arc_Array[i].head << "\n";

     for (j=0; j<Arc_Array[i].loc_count; j++)
     {
       fileout << "       Location " << j << ":  "
            << Arc_Array[i].location[j].level << "  "
            << Arc_Array[i].location[j].index << "\n";

     }
   }
   fileout << "\n";

   fileout.close();
}
else {
   cout << " ***********************************************\n";
   cout << " ***           THE ARC DATABASE              ***\n";
   cout << " ***********************************************\n\n";
   cout << "    Number of Arcs: " << Arc_Count << "\n\n";

   for (i=0; i<Arc_Count; i++) {
     cout << "    Arc " << i << ": "
          << Arc_Array[i].tail << "-"
          << Arc_Array[i].head << "\n";

     for (j=0; j<Arc_Array[i].loc_count; j++)
     {
       cout << "       Location " << j << ":  "
            << Arc_Array[i].location[j].level << "  "
            << Arc_Array[i].location[j].index << "\n";

     }
   }
   cout << "\n";
}

}


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::increase_db_size
//------------------------------------------------------------------
// Increases the size of the Arc_Array data structure.
//------------------------------------------------------------------
// Input:
//    None.
// Output:
//    Returns true if the operation has been succesful.
//////////////////////////////////////////////////////////////////////
inline bool arc_db_class::increase_db_size() {
   // Verify if we really need to increment the size of Arc_Array.
   // We increment the size of the Arc_Array in blocks of size
   // DB_BLOCK (originally 100).
   if (Arc_Count >= db_size) {
      int   i,
            k = Arc_Count/DB_BLOCK;
```

```
      // Use a temporary array to store Arc_Array before we increase
      // its size. Then we need to copy the values back.

      cout << "   Increasing size of the Arc Database.\n";

      arc_tuple  * temp_array = new arc_tuple[k*DB_BLOCK];

      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       the Arc Database.  (1)\n\n";
         return false;
      }

      temp_array = Arc_Array;
      Arc_Array  = new arc_tuple[(k+1)*DB_BLOCK];

      if (!Arc_Array) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       the Arc Database.  (2)\n\n";
         return false;
      }
      else {
         db_size = (k+1)*DB_BLOCK;

         // Copy each element of temp_array into the new Arc_Array.
         // Arc_Array
         for (i=0; i<Arc_Count; i++) {
            Arc_Array[i] = temp_array[i];
         }
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::increase_loc_size
//----------------------------------------------------------------
// Increases the size of the Arc_Array[].location data structure.
//----------------------------------------------------------------
// Input:
//    The index into Arc_Array.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////
inline bool arc_db_class::increase_loc_size(int idx) {
   // Verify if we really need to increment the size of location.
   // We increment the size of location in blocks of size
   // ARC_BLOCK (originally 1).
   if (Arc_Array[idx].loc_count >= Arc_Array[idx].loc_size) {
      int   i,
            k = Arc_Array[idx].loc_count/ARC_BLOCK;

      // Use a temporary array to store location before we increase
      // its size. Then we need to copy the values back.

      //cout << "   Increasing size of location for Arc_Array["
      //     << idx << "]\n";

      loc_tuple  * temp_array = new loc_tuple[k*ARC_BLOCK];

      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       Arc_Array[" << idx << "].location  (1)\n\n";
         return false;
      }

      temp_array              = Arc_Array[idx].location;
      Arc_Array[idx].location = new loc_tuple[(k+1)*ARC_BLOCK];

      if (!Arc_Array[idx].location) {
         cerr << "ERROR: Could not allocate memory to increase size of\n"
              << "       Arc_Array[" << idx << "].location  (2)\n\n";
         return false;
      }
      else {
         Arc_Array[idx].loc_size = (k+1)*ARC_BLOCK;

         // Copy each element of temp_array into the new
         // Path_Level[ ].Path_Array
         for (i=0; i<Arc_Array[idx].loc_count; i++) {
            Arc_Array[idx].location[i] = temp_array[i];
         }
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::decrease_loc_size
//----------------------------------------------------------------
// Decreases the size of the Arc_Array[].location data structure.
//----------------------------------------------------------------
// Input:
//    The index into Arc_Array.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////
inline bool arc_db_class::decrease_loc_size(int idx) {
   // Verify if we really need to decrement the size of location.
   // We decrement the size of location in blocks of size
   // ARC_BLOCK (originally 1).
   if ((Arc_Array[idx].loc_count + ARC_BLOCK) < Arc_Array[idx].loc_size) {
      int   i,
            k = (Arc_Array[idx].loc_count/ARC_BLOCK) + 1;

      // Use a temporary array to store location before we decrease
      // its size. Then we need to copy the values back.

