All Pairs Routing Path Enumeration Using Latin Multiplication and Julia

Haochen Sun

ALL PAIRS ROUTING PATH ENUMERATION USING LATIN MULTIPLICATION AND JULIA

by

Haochen Sun

A Dissertation submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Doctor of Philosophy

Milwaukee, Wisconsin

May 2022

ABSTRACT
ALL PAIRS ROUTING PATH ENUMERATION USING LATIN MULTIPLICATION AND JULIA

Haochen Sun

Marquette University, 2022

Enumerating all routing paths among Autonomous Systems (ASes) at an Internet-scale is an intractable problem. The Border Gateway Protocol (BGP) is the standard exterior gateway protocol through which ASes exchange reachability information. Building an efficient path enumeration tool for a given network is an essential step towards estimating the resiliency of the network to cyber security attacks, such as routing origin and path hijacking. In our work, we use the matrix Latin multiplication method to compute all possible paths among all pairs of nodes. We parallelize this computation through the domain decomposition for matrix multiplication and implement our solution in the Julia high performance programming language. We also compare our method with classical Monte Carlo method. Our results provide positive evidence for the applicability of the method.

ACKNOWLEDGMENTS

Haochen Sun

I would first like to thank my academic advisor, Dr. Debbie Perouli, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would also like to thank my father, my mother and my family for being with me every step of the way. You support, guidance, and love have meant the world to me, and you're always pushed me to my best.

To my friends, my committees, my professors , my faculties, and everyone else who has been part of my journey, I really appreciate you more than you know and none of this would be possible without you all. I was blessed to have the opportunity to join Marquette University and through the highs and lows. I will never forget that.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter1
INTRODUCTION

For researchers and network engineers, the network is a fantastic and unique world. It is practically impossible for one to know all connections and routes between Autonomous Systems(ASes), because ASes are administratively independent from each other. The network infrastructure consists of thousands of smaller networks classified as AS. The AS defines a set of connected one or more Internet Protocol (IP) routing prefixes managed by one or more network operators on behalf of an administrative agency or domain with a specific, clearly specified Internet routing policy [32, 8].

The Border Gateway Protocol (BGP) is the standard routing protocol of the Internet used to exchange network reachability information through ASes [51, 58]. The reachability information indicates the traffic between ASes, and the specific AS must transit to reach those networks.

On the Internet, routing announcements are accepted without almost any validation. This opens a possibility for a network operator to announce someone else's network prefixes without permission. Prefix hijacking is a significant threat to the BGP. Resource Public Key Infrastructure (RPKI) provides a tool to secure BGP. Origin validation is a mechanism using RPKI for authenticating route announcements as originating from an intended AS [44]. Path validation is a mechanism used to supplement the origin validation by verifying AS Path of the announcement [12].

Research in BGP security is deemed primarily necessary because BGP is the only available protocol for inter-domain routing[31, 41]. Moreover, despite its role in holding the Internet together, the BGP protocol was not initially designed for

security, making later efforts to secure it relatively hard.

1.1 Motivation

Evaluating the performance characteristics of new routing mechanisms on the Internet requires testing them in realistic simulation setups before deployment in the wild. Since the size of the Internet can significantly impact how novel protocols or enhancements behave, designing a simulator able to model how essential routing aspects scale is critical.

Compared with the other work, most BGP simulators just solve the best routing path enumeration problem, not capturing all possible routing paths within the whole network model. In our work, we attempt to simulate all possible routing paths to solve all routing path enumeration problems. Enumerating all possible routing paths among all nodes can improve the reliability of the network and against the cyber security, specifically for BGP prefix hijacking.

BGP allows policy-based routing. In our work, we aim to solve the the routing path enumeration problem not only at ASes level network, but also a routing path at router level network. Because of the routing policy always performs on the routers not on the ASes.

1.2 Research Goal and Contribution

Building an efficient and reliable BGP simulator for the network model is essential to solving a routing path enumeration problem. In our work, we do not intend to capture the details of the BGP message exchanges. We focus on building a solver to the routing problem that, given network topology and the announced prefixes, it produces all possible paths among all nodes. This type of simulator is proper when considering the resiliency of a network to topology changes and the alternative paths to many destinations, not just the best path. We specifically target modeling the effects of BGP origin and BGP path validation mechanisms [12, 43, 15].

A successful BGP simulator should have a reliable Internet topology model and an efficient routing path enumeration algorithm. In our work, we use a randomly generated topology network model to deploy the BGP simulator and the Latin- Multiplication theory to implement a path enumeration algorithm called the A* algorithm.

This dissertation aims to solve the routing path enumeration problem by providing not only the best but also a ranked list of other candidate paths. Our main contribution is implementing the new BGP simulator through the Latin-Multiplication theory and the Julia programming language.

More specifically, our contributions can be summarized as:

- Enhanced an efficient path enumeration method – Latin Multiplication. And we call it A* algorithm.

  The original Latin Multiplication only produces path connection between two simple path. In our work, we extended it to make it enumerate path between two matrix if and only if matrix is two dimensional and contains elements are paths.

- Design and implementation of a new path solver based on the Latin-Multiplication theory and A* algorithm.

  The new path solver produces not only the best path, but all candidate paths ranked based on the AS's policy preferences. This allows for experiments focusing on the resiliency of the network towards malicious or other unexpected routing events.

  Also, our new path solver produces path not only at AS level network, but also at router level network topology.

- Parallelization of the method using the Julia high-performance programming language and evaluation of different such techniques.

- Design and implementation new Monte Carlo method - All Paris Monte Carlo method.

  In our all pairs Monte Carlo method, we try to estimate all possible path among all nodes for the network. Not like original Monte Carlo method, just estimating path between two nodes.

- Publicly available implementation code.

1.3 Dissertation Outline

The rest of the dissertation is organized as follows.

Firstly, in the background and related work chapter, a description of the Border Gateway Protocol (BGP), BGP Hijacking, and BGP origin is given to help the reader understand the more specific details explained in the following chapters. Also, we provide a short description of Julia programming language to explain why we use it in our project. The different BGP simulator, path simulation method, BRP security and RPKI framework described to present the previous work on which this dissertation was based.

Next, in the Methodology chapter, we describe the general methodology and mathematical principles used to implement the BGP path simulator during this project to provide the available features that should be considered before we fully explain the implementation in later chapters. Moreover, we present the Monte Carlo method for this project to compare our results.

In the System Components chapter, we focus on using the methodology and mathematical principles to implement the BGP path simulator. A full explanation of the implementation process of the BGP path simulator is presented, followed by the technical details. Moreover, a simple example is given to help understand the whole simulation process.

In the Test Cases and Simulation Result chapter, we present all the test cases we used in the experiments and the experiment results. Moreover, the simple data analysis results are shown in this chapter. In this chapter, we also discuss our running environments.

In the Result Discussion chapter, we compared different method on different running environments.

Finally, in the Conclusion chapter, we review the result of our work.

Chapter2
BACKGROUND INFORMATION AND RELATED WORK

The necessary background to understand the proceedings of this project is presented in this chapter. Section 2.1 introduces the general information of Border Gateway Protocol (BGP). BGP Hijacking and BGP Security are presented in Section 2.2. Section 2.3 discuss some features of Julia and why we use Julia. Section 2.4 present related work of our project.

2.1 BGP Overview

Border Gateway Protocol (BGP) is an autonomous system routing protocol that runs on the Transmission Control Protocol(TCP) [58, 34]. BGP is the only protocol used to handle networks as large as the Internet and the only protocol that can adequately handle multiple connections between routing domains.

BGP is based on the experience of the Exterior Gateway Protocol(EGP) [34]. The primary function of the BGP system is to exchange network reachability information with other BGP systems. The network reachability information includes the information of the listed autonomous system (AS). This information effectively constructs the topology diagram of AS interconnection and thus eliminates routing loops, and at the same time, policy decisions can be implemented at the AS level.

BGP is used to exchange routing information between different autonomous systems (AS). When two ASes need to exchange routing information, each AS must designate a node running BGP to exchange routing information on the AS and other AS. This node can be a host, but it is usually a router to perform BGP. Routers that use BGP to exchange information in the two ASes are also called border gateways or border routers.

Figure 2.1 shows how BGP works.

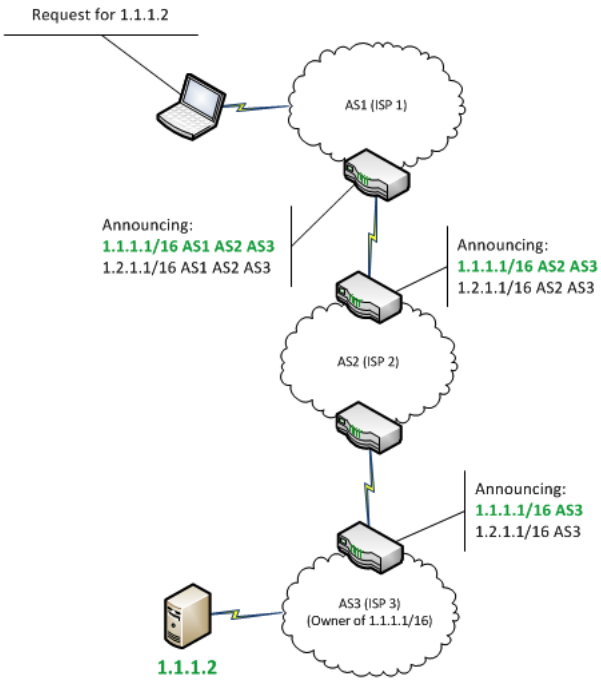


Figure 2.1: BGP Overview

Source from: https://know.bishopfox.com/blog/2015/08/an-overview-of-bgp-hijacking

Since it may be connected to different ASes, there may be multiple border routers running BGP within an AS. BGP running between two or more peer entities in the same autonomous system (AS) is called iBGP (Internal/Interior BGP). BGP running between peer entities belonging to different AS is called eBGP (External/Exterior BGP).

The following Figure 2.2 describe the relationship between iBGP and eBGP.

There are 3 ASes and 5 routers in the figure, we can create typical customer-provider scenario, AS-1 and AS-3 are customers of AS-2. When AS-1 want to reach AS-3, we have to pass through AS-2. The black line in Figure 2.2 represent connection status between routers.

From this customer-provider scenario, we can learn:

- We need eBGP between AS-1 and AS-2 because these are two different AS.

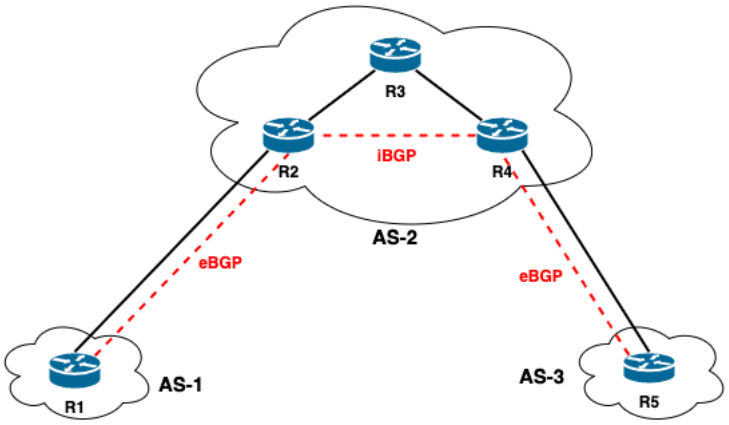


Figure 2.2: iBGP and eBGP

This allows us to advertise a prefix on R1 in BGP so that AS-2 can learn it.