      //cout << "   Decreasing size of location for Arc_Array["
      //     << idx << "]\n\n";

      loc_tuple  * temp_array = new loc_tuple[Arc_Array[idx].loc_size];

      if (!temp_array) {
         cerr << "ERROR: Could not allocate memory to decrease size of\n"
              << "       Arc_Array[" << idx << "].location  (1)\n\n";
         return false;
      }

      temp_array              = Arc_Array[idx].location;
      Arc_Array[idx].location = new loc_tuple[k*ARC_BLOCK];

      if (!Arc_Array[idx].location) {
         cerr << "ERROR: Could not allocate memory to decrease size of\n"
              << "       Arc_Array[" << idx << "].location  (2)\n\n";
         return false;
      }
      else {
         Arc_Array[idx].loc_size = k*ARC_BLOCK;

         // Copy each element of temp_array into the new
         // Path_Level[ ].Path_Array
         for (i=0; i<Arc_Array[idx].loc_count; i++) {
            Arc_Array[idx].location[i] = temp_array[i];
         }
      }

      delete [] temp_array;
      temp_array = NULL;
   }

   return true;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::find_arc_in_db   (Overloaded function)
//----------------------------------------------------------------
// Finds an arc into the Arc_Array data structure.
//----------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The current level where the arc resides.
//    d) The current index where the arc resides.
// Output:
//    Returns a pointer to an array that contains the 2 indexes
//    needed to locate the position of arc: The Arc_Array index and
//    the location index. If the arc was not found it returns a NULL
//    pointer.
////////////////////////////////////////////////////////////////////
```

```
inline int *arc_db_class::find_arc_in_db(int a, int b, int c, int d) {
   int   i, j;

   static int   ret_array[2];

   for (i=0; i<Arc_Count; i++) {
      if ((Arc_Array[i].tail == a) &&
          (Arc_Array[i].head == b)) {
         for (j=0; j<Arc_Array[i].loc_count; j++) {
            if ((Arc_Array[i].location[j].level == c) &&
                (Arc_Array[i].location[j].index == d)) {
               ret_array[0] = i;
               ret_array[1] = j;
               return ret_array;
            }
         }
      }
   }

   return NULL;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::find_arc_in_db   (Overloaded function)
//----------------------------------------------------------------
// Finds an arc into the Arc_Array data structure.
//----------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
// Output:
//    Returns the index of Arc_Array where the arc is stored; or -1
//    if the arc could not be found.
////////////////////////////////////////////////////////////////////
inline int arc_db_class::find_arc_in_db(int a, int b) {
   int   i;

   for (i=0; i<Arc_Count; i++) {
      if ((Arc_Array[i].tail == a) && (Arc_Array[i].head == b)) {
         return i;
      }
   }

   return -1;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::find_arc_in_db   (Overloaded function)
//----------------------------------------------------------------
// Finds an arc into the Arc_Array data structure.
//----------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The current level where the arc resides.
//    d) The current index where the arc resides.
// Output:
//    Returns a pointer to an array that contains the 2 indexes
//    needed to locate the position of arc: The Arc_Array index and
//    the location index. If the arc was not found it returns a NULL
//    pointer.
////////////////////////////////////////////////////////////////////
inline int *arc_db_class::find_last_vtx_in_db(int b) {
   int   i;
   static int   ret_array[21];


   ret_array[0] = 0;

   for (i=0; i<Arc_Count; i++) {
      if (Arc_Array[i].head == b) {
         if (ret_array[0] >= 20) {
            cerr << "WARNING: Search for Arcs that have head" << b <<"\n"
                 << "         has found more than 20 valid results.\n\n";
            break;
         }
         else {
            ret_array[0]++;
            ret_array[ret_array[0]] = Arc_Array[i].tail;
         }
      }
   }