- We also need eBGP between AS-2 and AS-3 so that R5 can learn prefixes through BGP.

- We need to get the prefix that R2 learned from R1 somehow to R5. So, we need configure iBGP between R2 and R4, this allows R4 to advertise it to R5.

BGP is an external or inter-domain routing protocol. The main goal of BGP is to provide routing information communication between routers in different ASes. BGP is neither a vector distance protocol nor a link-state protocol. It is usually called a path vector protocol. BGP, while publishing the reachability to a destination network, contains a list of ASes that must pass through when IP packets reach the destination network. Path vector protocol is beneficial because it's simply looking up the AS number updated by the BGP route can effectively avoid loops. BGP has no restrictions on the network topology, and its characteristics include: [58]

- Path Information. When BGP advertises the reachability information of the destination network, in addition to processing the next hop information of the specified destination network, the advertisement also includes a path vector,

that is, a list of ASs that need to pass through to the destination network. It enables the recipient to understand the access information to the destination network.

- Policy-Based Routing. BGP also allows policy based routing. The network operator can design and implement different policies by programming on it.

- Security Extension. Since the BGP does not contains any security features. It supports security extensions, like the original BGP protocol based on the RPKI framework. It allows the receiver to validate and authenticate the message to verify the identity of the sender.

The router sends a BGP update message to the network, the updated BGP metric is called the path attributes.

- Origin. Origin attribute indicates the origin of the path.

- AS path. This attribute consists of all the different AS numbers from which the the advertisement passed from.

- Local Preference. This attribute is used to choose the exit path for an AS.

The following Figure 2.3 describes the BGP operational algorithm.

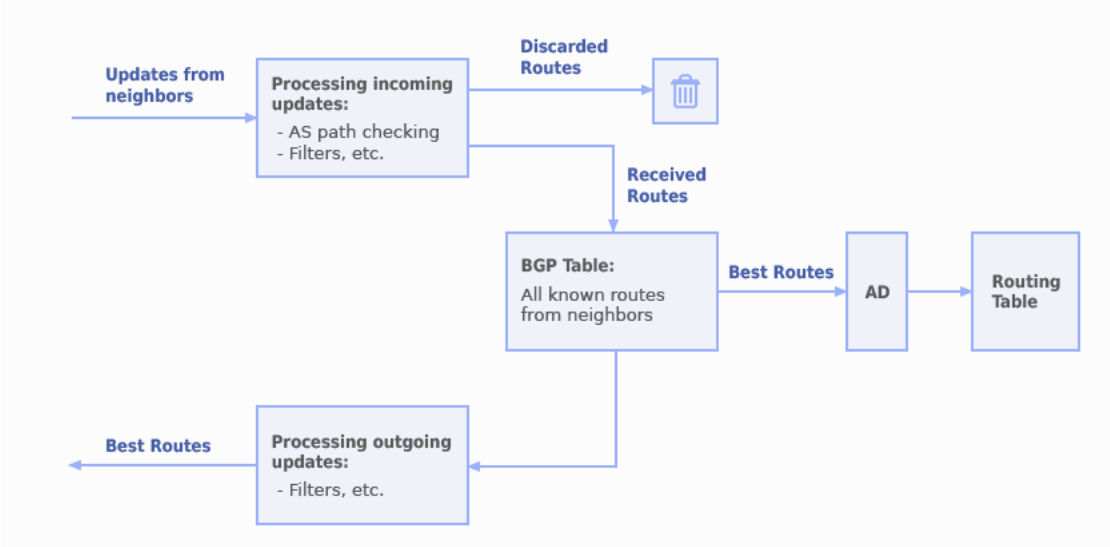


Figure 2.3: BGP Operational Algorithm

Source from: https://www.bgp.us/wp-content/uploads/2016/01/BGP-Border-Gateway-Protocol.png

The neighbor table contains a list of all BGP neighbors, and the BGP operational algorithm learns from all neighbors, including AS path, local preference, and other BGP attributes. BGP Table contains BGP attributes for each path and lists networks received from each neighbor. The BGP table also might have multiple paths to a destination network. Then, the router will advertise the best routes to the routing table. The Routing table list all of the best routes to destination networks.

2.2 BGP Hijacking and BGP Security

BGP Hijacking, sometimes referred to as prefix hijacking or IP hijacking, is an attacker maliciously changing the Internet traffic. Attackers achieve this by mistakenly claiming ownership of IP address groups (called IP prefixes) that they do not actually own or control it [63, 11, 17].

When an AS announces a route with an IP prefix that is not under its control, this announcement, can be propagated and added to the routing table of BGP routers on the Internet. From then until somebody notices and corrects the routes, traffic to

those IPs will be routed to that AS [7].

BGP always prefer the shortest, most specific prefix to the desired IP address. In order to understand the BGP hijacking, Figure 2.4 give a clearly example how BGP hijacking work.

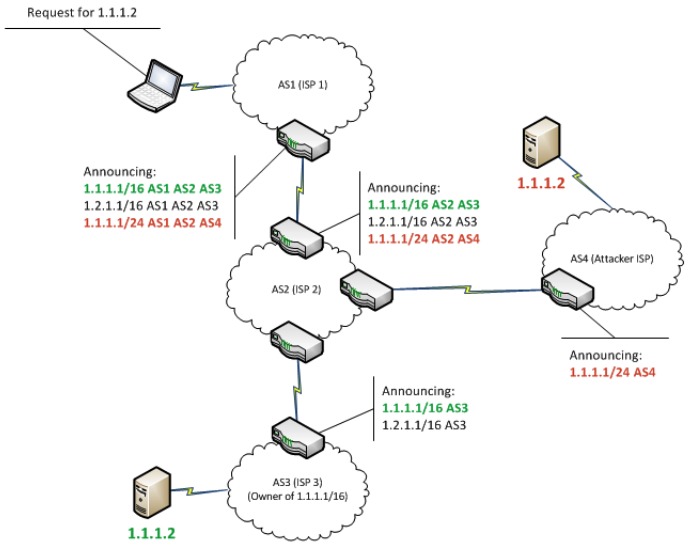


Figure 2.4: BGP Hijacking

Source from: https://know.bishopfox.com/blog/2015/08/an-overview-of-bgp-hijacking

Resource Public Key Infrastructure (RPKI) is a public key infrastructure framework designed to protect the routing protocol, specifically for BGP. RPKI provides a way to connect Internet number resource information (such as IP Addresses) to a trust anchor [3]. RPKI is an encryption method to sign the record with origin IP prefix and AS number. Five regional Internet registry(RIPs) provide method for taking an IP prefix/AS number pair and signing a Route Origin Authorisation(ROA) record to database [39].

The Figure 2.5 shows this encryption process.

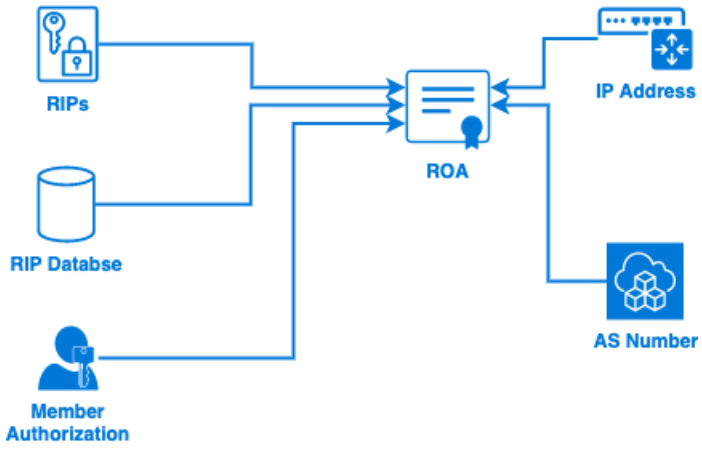
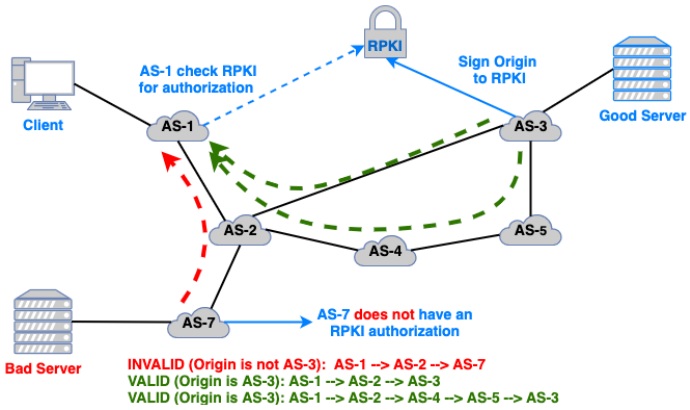


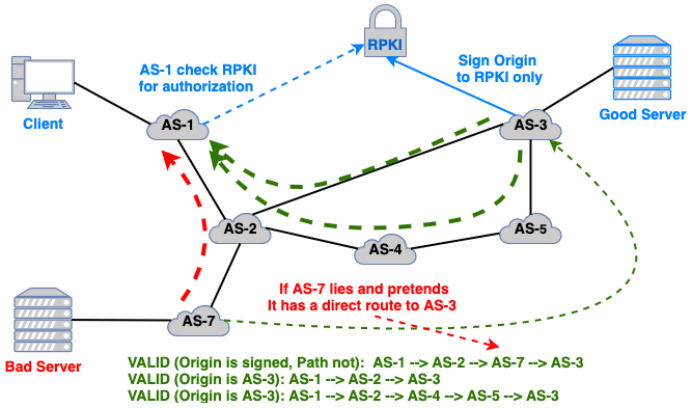
Figure 2.5: RPKI Encryption Process

Origin validation helps to prevent the unintentional advertisement of routes. Origin validation is a mechanism using RPKI for authenticating route announcements as originating from an intended AS [44].Origin validation performs authentication on one or more RPKI servers for specified BGP prefixes. The border router queries the validated prefix-to-AS mapping database on the RPKI server, to authenticate a prefix and ensures that the prefix originates from an intended AS. The Figure 2.6a shows how origin validation work with whole AS path.

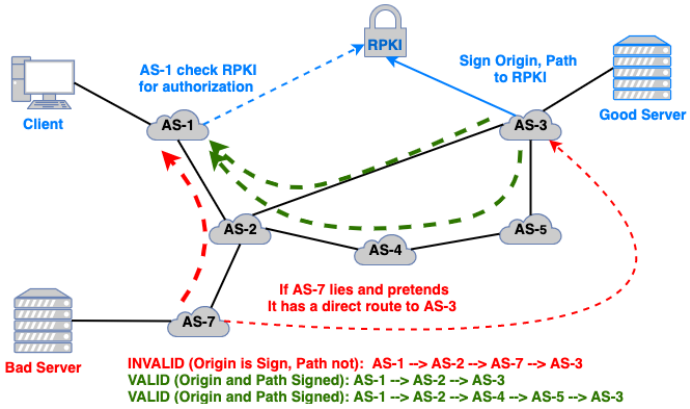
Path validation is similar as origin validation, they both use PRKI to sign certification. Path validation is a mechanism used to supplement the origin validation by verifying AS Path of the announcement [12]. Path validation use one or more RPKI servers to perform authentication for AS. The origin AS check its following AS in RPKI server to ensures the following AS in sign the certification. Figure 2.6b and Figure 2.6c detailed that how path validation work.



(a) BGP Origin Validation



(b) Without BGP Path Validation



(c) With BGP Path Validation

Figure 2.6: BGP Validation Based on RPKI

2.3 Julia

Julia is a fairly modern language, developed in 2009 by Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman who had the idea of designing a language that was free, high-level, and fast. It has following features [35]:

- As high-level and interactive as Matlab or Python
- AS general purpose as Python
- As productive for technical work as Matlab or Python
- But as fast as C

Moreover, Julia is a general purpose programming language and can be used to write any application, many of its features are well suited for numerical analysis and computational sciences. Based on official statistics result [14], and if normalized that C speed equal 1, the Figure 2.7 shows has fast Julia is.

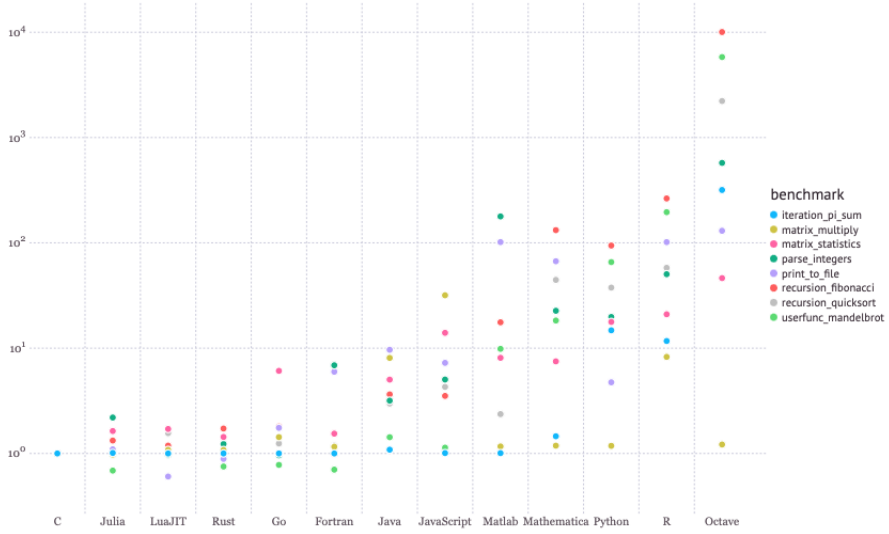


Figure 2.7: How fast is Julia?

Source from: https://julialang.org/benchmarks/

It is necessary to state that the benchmark codes are not written for absolute maximize performance. Rather than, the benchmarks are written to test the performance of identical algorithms and code patterns implemented in each language.

Here is a quote from the creators of Julia from their first official blog article "Why We Created Julia" [13]:

"We want a language that's open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that's homo iconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled."

2.4 Related Work

Previous section, we provided basic background information of our project, and in this section, we discussed the useful related work.

2.4.1 BGP Simulation

An important segment of related work [22, 50, 62] aims at capturing the packet-level details of the Border Gateway Protocol (BGP), which has been the default interdomain routing protocol for decades. BGP++ [22] is built by combining the ns-2 network simulator with the GNU Zebra routing package and the parallel distributed network simulator PDNS. For a topology size of 1000 nodes, the total execution time is around 1000 seconds, and when the topology increases to 2500, the total execution time is about 9000 seconds.

A recent related work [10] develops a distributed network control plane which is called Tiramisu. It uses Path Vector Protocol(TPVP) algorithm to enumerate

paths. A complete verification process should consider coverage and performance. The Tiramisu platform achieves good performance without losing too much coverage at the same time.

A significant difference to related work [10, 56] is that our solver produces all candidate paths towards a destination, not just the best path. BGP allows policy-based routing, that is, the best path is not necessarily the shortest. A lot of the BGP literature has used for testing purposes the business relationships identified by Gao [25] as peer-to-peer (p2p), customer-to-provider (c2p), and provider-to-customer (p2c). Although our initial experiments follow these simple policies, our framework is not restricted by these specific policies. However, it can be used to model any arbitrary relationship needed, as more recent literature suggests, e.g. [47, 55].

Ding *et al.* [40] designs the model based on Quagga Routing Software Suite to study BGP routing update messages in parallel processing with multithreading technology. Compared to the original module, the parallel module performs better at the mid-stage. However, as time goes by, the efficiency worsens the original module. To improve efficiency, the authors also design the garbage collection mechanism to discard useless resources.

Feamster *et al.* [23] implements the Route Prediction Algorithm to model the routing decision process for each router in a single AS. The Route Prediction Algorithm has three phases. The first phase defines the algorithm as straightforward because it aims for a single router. The second phase of the algorithm computes the set of the best eBGP routes, and the third phase determines the best BGP route from the set of the best eBGP routes. Compared to our work, we solved all candidate paths towards a destination, not just the best path.

In the thesis, "An Analysis of Convergence Properties of the Border Gateway Protocol Using Discrete Event Simulation," Brian J. Premore [48] proposes to implement

and run BGP protocol on top of the SSFNet [62] discrete event simulator. He has studied BGP behavior with several parameters, network characteristics, and protocol characteristics. He also analyzes four network topologies: line, ring, focus, and clique topology with at most 30 hosts.

C-BGP [1, 50] is a BGP simulator addressing routing policy evaluation. C-BGP uses a full BGP implementation, we can see all BGP parameters of each node during simulation process. In the process, each BGP decision by making all routing policies. Also, C-BGP is state driven simulator. It stable, and not change over time for the BGP routers.

McDaniel *et al.* [42] have developed lseb, a large scale BGP simulator, is capable of simulating every AS in the Internet using realistic data. The authors also add attack and defense modules so that different strategies for attacking and defending is easily modified and observed.

2.4.2 Securing BGP

A nice classification of attacks on BGP was made by [43, 45, 15]. Mitseva *et al.* review three fundamental vulnerabilities and attack vectors. First, BGP infrastructure face attack from the outside, e.g., hardware damage. Second, tampering with the underlying protocol data by outsiders. Because the BGP messages transfer based on TCP sessions. Third one is outsiders' intentional corruption of control messages, e.g., Prefix hijacking.

In our project, we focus more on prefix hijacking. The common prefix hijacking is AS falsely announcement which is discussed in Section 2.2. A recent study [59] shows that more than 20 % of the global IPv4 address space is not available to the public, making it potentially vulnerable to attack, like BGP hijacking.

BGP prefix hijacking allows malicious ASes falsely announcement its IP prefixes to intercepte or black network traffic [17]. So, BGP prefix hijacking is threat to

network administrator and users. In the paper [57], the author give the survey to network administrator around world. The result shows the RPKI deployment is limited, and more than 71% network administrator said that they have not deploy RPKI infrastructure.

A true story happened to us, China Telecom's BGP Hijacking [21, 9, 30]. On April 8th, 2010 China Telecom hijacked 15% of the Internet traffic for 18 minutes. And now, China Telecom out of the US market.

The study [16] of one-year BGP traffic study of 40 globally distributed ASs shows that, on average, less than 2% of prefixes are advertised using more than 10 paths, and less than 0.06% of prefixes are advertised using more than 20 paths. Less than 0.06% of prefixes are advertised using more than 20 paths in a month. Butler *et al.* design and implement BGP path authentication mechanism to reduce cost by exploiting path stability.

RPKI is an encryption method to sign the record with the origin IP prefix and AS number. It is also an efficient method to secure the inter-domain routing system. Wählisch *et al.* [60] look at BGP tables and updates from RIPE's RRC00 [4] and RouteViews [5] and check updates which against RPKI system. And recent study [18] shows that more than 12.1 % of the global IPv4 address space is covered by Route Origin Authorizations (ROAs). 94.3% of announcements covered by ROAs are valid based on RouteViews dataset. But we also should know, the RPKI certificate is limit, only 35% IPv4 Prefix-Origin Pairs is valid in the RPKI-ROV system [2].

Although RPKI is a critical security tool, we still face many challenges and problems that show in [33, 26, 52, 24], specific for securing origin. Gilad *et al.* [27] design and implement DISCO, a Decentralized Infrastructure for Securing and Certifying Origins, to securing and against BGP prefix hijacking. In comparison to RPKI ROA, DISCO ROA has the same authorization process. However, DISCO ROA also spec-

ifies a list of excluded sub-prefixes for each prefix in the ROA. For the Path-End validation question, DISCO adopt basic and simple method, just like [20, 19].

There is the chicken-and-egg problem in PRKI deployment, authentication is only valid when ROVs are deployed, and ROVs are valid only for certified IP address blocks. As a result, almost no one performs ROV [33, 26]. Oryu *et al.* [46] simulate RPKI deployment process to find a balance between the cost and benefits decision of each AS. The authors compute balance between cost of RPKI deployment and reduction of BGP prefix hijacking.

2.4.3 Monte Carlo Simulation and Julia

Yeh [61] introduces a Hybrid Monte Carlo method to estimate reliability of network. The important term in this paper, the author mentioned Crude Monte Carlo (CMC) method. And every proposed Monte Carlo method should compare with CMC to reflect its efficiency.

A significant difference to related work [53] is that our All Paris Monte Carlo method tries to produce all possible paths for the whole network topology, not just the all possible paths between two nodes. In the paper, Roberts *et al.* designs and implements two Monte Carlo methods: original Monte Carlo method and length distribution Monte Carlo method. We will discuss the original Monte Carlo method in Section 3.6. The length distribution Monte Carlo method based on the original Monte Carlo method, add new parameters l. And the authors estimate l using the Crude Monte Carlo (CMC) estimator [61, 38].

Kwon [36] also introduces the same original Monte Carlo method in his book. Compared with another related work [53], Kwon is more focused on how to use Julia programming language to implement the original Monte Carlo method. He also created a new Julia package, PathDistribution [37], which includes two Monte Carlo methods that we discussed in the previous paragraph.

Chapter3
METHODOLOGY

The methodology to be used in this project is described in this chapter. Section 3.1 explains how do we generate a random network and represent it in a matrix. Latin-Multiplication method and A* algorithm are described in Section 3.2 and 3.3. In the Section 3.4, we introduce the Autonomous System(AS) Relationship Policy that is used during path enumeration process. Section 3.5 describes how do we generate routers within the existed the random networks.Section 3.6 introduces the Monte Carlo method and how to use it to estimate the path between node and estimate all possible paths for the network.

3.1 Network Model

We use the Julia Geometry2D package [54] to generate the network topology. This Julia package generates random ellipses and points, which are used to represent the Autonomous Systems (ASes) and PoPs (Points of Presence). The links between ASes are established when both ASes are present in the same PoP. If the points do not fall into the intersection of two ellipses, we will ignore them. The links also describe the AS business relationships, and will discuss it in Section 3.4.

Based on the area of random ellipses, we define the type of Autonomous Systems:

Tier-1: ASes that only provide transit and never buy it.

Tier-X: ASes that both provide and buy transit. Providers can be other Tier-1 or other Tier-X.

Stubs: ASes that appear only at the beginning of the AS path. They only buy transit (from both Tier-1 and Tier-X) and never provide it.

The following Figures 3.1, 3.2, and 3.3 describe network generation process.