   return ret_array;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::query_arc_db_for_loc_count
//----------------------------------------------------------------
// Obtains the loc_count for a specific Arc_Array element.
//----------------------------------------------------------------
// Input:
//    idx) The index into Arc_Array.
// Output:
//    Returns the loc_count for the element requested.
////////////////////////////////////////////////////////////////////
inline int arc_db_class::query_arc_db_for_loc_count(int idx) {
   return Arc_Array[idx].loc_count;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::query_arc_db_for_arc_level
//----------------------------------------------------------------
// Obtains the level value for a specific Arc_Array[].location[]
// element.
//----------------------------------------------------------------
// Input:
//    i) The index into Arc_Array.
//    j) The index into location.
// Output:
//    Returns the level value for the element requested.
////////////////////////////////////////////////////////////////////
inline int arc_db_class::query_arc_db_for_arc_level(int i, int j) {
   return Arc_Array[i].location[j].level;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::query_arc_db_for_arc_index
//----------------------------------------------------------------
// Obtains the index value for a specific Arc_Array[].location[]
// element.
//----------------------------------------------------------------
// Input:
//    i) The index into Arc_Array.
//    j) The index into location.
// Output:
//    Returns the index value for the element requested.
////////////////////////////////////////////////////////////////////
inline int arc_db_class::query_arc_db_for_arc_index(int i, int j) {
   return Arc_Array[i].location[j].index;
}


////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::add_new_arc_to_db
//----------------------------------------------------------------
// Add new arc into the Arc_Array data structure.
//----------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The current level where the arc resides.
//    d) The current index where the arc resides.
// Output:
//    Returns true if the operation has been succesful.
////////////////////////////////////////////////////////////////////
inline bool arc_db_class::add_new_arc_to_db(int a, int b, int c, int d) {
   if (!increase_db_size())
      return false;

   // Let's add the new loc_tuple that holds the location info.
   Arc_Array[Arc_Count].location = new loc_tuple[ARC_BLOCK];

   if (!Arc_Array[Arc_Count].location) {
      cerr << "ERROR: Could not allocate memory to create\n"
           << "       Arc_Array[" << Arc_Count << "].location\n\n";
      return false;
   }
   else {
      Arc_Array[Arc_Count].loc_size = ARC_BLOCK;
   }

   // Let's add the new arc to Arc_Array.
```

```
   Arc_Array[Arc_Count].tail   = a;
   Arc_Array[Arc_Count].head   = b;
   Arc_Array[Arc_Count].loc_count = 1;
   Arc_Array[Arc_Count].location[0].level = c;
   Arc_Array[Arc_Count].location[0].index = d;
   Arc_Count++;

   return true;
}


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::mod_arc_location
//------------------------------------------------------------------
// Modifies the level and index (the location) of an arc present in
// Arc_Array.
//------------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The current level where the arc resides.
//    d) The current index where the arc resides.
//    e) The new level where the arc needs to reside.
//    f) The new index where the arc needs to reside.
// Output:
//    Returns true if the operation has been succesful.
//////////////////////////////////////////////////////////////////////
inline bool arc_db_class::mod_arc_location(int a, int b,
                                           int c, int d,
                                           int e, int f) {
   int   *var;

   // Find where is the arc.
   var = find_arc_in_db(a, b, c, d);

   // If we found it we can modify, if not return false.
   if (var) {
      Arc_Array[var[0]].location[var[1]].level = e;
      Arc_Array[var[0]].location[var[1]].index = f;

      return true;
   }
   else {
      return false;
   }
}


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::add_arc_location
//------------------------------------------------------------------
// Add new level and index data (another location) to an arc already
// stored in Arc_Array.
// Note: This function will fail and return false if the arc does not
// exist in Arc_Array.
//------------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The new level added.
//    d) The new index added.
// Output:
//    Returns true if the operation has been succesful.
//////////////////////////////////////////////////////////////////////
inline bool arc_db_class::add_arc_location(int a, int b, int c, int d) {
   int   j, var;

   // Find where is the arc.
   var = find_arc_in_db(a, b);

   // If we found it we can add a new location, if not return false.
   if (var == -1) {
      return false;
   }
   else {
      if (!increase_loc_size(var))
         return false;