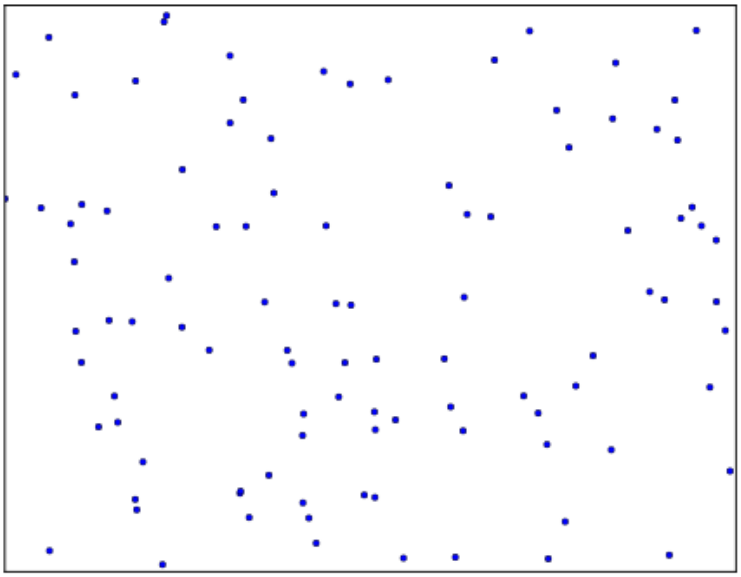


Figure 3.1: Generated PoPs

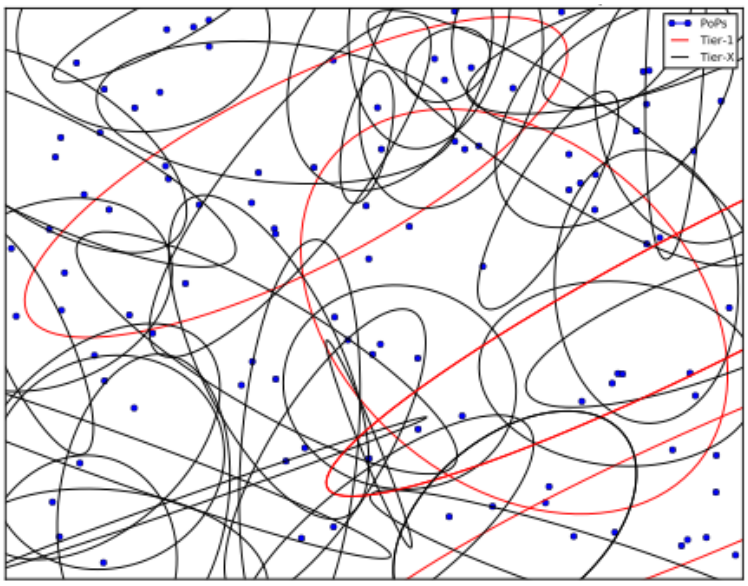


Figure 3.2: Generated Ellipses and PoPs

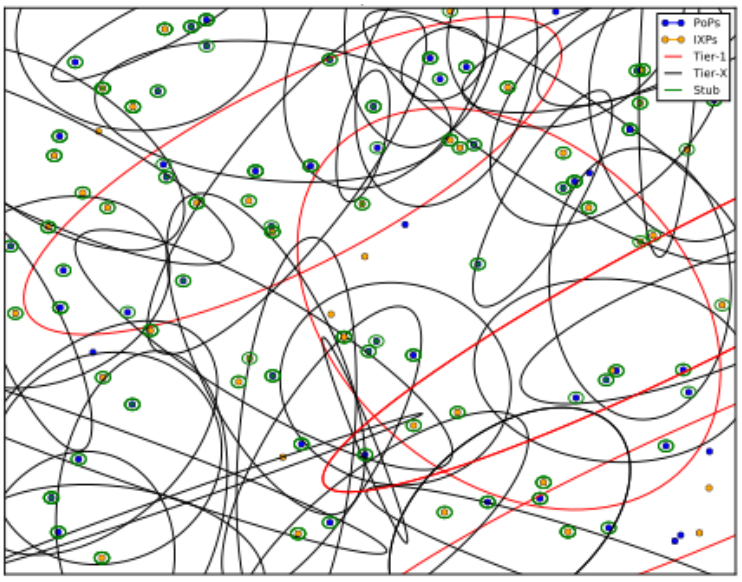


Figure 3.3: Final Ellipses and PoPs

We use the metrics to represent the generated data for further study. The network topology is captured through the Connection Matrix C. Each matrix element indicates whether two ASes have a connection at a specific PoPs. For instance, in the following connection matrix C, the C_{23} value means that AS-2 and AS-3 are connected at PoP 4.

$$Connection\ Matrix\ C\ = \begin{pmatrix} 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 3 & 1 & 2 & 0 & 0 & 5 \\ 0 & 3 & 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The business relationships are captured through the relationship matrix R. Our

model does not pose any limitation on the number or type of policies that can be captured. On the contrary, our design was guided by the idea to allow a diverse set of policies for the user to define.

For our initial experiments, we maintain a set of policies often used in the literature, and we codify each relationship type through an integer number.

1 : provider-to-customer (p2c);

2 : customer-to-provider (c2p);

3 : peer-2-peer (p2p)

A more diverse set of policies can be modeled by assigning additional integers to corresponding relationship types. A relationship type could also be compound, i.e., being the result of merging together with other policies. In the example relationship matrix R below, the value R_{23} is 1 and indicates that AS-2 is a provider to As-3.

$$Relationship\ Matrix\ R\ = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 3 & 2 & 2 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The example network corresponding to matrices C and R is depicted in Figure 3.4.

3.2 Latin Multiplication Method

Latin-Multiplication [29] is an algebraic method useful in enumerating paths. Let X* be the set of all paths, which includes the empty path $\emptyset$. Latin-Multiplication $\otimes$

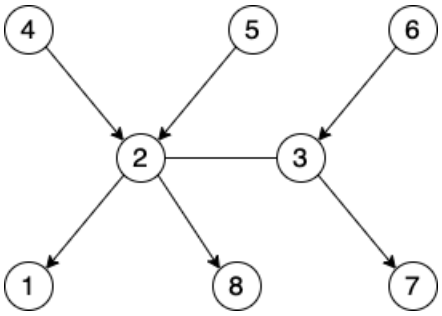


Figure 3.4: Network modeled through matrices C and R.

between two paths $u_{\alpha i}$ and $u_{\beta j}$ means path concatenation if the last node of $u_{\alpha i}$ is the same as the first node of $u_{\beta j}$. In case this condition does not hold, the Latin-Multiplication between the two paths produces the empty path.

More formally,

if

$$\mathrm{u} \otimes \emptyset = \emptyset \otimes \mathrm{u} \qquad \forall\ \mathrm{x} \in \mathrm{S}$$

$$u_{\alpha i} = (i_1, i_2, ...i_k)$$

$$u_{\beta j} = (j_1, j_2, ...j_l)$$

then

$$u_{\alpha i} \otimes u_{\beta j} = \begin{cases} i_1, i_2, ...i_k, j_2, j_3, ..., j_l & \text{if } i_k = j_1 \\ \emptyset & \text{otherwise} \end{cases}$$

Assume that matrix u_α is two dimensional and contains elements that are paths. For instance, element $u_{\alpha i}$ would be a path starting from node i_1 and ending at node i_k located at position $(1, k)$ of the matrix. If matrix u_β also contains paths of the same nodes, then we can define multiplication between two path matrices as:

$$\text{if } u_\alpha = (u_{\alpha i}) \qquad \text{with } u_{\alpha i} \in \mathrm{X}^*,$$

$$u_\beta = (u_{\beta j}) \qquad \text{with } u_{\beta j} \in \mathrm{X}^*,$$

$$\text{then } u_\alpha \otimes u_\beta = (u_{\alpha i} \otimes u_{\beta j}).$$

3.3 A* Algorithm and Path Matrix Multiplication

The A* algorithm, algorithm 1, uses matrix multiplication theory and Latin Multiplication [29] to enumerate all possible paths in a network of nodes. Matrix A is the adjacency matrix whose elements are the single link paths between nodes. Multiplying A with itself will result in matrix $A^{(2)}$ that contains all paths of length at most three edges. Multiplying A with $A^{(2)}$ results in $A^{(3)}$, which contains all paths of length at most four and so on. By continuing to multiply A with the newest result of the series every time, say $A^{(n)}$, the algorithm is proven to terminate and eventually produce a matrix containing all possible paths among all nodes.

Algorithm 1: A* Algorithm

Input: Adjacency Matrix A (single link paths)
Output: $A^{(2)}$

```
initialization;
m = size(A)[1] ; // # of Rows for A
n = size(A)[2] ; // # of Columns for A
for i ← 1 to m do
    for j ← 1 to n do
        for k ← 1 to n do
            // If two nodes are equal
            fA[i, j][end] == A[j, k][1] F = A[i, j];
            S = A[j, k][2:end];
            A^(2)[i,k] = [F,S];
        end
    end
end
return A^(2)
```

The A* algorithm can enumerate all possible paths of a network of nodes. In our project, we use the matrix to represent network connection status and the relationship between ASes. We implement the path matrix multiplication method based on the A* algorithm.

Link Matrix L is the adjacency matrix whose elements are the single link paths

between the ASes. Path Matrix P is the incremental matrix whose elements are routing paths between the ASes. Routing Table RT is the matrix that is used to save our routing paths results. And elements of RT may be a set of the path between the ASes.

According to A* algorithm, path matrix multiplication algorithm, algorithm 2, multiplying L with P_1 will get result P_2, multiplying L with P_2 results in P_3 and so on. In each iteration, we save result P_i to RT until we cannot find new path in Path Matrix P which means Path Matrix P is the zero matrix.

Algorithm 2: Path Matrix Multiplication

Input: Path Matrix P_i
Input: Link Matrix L
Output: Routing Table RT

initialization;
RT = Path Matrix P_i
m = size(P_i)[1] ; `// # of Rows for` P_i
n = size(P_i)[2] ; `// # of Columns for` P_i
RT = Path Matrix P_i
flag = m * n ; `// Stop iteration flag`
iter = 0 ; `// # of Iteration`
while *# of 0 in* $P_{i+1} \neq$ *flag* **do**
    P_{i+1} = A* Algorithm(P_i, L);
    RT = RT + P_{i+1} ;
    iter = iter +1 ;
end
return RT

3.4 Autonomous System(AS) Relationship Policy

The Autonomous System(AS) relationship policies often imply rules on path validity and preference. In our example set of policies, we have implemented the valley-free set of rules [28] to distinguish between valid and invalid paths. A path that does not follow the valley-free rules is invalid.

The valley-free rules [28] define routing path patterns. Assuming n and m are positive integers, a routing path is valley-free, if and only if it consists of links that

follow one of these two business relationship patterns:

$$n \times c2p + m \times p2c$$

$$n \times c2p + p2p + m \times p2c$$

where n and m >0.

In other words, there should not be adjacent p2p links within a valid routing path, and a c2p link should not follow a p2c link. Figure 3.5 shows a valley-free rule.

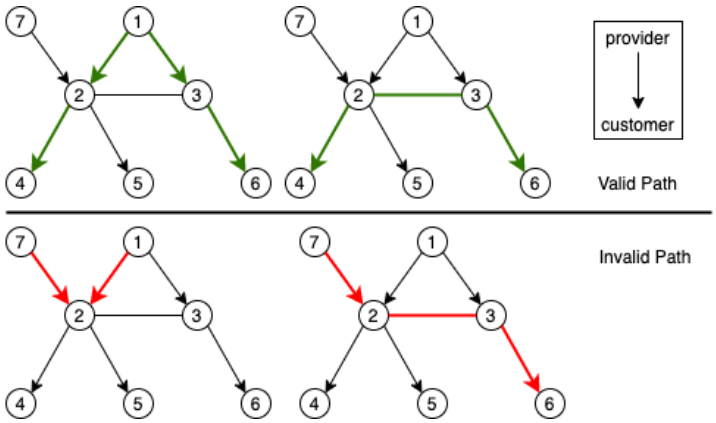


Figure 3.5: Relationship Policy - Valley Free

Our model allows one to define arbitrary policy rules, the valley free set being just an example. We also mark a path as invalid if it contains the same node twice, calling it a loop. A loop describes the case where a packet keeps getting routed in an endless circle through the same nodes rather than reaching its intended destination. The loop example is shown in Figure 3.6.

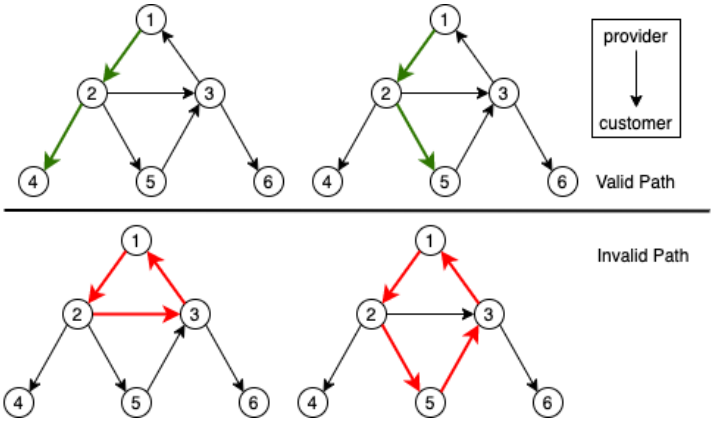


Figure 3.6: Relationship Policy - Loop

In our initial experiments, a path is valid if it obeys both the valley-free rules and loop-free rule. In the pseudo-code, we can write AS relationship policy as following algorithm 3.

Path Matrix P is the incremental matrix whose elements are routing paths between the ASes. We perform the algorithm 3 during simulation process for every new Path Matrix.

3.5 Router Level Network Generation

In the previous section, we discuss the Network model. The original network model used the Julia Geometry2D to generate the AS level network topology. In this section, we will discuss how to generate router-level network topology and how to simulate router-level routing paths.

3.5.1 Router Generation for the Network Model

For the router generation, we use the Julia Geometry2D package to generate random points within the ellipses(ASes), which represent the routers. Based on the type of ASes, we generate the different number of routers for each ASes:

- Tier-1 AS: 5 routers
- Tier-X AS: 3 routers

Algorithm 3: AS Relationship Policy Check

```
   Input: temp Path Matrix P
   Input: Connection Matrix C
   Input: Relationship Matrix R
   Output: Valid Path Matrix P
 1 initialization;
 2 m = size(P)[1] ; // # of Rows for P
 3 n = size(P)[2] ; // # of Columns for P
 4 Valid Path Matrix P = Matrix[m,n] for i ← 1 to m do
 5 |   for j ← 1 to n do
 6 |   |   if P[i, j] == union(P[i, j]) then
   |   |   |   // Check Loop
 7 |   |   |   temp = P[i, j] ;
 8 |   |   |   if Relationship Patterns of P[i, j] is valley free then
   |   |   |   |   // Check Valley Free
 9 |   |   |   |   Valid Path Matrix P[i, j] = temp ;
10 |   |   |   else
11 |   |   |   |   Valid Path Matrix P[i, j] = 0 ;
12 |   |   |   end
13 |   |   else
14 |   |   |   Valid Path Matrix P[i, j] = 0 ;
15 |   |   end
16 |   end
17 end
18 return Valid Path Matrix P
```

- Stubs AS: 1 router

And for each router, we random generate local preference value between 1 to 6.

3.5.2 Border Router Selection Between Two ASes

The BGP border router selection process based on the router's distance in different ASes. The following Figure 3.7 shows the simple router level network model, AS-1 has three routers, and AS-2 has four routers.

We use the matrices to represent the distance data of two ASes. Each matrix element indicates a distance between two routers between two ASes. For instance, in the following distance matrix D, the d_{23} means the distance between Router-2 in

Figure 3.7: Simple Router Level Network.

AS-1 and Router-3 in AS-2

$$Distance\ Matrix\ D\ =\ \begin{array}{c} \\ AS-1 \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} AS-2 & & & \end{array} \\ \begin{pmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \end{pmatrix} \end{array}$$

Based on the distance matrix between two ASes, we create the router pair matrix for the whole network model which used to represent border router for each AS.

Single Router Pair Matrix

The Single Router Pair Matrix SP only choose the one distance value in distance matrix D. If we choose same distance value in previous example. For instance, in the following Single Router Pair Matrix SP, the SP_{12} element is [2,3], and it means that Router-2 in AS-1 and Router-3 in AS-2 are selected (if AS-2 and AS-3 has connection during simulation process).

$$Single\ Router\ Pair\ Matrix\ SP\ = \begin{array}{c} \\ AS-1 \\ AS-2 \\ AS-3 \end{array} \begin{array}{c} \begin{array}{ccc} AS-1 & AS-2 & AS-3 \end{array} \\ \begin{pmatrix} 0 & [2,3] & [1,5] \\ [3,2] & 0 & [3,5] \\ [5,1] & [5,3] & 0 \end{pmatrix} \end{array}$$

Multiple Router Pair Matrix

The Multiple Router Pair Matrix MP choose the multiple distance value in distance matrix D, For instance, in the following Multiple Router Pair Matrix MP, the DP_{12} values are [2,3] and [1,4]. And [2,3] means Router-2 in AS-1 and Router-3 in AS-2 are selected (if AS-2 and AS-3 has connection during simulation process). Similarly, [1,4] means Router-1 in AS-1 and Router-4 in AS-2 are selected (if AS-2 and AS-3 has connection during simulation process).

$$Multiple\ Router\ Pair\ Matrix\ MP\ = \begin{array}{c} \\ AS-1 \\ AS-2 \\ AS-3 \end{array} \begin{array}{c} \begin{array}{ccc} AS-1 & AS-2 & AS-3 \end{array} \\ \begin{pmatrix} 0 & [2,3][1,4] & [1,5][2,4] \\ [3,2][4,1] & 0 & [3,5][1,2] \\ [5,1][4,2] & [5,3][2,1] & 0 \end{pmatrix} \end{array}$$

3.5.3 Routing Policy

Border Gateway Protocol routers typically receive multiple paths to the same destination. Policy-based routing is a technique used to make routing decisions based on policies. This section will discuss the path selection algorithm that we used during the router-level simulation process.

- Prefer Local Preference

  Prefer local preference means to prefer the routing path with the highest local preference.

- Prefer Shortest Distance

  Prefer the shortest distance refers to finding a path through a network with a minimum distance.

- Prefer Neighbor

  Prefer neighbor means prefer the routing path with the closest neighbor router.

3.6 Monte Carlo Method

In the previous section, we discussed the Latin-Multiplication method and A* algorithm to enumerate all possible paths with a given network. This section will discuss the original Monte Carlo method [53, 36] in Section 3.6.1. And how to modify the original Monte Carlo method to enumeration all possible path in Section 3.6.2.

3.6.1 Original Monte Carlo Method

The Monte Carlo method is a straightforward approach to simulation results. Consider the simple 5-nodes network in Figure 3.8, and there are 64 paths for the whole network and 4 path from origin node 1 to destination node 5.

Let P be set of all possible path, and $|P|$ denote number of all possible path. With a same example, from node 1 to node 5, P = {[1, 2, 4, 3, 5], [1, 4, 2, 5], [1, 2, 5], [1, 4, 3, 5]} and $|P| = 4$.

Following the same structure with the Latin Multiplication method, we use the adjacency matrix A to represent the network, and the A_{ij} means that a link between node i and j. For instance, the network in Figure 3.8 can be rewritten in adjacency matrix A as follows:

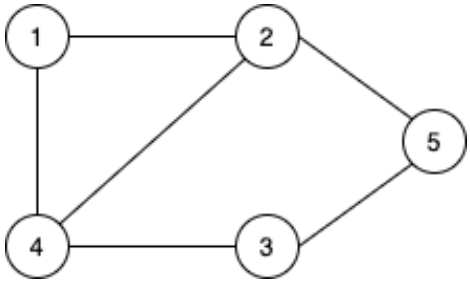


Figure 3.8: A simple network with 5 nodes.

$$Adjacency\ Matrix\ A\ = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

Like in other Monte Carlo method [61], we need generate many samples to estimate $|P|$. To illustrate this process, let's use the network in Figure 3.8 to find all possible paths between origin node 1 and destination node 5.

1. Start with origin node 1. Node 1 connects with node 2 and 4. Then randomly choose a node with uniform distribution. Let's say node 2 is chosen with probability 1/2 , and current path is [1,2].

2. Next work with node 2. Node 2 connects with node 1, 4 and 5. Since node 1 is already visited, so random choose node 4 or 5 with uniform distribution. Let's say node 4 is chosen with probability 1/2 , and the current path is [1,2,4].

3. Then works with node 4. Node 4 connects with 1, 2, and 3. Since node 1 and 2 are already visited, node 3 is chosen with probability 1 and the current path [1,2,4,3].

4. At last, work with node 3. Node 3 connects with 4 and 5. Since node 4 is already visited, node 5 is chosen with probability 1 and the current path [1,2,4,3,5].

5. Since we arrive at destination node 5, stop the iteration.

By above process, one sample path is generated. Let's denote this sample path by S_1, and probability P for this sample path is $\mathrm{P}(S_1) = 1/2 * 1/2 * 1 * 1 = 1/4$. If we repeat this process N times, we will get N sample paths. In each iteration, we stop if either the destination node is reached or there is no more unvisited node connected to the current node. We can obtain:

S_1	S_2	...	S_{N-1}	S_N
$\mathrm{P}(S_1)$	$\mathrm{P}(S_2)$	...	$\mathrm{P}(S_{N-1})$	$\mathrm{P}(S_N)$

To estimate the number of paths based on the above N samples, we compute in the following equation:

$$\widehat{|P|} = \frac{1}{N}\sum_{i=0}^{N}\frac{S_i}{P(S_i)} \tag{3.1}$$

If N is large enough, we have $\widehat{|P|} \approx |P|$. The original Monte Carlo method can be defined in following algorithm 4.

3.6.2 All Paris Monte Carlo Method

To estimate all possible paths just like the original Monte Carlo process, we need to obtain the probability for each sample path, not just the sample path with a specific origin and destination. To illustrate this process, let's use the network in Figure 3.8.

1. Start with node 1. Node 1 connects with node 2 and 4. Then randomly choose a node with uniform distribution. Node 2 is chosen with probability 1/2, so path is [1,2] and its probability $P_{[1,2]} = 1/2$. Node 4 is is chosen with probability 1/2, so path is [1,4] and its probability $P_{[1,4]} = 1/2$.

2. Next work with node 2 and 4. Node 2 connects with node 1, 4 and 5. Since node 1 is already visited, so random choose node 4 or 5 with uniform distribution. Node 4 is chosen with probability 1/2, so path is [1,2,4] and its probability $P_{[1,2,4]} = 1/2 * 1/2 = 1/4$. Node 5 is chosen with probability 1/2, so path is [1,2,5] and its probability $P_{[1,2,5]} = 1/2 * 1/2 = 1/4$.

   Node 4 connects with 1, 2 and 3. Since node 1 is already visited, so random choose node 2 or 3 with uniform distribution. Node 2 is chosen with probability 1/2, so path is [1,4,2] and its probability $P_{[1,4,2]} = 1/2 * 1/2 = 1/4$. Node 3 is chosen with probability 1/2, so path is [1,4,3] and its probability $P_{[1,4,3]} = 1/2 * 1/2 = 1/4$.