      // Let's add the new location of this arc to Arc_Array.
      j = Arc_Array[var].loc_count;
      Arc_Array[var].location[j].level = c;
      Arc_Array[var].location[j].index = d;
      Arc_Array[var].loc_count++;

      return true;
   }
}


//////////////////////////////////////////////////////////////////////
//
// Function: arc_db_class::del_arc_location
//------------------------------------------------------------------
// Deletes arc locations that are not used anymore and tries to free
// memory.
//------------------------------------------------------------------
// Input:
//    a) The tail of the arc.
//    b) The head of the arc.
//    c) The level deleted.
//    d) The index deleted.
// Output:
//    Returns true if the operation has been succesful.
//////////////////////////////////////////////////////////////////////
inline bool arc_db_class::del_arc_location(int a, int b, int c, int d) {
   int   i, j, *var;

   // Find where is the arc.
   var = find_arc_in_db(a, b, c, d);

   // If we found it we can delete it, if not return false.
   if (var) {
      // If this is the last location, we just delete; if not we will
      // need to move tuples.
      j = Arc_Array[var[0]].loc_count -1;

      if (j == var[1]) {
         Arc_Array[var[0]].location[var[1]].level = -1;
         Arc_Array[var[0]].location[var[1]].index = -1;
         Arc_Array[var[0]].loc_count--;
      }
      else {
         //This is NOT the last location; we will need to move tuples
         //before we can drop the last one.
         for (i=var[1]; i<j; i++) {
            Arc_Array[var[0]].location[i] = Arc_Array[var[0]].location[i+1];
         }

         Arc_Array[var[0]].loc_count--;
      }

      // Let's free memory we do not need to use.
      decrease_loc_size(var[0]);

      return true;
   }
   else {
      return false;
   }
}
```

C.4 C++ module: queue.cpp

```
//////////////////////////////////////////////////////////////////////////
//
//
//                            QUEUE
//
//
// This class implements the queue used by the BFS algorithm.
//
//------------------------------------------------------------------------
//
// Created by: David Arjona Villicana
// Created on: 20/Oct/2008
// Last modified on:
//
//
//////////////////////////////////////////////////////////////////////////




//////////////////////////////////////////////////////////////////////////
//
//   CONSTANTS
//
//////////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////////////
// MAX_NUM_VERTICES: Maximum number of vertices, in the adjacency
//    matrix, that this program supports.
//--------------------------------------------------------------------
const MAX_NU_VERTICES = 100;
```

```
//////////////////////////////////////////////////////////////////////////////
//
//   CLASS: queue
//
//////////////////////////////////////////////////////////////////////////////
//
// Private functions and variables:
//
// Public functions and variables:
//
//////////////////////////////////////////////////////////////////////////////
class queue {

   int start,
       length,
       Qarray[100];

public:
   int head,
       tail;

       queue();
       ~queue();

   int get_head();

   bool put_tail(int x);
};
```

```
//////////////////////////////////////////////////////////////////////
//
// Function: queue::queue
// ------------------------------------------------------------------
//    Constructor for this class.
//////////////////////////////////////////////////////////////////////
inline queue::queue() {
   start  = 0;
   length = 0;
   head   = -1;
   tail   = -1;
}
```

```
//////////////////////////////////////////////////////////////////////
//
// Function: queue::~queue
// ------------------------------------------------------------------
//    Destructor for this class.
//////////////////////////////////////////////////////////////////////
inline queue::~queue() {
   int   i;

   for (i=0; i<length; i++) {
      Qarray[i] = 0;
   }
}
```

```
//////////////////////////////////////////////////////////////////////
//
//   Function: queue::get_head
//------------------------------------------------------------------
// Returns the head of the queue and dicreses its size.
//------------------------------------------------------------------
// Input:
//    None.
// Output:
//    Returns the head of the queue or "-1" if the operation has been
//    unsuccesful.
//------------------------------------------------------------------
inline int queue::get_head() {
   head = Qarray[start];

   start++;
   length--;

   if (length < 0) {
      return -1;
   }

   return head;
}
```

```
//////////////////////////////////////////////////////////////////////
//
//   Function: queue::put_tail
//------------------------------------------------------------------
// The value passed as an argument is added at the tail of the queue.
//------------------------------------------------------------------
// Input:
//    x: The value added to the queue's tail.
// Output:
//    Returns true if the operation has been successful.
//------------------------------------------------------------------
inline bool queue::put_tail(int x) {
   int idx = start+length;

   if (idx >= MAX_NU_VERTICES) {
      return false;
   }

   Qarray[idx] = x;

   length++;

   return true;
}
```