3. Next step, working with node 4, 5, 2, and 3 since we get path [1,2,4], [1,2,5], [1,4,2], and [1,4,3]. Then repeat the last step to find each path with a non-visited node and its probability.

4. Repeating the above step until the path length is same as the number of nodes. Then stop the iteration.

By above process, in each iteration process we will generate multiple sample paths. We use $S_{[path]}$ to represent generated sample path, and use $P_{[path]}$ to represent the probability of this sample path. If we repeat this process N times, we will get more than N sample paths. In each iteration, we stop if either the path is equal number of nodes or there is no more unvisited node connected to the current node. We can obtain:

$$\begin{array}{cccccccc} S_{[1,2]} & S_{[1,4]} & S_{[1,2,4]} & S_{[1,2,5]} & S_{[1,4,2]} & S_{[1,4,3]} & ..... \\ P_{[1,2]} & P_{[1,4]} & P_{[1,2,4]} & P_{[1,2,5]} & P_{[1,4,2]} & P_{[1,4,3]} & ..... \end{array}$$

To estimate the number of paths based on the above sample paths, we use the same equation 3.1 to compute the result. Therefore, the all pairs Monte Carlo method

is defined in the following algorithm 5.

Algorithm 4: Original Monte Carlo Method

```
   Input: Adjacency Matrix A
   Input: Sample Size N
   Input: Origin Node org
   Input: Destination Node dest
   Output: PathSample S
 1 initialization
 2 S = PathSample[]
 3 for i ← 1 to N do
 4 |   path = [org]
 5 |   g = 1
 6 |   current = org
 7 |   A[:,org] .= 0.
 8 |   while current != dest do
 9 |   |   V = []
10 |   |   m =  of node
11 |   |   for j ← 1 to m do
12 |   |   |   if A[current,j] == 1 then
13 |   |   |   |   push!(V, j)
14 |   |   |   end
15 |   |   end
16 |   |   if length(V) == 0 then
17 |   |   |   Break
18 |   |   end
19 |   |   next = rand(V)
20 |   |   path = [path; next]
21 |   |   current = next
22 |   |   A[:,next] .= 0
23 |   |   g = g / length(V)
24 |   end
25 |   if path[end] == dest then
26 |   |   this_sample = PathSample(path, g)
27 |   |   push!(S, this_sample)
28 |   end
29 |   return S
30 end
```

Algorithm 5: All Pairs Monte Carlo Method

```
Input: Adjacency Matrix A
Input: Sample Size N
Input: Number of node M
Output: PathSample S
initialization
S = PathSample[]
for i ← 1 to N do
    for origin ← 1 to M do
        path = [origin]
        g = 1
        current = origin
        A[:, origin] .= 0
        while length(path) != M do
            V = []
            m =  of node
            for j ← 1to m do
                if A[current,j] == 1 then
                    push!(V, j)
                end
            end
            if length(V) == 0 then
                Break
            end
            next = rand(V)
            path = [path; next]
            current = next
            A[:,next] .= 0
            g = g / length(V)
            this_sample = PathSample(path, g)push!(S, this_sample)
        end
    end
    return S
end
```

Chapter4
SYSTEM COMPONENTS

In the previous chapter, we provided the background on the A* algorithm we use for path enumeration. We also discussed our network topology, AS relationship policy, router level network topology, and Monte Carlo method.

In this chapter, we have concluded by describing the interactions among our system components and the path prefix attack.

4.1 Routing Path Solver

To enumerate all possible valid routing paths, we have enhanced the A* algorithm and path matrix multiplication method described in Section 3.3 with an additional rule regarding the concatenation of paths. The Latin-Multiplication $u_{\alpha i} \otimes u_{\beta j}$ results in the non-empty path, if both $i_k = j_1$ and the resulting path is valid according to the AS relationship rules described in Section 3.4.

4.1.1 General Path Solver

We have implemented three versions of the routing path solver based on the A* algorithm and path matrix multiplication method in the high-performance programming language Julia:

1. Serial Implementation with Julia Build in Array
2. Parallel Implementation with Julia Build in Array
3. Parallel Implementation with Julia Shared-Array

The Julia built-in array is an ordered collection of elements. Julia Shared Arrays use system shared memory to map the same array across many processes [49]. Im-

plementing the Shared Array data structure takes advantage of the shared memory parallel programming model across all processes.

In Julia, only "isbits" elements are supported in a Shared-Array. Each element of the Shared-Array must be of Bit type, which is a Julia "plain data" type, meaning it is immutable and contains no references to other values [49].

To represent the path in our path solver, we use arrays of the array with Julia Build in Array. For instance, path P = [[1,2,3],[1,4,3]]. But in the Julia Shared-Array, arrays of the array is not allowed since arrays are not the Bit type.

The common elements of all three implementations are depicted in the following Figure 4.1.

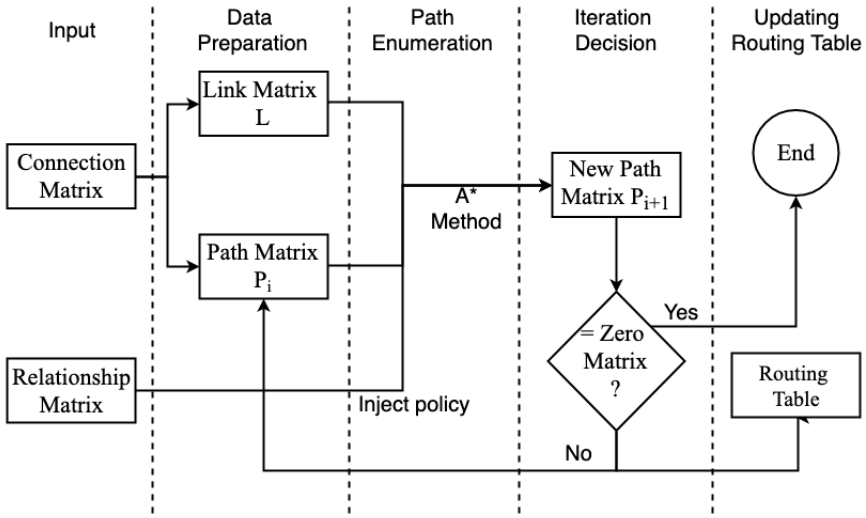


Figure 4.1: Flowchart of the Path Solver.

With the simple network from Figure 4.2, the connection matrix C and relationship matrix R are:

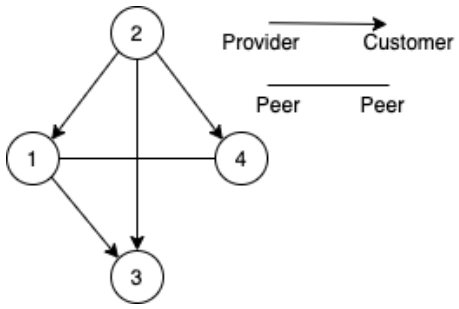


Figure 4.2: Simple Network.

$$Connection\ Matrix\ C\ = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 2 & 4 \\ 2 & 4 & 0 & 0 \\ 3 & 1 & 0 & 0 \end{pmatrix}$$

$$Relationship\ Matrix\ R\ = \begin{pmatrix} 0 & 2 & 1 & 3 \\ 1 & 0 & 1 & 1 \\ 2 & 2 & 0 & 0 \\ 3 & 2 & 0 & 0 \end{pmatrix}$$

In the data preparation process, we construct the Link Matrix L and the Path Matrix P_i from connection matrix C and relationship matrix R. The link matrix contains the adjacent links of the topology and does not change. Therefore, the link matrix is:

$$\mathrm{L} = P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

At path enumeration step of the path solver performs A* algorithm between L and P_i to construct P_{i+1}. As mentioned, we have modified the original A* algorithm so that we eliminate from P_i those paths that do not conform to the AS relationship policy rules. For instance, P_1 does not contain any loops or paths that are not valley free:

$$\mathrm{L} \otimes P_0 = P_1 = \begin{pmatrix} 0 & 0 & [1,2,3] & [1,2,4] \\ 0 & 0 & [2,1,3] & 0 \\ [3,2,1] & [3,1,2] & 0 & [3,2,4] \\ [4,2,1] & 0 & [4,2,3] & 0 \end{pmatrix}$$

The routing table RT at a given time is produced by adding the P_i matrices calculated up to that point. Note that matrix P_{i+1} contains paths that are longer by one link compared to the paths of matrix P_i.

$$\mathrm{RT} = \mathrm{RT} + P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_1 = \begin{pmatrix} 0 & [1,2] & [1,3][1,2,3] & [1,4][1,2,4] \\ [2,1] & 0 & [2,3][2,1,3] & [2,4] \\ [3,1][3,2,1] & [3,2][3,1,2] & 0 & [3,2,4] \\ [4,1][4,2,1] & [4,2] & [4,2,3] & 0 \end{pmatrix}$$

It is essential to note an important feature of our path solver solution with the A* algorithm. The element at position (l, m) of the routing table RT matrix contains a *set* of paths rather than a single path: it contains all paths from node l to node m

that are acceptable under given policies.

4.1.2 Shared-Array Path Solver

In Section 4.1.1, we have learned that arrays of Shared-Array is not allowed. To use the advantage of the Shared-Array, we need to declare the Bit-type data structure to replace arrays.

Our new Bit-type data structure *Path* is defined as:

```
struct Path{N}
    o::Int64 #Origin Node
    d::Int64 #Destination Node
    p::NTuple{N,Int64} #Path
end
```

With Bit type Shared-Array, we construct the Link Matrix L and the Path Matrix P_i by using the Bit type *Path*. Using the same example in Section 4.1.1, the simulation process looks like this:

At the data preparation step:

$$\mathrm{L} = P_0 = \begin{pmatrix} P(0,0,(0,0)) & P(1,2,(1,2)) & P(1,3,(1,3)) & P(1,4,(1,4)) \\ P(2,1,(2,1)) & P(0,0,(0,0)) & P(2,3,(2,3)) & P(2,4,(2,4)) \\ P(3,1,(3,1)) & P(3,2,(3,2)) & P(0,0,(0,0)) & P(0,0,(0,0)) \\ P(4,1,(4,1)) & P(4,2,(4,2)) & P(0,0,(0,0)) & P(0,0,(0,0)) \end{pmatrix}$$

where P = Path{Int64,Int64,Tuple{}}.

At the path enumeration step:

$$\mathrm{L} \odot P_0 = P_1 \begin{pmatrix} 0 & 0 & P(1,3,(1,2,3)) & P(1,4,(1,2,4)) \\ 0 & 0 & P(2,3,(2,1,3)) & 0 \\ P(3,1,(3,2,1)) & P(3,2,(3,1,2)) & 0 & P(3,4,(3,2,4)) \\ P(4,1,(4,2,1)) & 0 & P(4,3,(4,2,3)) & 0 \end{pmatrix}$$

where P = Path{Int64,Int64,Tuple{}}.

The data type conversion from Julia Shared-Array to Julia Build-in Array during

the routing table calculation process.

$$\mathrm{RT} = \mathrm{RT} + P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_1 = \begin{pmatrix} 0 & [1,2] & [1,3][1,2,3] & [1,4][1,2,4] \\ [2,1] & 0 & [2,3][2,1,3] & [2,4] \\ [3,1][3,2,1] & [3,2][3,1,2] & 0 & [3,2,4] \\ [4,1][4,2,1] & [4,2] & [4,2,3] & 0 \end{pmatrix}$$

4.1.3 Parallel Path Solver

The parallel A* implementation parallelizes the matrix multiplication dividing column of Link matrix L among different threads that all share Path matrix P_i and produce other rows of the matrix P_{i+1}. The parallel process is described in Figure 4.3.

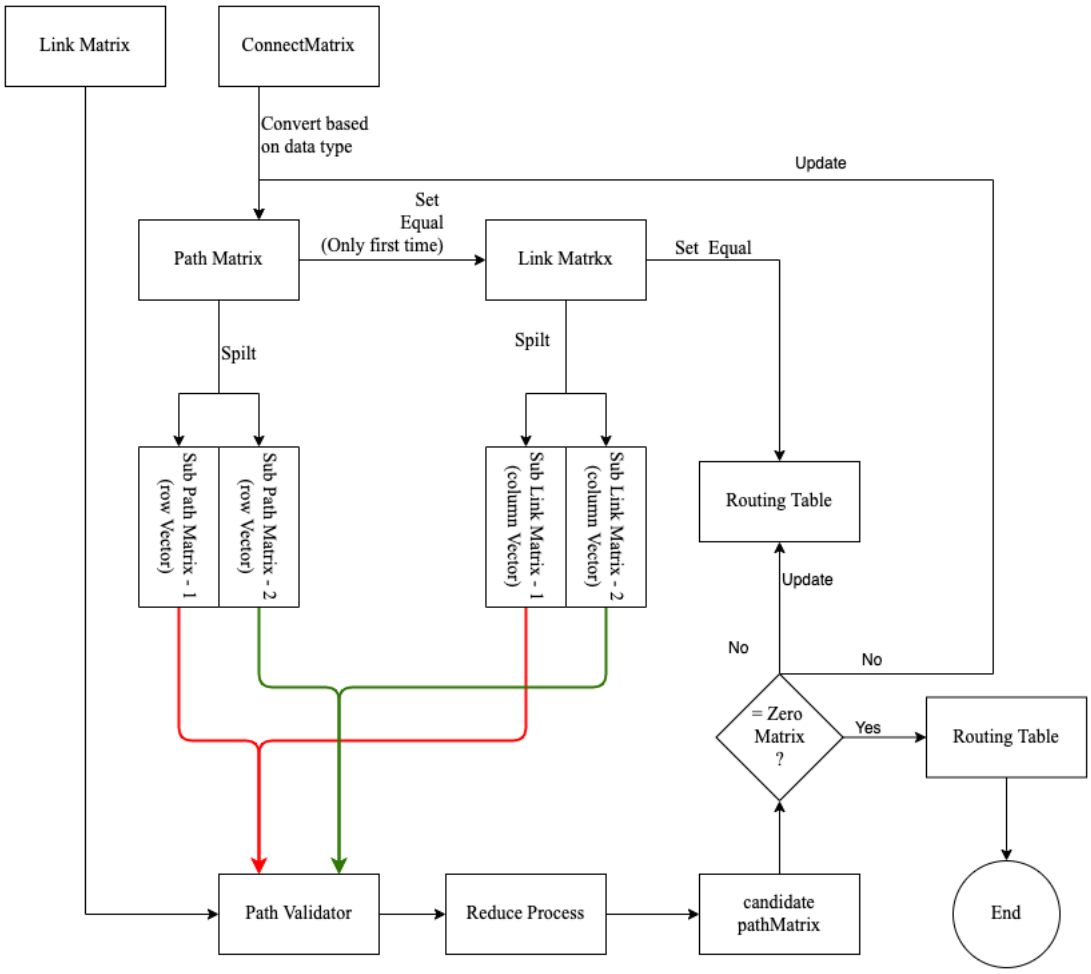


Figure 4.3: Flowchart of the Parallel Path Solver.

If we use the same example from Section 4.1.1, the parallel simulation process with two cores looks like this:

$$\mathrm{L} = P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

On the core-1:

$$\text{Sub-}P_0\text{-1} = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \end{pmatrix}$$

$$\text{Sub-L-1} = \begin{pmatrix} 0 & [1,2] \\ [2,1] & 0 \\ [3,1] & [3,2] \\ [4,1] & [4,2] \end{pmatrix}$$

$$\text{Sub-L-1} \otimes \text{Sub-}P_0\text{-1} = \text{Sub-}P_1 - 1 = \begin{pmatrix} 0 & 0 & [1,2,3] & [1,2,4] \\ 0 & 0 & [2,1,3] & 0 \end{pmatrix}$$

On the core-2:

$$\text{Sub-}P_0\text{-2} = \begin{pmatrix} [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\text{Sub-L-2} = \begin{pmatrix} [1,3] & [1,4] \\ [2,3] & [2,4] \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

$$\text{Sub-L-2} \otimes \text{Sub-}P_0\text{-2} = \text{Sub-}P_1\text{-2} = \begin{pmatrix} [3,2,1] & [3,1,2] & 0 & [3,2,4] \\ [4,2,1] & 0 & [4,2,3] & 0 \end{pmatrix}$$

Reduce process:

$$\text{Sub-}P_1\text{-1} + \text{Sub-}P_1\text{-1} = P_1 = \begin{pmatrix} 0 & 0 & [1,2,3] & [1,2,4] \\ 0 & 0 & [2,1,3] & 0 \\ [3,2,1] & [3,1,2] & 0 & [3,2,4] \\ [4,2,1] & 0 & [4,2,3] & 0 \end{pmatrix}$$

$$\text{RT} = \text{RT} + P_1 = \begin{pmatrix} 0 & [1,2] & [1,3][1,2,3] & [1,4][1,2,4] \\ [2,1] & 0 & [2,3][2,1,3] & [2,4] \\ [3,1][3,2,1] & [3,2][3,1,2] & 0 & [3,2,4] \\ [4,1][4,2,1] & [4,2] & [4,2,3] & 0 \end{pmatrix}$$

4.1.4 Router Level Path Solver

The router-level path solver uses the Single Router Pair Matrix in Section 3.5.2 or Multiple Router Pair Matrix in Section 3.5.2 during the general path solver process to simulate router-level routing path and create a routing table between two specific ASes with the routing policy mark.

For instance, we use the simple network model from Section 4.1.1. Based on AS level network topology, we create a simple router-level network topology that shows in the following Figure 4.4 if we ignore the type of AS.

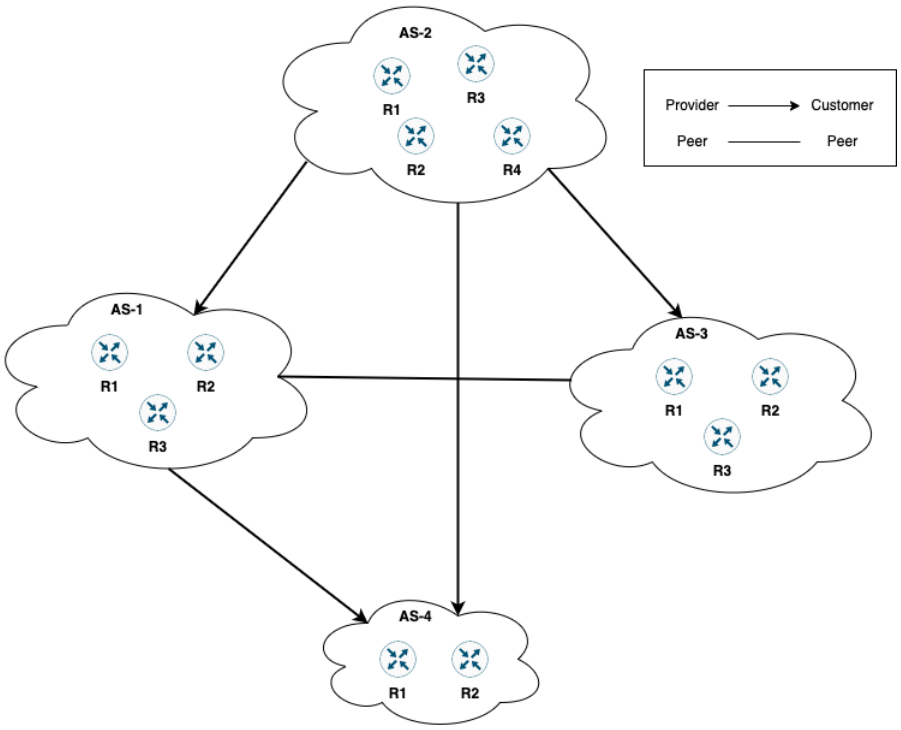


Figure 4.4: Simple Router-Level Network

Based on the router-level network topology, we can create five distance matrix: $D_{AS2,AS1}$, $D_{AS2,AS3}$, $D_{AS2,AS4}$, $D_{AS1,AS3}$, $D_{AS1,AS4}$ used to select border router between ASes. For instance, the distance matrix $D_{AS2,AS1}$ used to select border router pairs between AS-2 and AS-1, and he distance matrix $D_{AS2,AS1}$ is shown in the following:

$$Distance\ Matrix\ D_{AS2,AS1} = \begin{array}{c} \\ AS-2 \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccc} AS-1 & & \end{array} \\ \begin{pmatrix} d_{11} & d_{12} & d_{13} \\ d_{21} & d_{22} & d_{23} \\ d_{31} & d_{32} & d_{33} \\ d_{41} & d_{42} & d_{43} \end{pmatrix} \end{array}$$

The next step is creating the Single Router Pair Matrix or Multiple Router Pair Matrix based on the distance matrix. We use the router-level network in Figure 4.4.

We construct Single Router Pair Matrix SP, where the router pair is the minimum distance between two ASes, and Multiple Router Pair Matrix MP, where the router pair is the minimum and maximum distance between two ASes.

If the Single Router Pair Matrix SP is:

$$SP = \begin{array}{c} \\ AS-1 \\ AS-2 \\ AS-3 \\ AS-4 \end{array} \begin{array}{c} \begin{array}{cccc} AS-1 & AS-2 & AS-3 & AS-4 \end{array} \\ \begin{pmatrix} 0 & [3,2] & [3,1] & [3,2] \\ [2,3] & 0 & [2,1] & [2,1] \\ [1,3] & [1,2] & 0 & 0 \\ [2,3] & [1,2] & 0 & 0 \end{pmatrix} \end{array}$$

The whole simulation process looks like this:

$$\mathrm{L} = P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_0 + \mathrm{SP} = \begin{pmatrix} 0 & [AS1R3, AS2R2] & [AS1R3, AS3R1] & [AS1R3, AS4R2] \\ [AS2R2, AS1R3] & 0 & [AS2R2, AS3R1] & [AS2R2, AS4R1] \\ [AS3R1, AS1R3] & [AS3R1, AS2R2] & 0 & 0 \\ [AS4R2, AS1R3] & [AS4R1, AS2R2] & 0 & 0 \end{pmatrix}$$

$$\mathrm{L} \otimes P_0 = P_1 = \begin{pmatrix} 0 & 0 & [1,2,3] & [1,2,4] \\ 0 & 0 & [2,1,3] & 0 \\ [3,2,1] & [3,1,2] & 0 & [3,2,4] \\ [4,2,1] & 0 & [4,2,3] & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_1 + \mathrm{SP} =$$
$$\begin{pmatrix} ... & ... & [AS1R3, AS3R1][AS1R3, AS2R2, AS3R1] & ... \\ ... & ... & [AS2R2, AS3R1][AS2R2, AS1R3, AS3R1] & ... \\ ... & ... & 0 & ... \\ ... & ... & [AS4R1, AS2R2, AS3R1] & ... \end{pmatrix}$$

If the Multiple Router Pair Matrix MP is:

$$MP = \begin{array}{c} \\ AS-1 \\ AS-2 \\ AS-3 \\ AS-4 \end{array} \begin{array}{c} \begin{array}{cccc} AS-1 & AS-2 & AS-3 & AS-4 \end{array} \\ \begin{pmatrix} 0 & [3,2][2,1] & [3,1][2,3] & [3,2][2,1] \\ [2,3][1,2] & 0 & [2,1][4,2] & [2,1][1,2] \\ [1,3][3,2] & [1,2][2,4] & 0 & 0 \\ [2,3][1,2] & [1,2][2,1] & 0 & 0 \end{pmatrix} \end{array}$$

The whole simulation process looks like this:

$$\mathrm{L} = P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_0 + \mathrm{SP} =$$
$$\begin{pmatrix} ... & [AS1R3, AS2R2][AS1R2, AS2R1] & ... & ... \\ ... & 0 & ... & ... \\ ... & [AS3R1, AS2R2][AS3R2, AS2R4] & ... & ... \\ ... & [AS4R1, AS2R2][AS4R2, AS2R1] & ... & ... \end{pmatrix}$$

$$\mathrm{L} \otimes P_0 = P_1 = \begin{pmatrix} 0 & 0 & [1,2,3] & [1,2,4] \\ 0 & 0 & [2,1,3] & 0 \\ [3,2,1] & [3,1,2] & 0 & [3,2,4] \\ [4,2,1] & 0 & [4,2,3] & 0 \end{pmatrix}$$

$$\mathrm{RT} = \mathrm{RT} + P_1 + \mathrm{SP} = \begin{pmatrix} \ldots & & [AS1R3,AS2R2][AS1R2,AS2R1] & & \ldots \ldots \\ \ldots & & 0 & & \ldots \ldots \\ \ldots & [AS3R1,AS2R2][AS3R2,AS2R4][AS3R1,AS1R3,AS2R2,][AS3R2,AS1R2,AS2R1] & & & \ldots \ldots \\ \ldots & & [AS4R1,AS2R2][AS4R2,AS2R1] & & \ldots \ldots \end{pmatrix}$$

Using the router-level path solver, we can generate a routing table between two specific ASes. For the same example, the following two screenshots Figure 4.5 and Figure 4.6 show the routing table with the routing policy mark between AS-2 and AS-3.

There are 2 Routing Paths From AS-2 to AS-3.
2×5 DataFrame

Row	AS_Path	Router_path	Prefer_Shortest	Prefer_Neighbor	Prefer_Local
	Any	Any	Any	Any	Any
1	[2, 3]	AS2-R2 --> AS3-R1	Yes	Yes	Yes
2	[2, 1, 3]	AS2-R2 --> AS1-R3 --> AS3-R1			

Figure 4.5: Screenshot of Single Pair Routing Table.

There are 6 Routing Paths From AS-2 to AS-3.
6×5 DataFrame

Row	AS_Path	Router_path	Prefer_Shortest	Prefer_Neighbor	Prefer_Local
	Any	Any	Any	Any	Any
1	[2, 3]	AS2-R2 --> AS3-R1	Yes	Yes	Yes
2		AS2-R4 --> AS3-R2			
3	[2, 1, 3]	AS2-R2 --> AS1-R3 --> AS3-R1			
4		AS2-R2 --> AS1-R3 --> AS1-R2 -->…			
5		AS2-R1 --> AS1-R2 --> AS1-R3 -->…			
6		AS2-R1 --> AS1-R2 --> AS3-R3			

Figure 4.6: Screenshot of Multiple Pair Routing Table.

From Figure 4.5, we find 2 router paths and 2 AS paths between AS-2 and AS-3. But in Figure 4.6, we find 6 router paths and 2 AS paths between AS-2 and AS-3.

Let's find router paths for AS path [2,3] in these two figures.

In Figure 4.5, we find only one router path: $AS2-R2 \longrightarrow AS3-R1$ for AS path [2,3]. Because in the Single Router Pair Matrix SP, $SP_{2,3}$ is [2,1]. In Figure 4.6, we find two router paths: $AS2-R2 \longrightarrow AS3-R1$ and $AS2-R4 \longrightarrow AS3-R2$ for AS path [2,3]. Because in the Multiple Router Pair Matrix MP, $MP_{2,3}$ is [2,1] and [4,2].

4.2 Path Prefix Attack

In our original routing path solver, each node of the network announces a unique IP prefix. In the path prefix attack model, we select a node to act maliciously and pretend to be the origin of a prefix that belongs to a different node. The path prefix attack process is depicted in Figure 4.7.

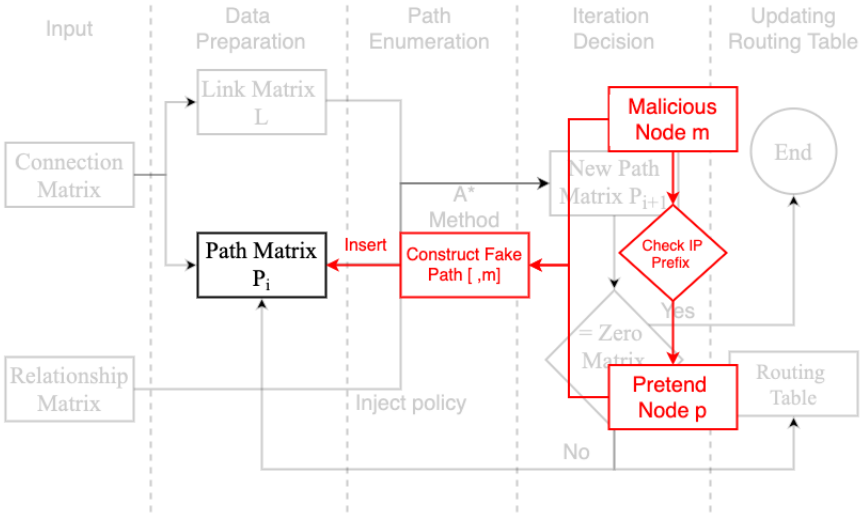


Figure 4.7: Process of Path Prefix Attack.

In order to explain this process, we use the same example in Section 4.1.1. And we make a simple assumption that node 4 pretends its prefix belongs to node 3. In other words, node 4 has the same prefix as node 3.

$$\text{Link Matrix } L = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\text{original Path Matrix } P_0 = \begin{pmatrix} 0 & [1,2] & [1,3] & [1,4] \\ [2,1] & 0 & [2,3] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

$$\text{new Path Matrix } P_0 = \begin{pmatrix} 0 & [1,2] & [1,3][1,4] & [1,4] \\ [2,1] & 0 & [2,3][2,4] & [2,4] \\ [3,1] & [3,2] & 0 & 0 \\ [4,1] & [4,2] & 0 & 0 \end{pmatrix}$$

The only difference with the general path solver process, Link Matrix L does not equal Path Matrix P_0. We build a new Path Matrix P_0 in the path prefix attack process by inserting fake paths.

We analyze the network depicted to find the most vulnerable nodes for a path origin attack. For each network node, we run experiments in which one of the other network nodes performed a path origin attack impersonating it. We also counted every node result and calculated the total attack percentage of the network. The superficial result displays in the following Figure 4.8.

We also analyze the successful attack percentage based on the AS type. The following Figure 4.9. shows an example of the attack statistics result.

Latin Multiplication Method Result With Attack
Network Size: 38
Total of Attack: 1406
Attack Successful: 796
Attack Successful Percentage: 0.566145092460882

Row	Attack_AS Int64	Pretend_AS Int64	Pretend_Successful_Path Int64	Total_Path Int64	Attack_Successful_Path Int64	Attack_Successful String	Elapsed_Time Float64
1	1	2	4	3549	27	Yes	0.00778567
2	1	3	4	3522	0	No	0.00116126
3	1	4	4	3935	413	Yes	0.00843995
4	1	5	4	4064	542	Yes	0.0181351
5	1	6	4	3859	337	Yes	0.00733782
6	1	7	4	3771	249	Yes	0.00687877
7	1	8	4	3771	249	Yes	0.00681393
8	1	9	4	3771	249	Yes	0.00685095
9	1	10	4	3771	249	Yes	0.00688471
10	1	11	4	3771	249	Yes	0.00683292
11	1	12	4	3725	203	Yes	0.00493584
12	1	13	4	3771	249	Yes	0.00682305
13	1	14	4	3771	249	Yes	0.00683352
14	1	15	4	3725	203	Yes	0.00494506
15	1	16	4	3771	249	Yes	0.00684362
16	1	17	4	3771	249	Yes	0.00726772
17	1	18	4	3771	249	Yes	0.0127992
18	1	19	4	3771	249	Yes	0.00680272
19	1	20	4	3771	249	Yes	0.00684899
20	1	21	4	3771	249	Yes	0.00680106
21	1	22	4	3771	249	Yes	0.00681214

Figure 4.8: Screenshot of Attack Result.

Attack Statistics Result
Network Size: 38

Row	Name String	Total_Attack Int64	Attack_Successful Int64	Attack_Successful_Percentage Float64
1	Tier_1 Attack	37	36	0.972973
2	Tier_X Attack	185	184	0.994595
3	Stubs_att Attack	1184	576	0.486486
4	Tier_1 Attack Tier_X	5	4	0.8
5	Tier_1 Attack Stubs	32	32	1.0
6	Tier_X Attack Tier_1	5	5	1.0
7	Tier_X Attack Stubs	160	160	1.0
8	Stubs Attack Tier_1	32	32	1.0
9	Stubs Attack Tier_X	160	160	1.0

Figure 4.9: Screenshot of Attack Statistics.

Chapter5

TEST CASES AND SIMULATION RESULTS

In the previous chapters, we have presented the design and implementation of the routing path solver, router-level path solver, path prefix attack, and All Pairs Monte Carlo method. This chapter will present all the test cases we used in the project. The hardware and software infrastructure that supports the running code is also described in this chapter. We also summarized and visualized the results in Figure. At last, we provide all simulation results in the Table listed in Appendix B.

5.1 All Test Cases

We tested our path solver on ten random network models of differing sizes, and descriptions of these are shown in Table 5.1.

Table 5.1: Path Solver Test Cases

Case	Network Size	Number of Tier-1	Number of Tier-X	Number of Stubs
1	38	1	5	32
2	94	5	10	79
3	219	10	25	184
4	406	15	50	341
5	688	10	100	578
6	1031	15	150	866
7	1656	15	250	1391
8	2281	15	350	1916
9	3281	25	500	2756
10	4844	25	75	4744

We also generated eight random network topologies of differing sizes with different density to compare the All Pairs Monte Carlo method and the A* Algorithm. These test cases are shown in Table 5.2 and the adjacency matrix is given in Appendix A.

The density of the network defines the ratio of connected nodes and total nodes.

Table 5.2: Monte Carlo Method Test Cases

Case	Network Size	Density
1	4	0.375
2	4	0.75
3	8	0.375
4	8	0.875
5	12	0.354
6	12	0.868
7	16	0.340
8	16	0.719

5.2 Hardware and Software Configuration

All programs are written in Julia language, and the hardware configuration used in our experiments list in the following Table 5.3. The only required software is Julia. For Julia, Software Version V1.0.1 or higher is required. We ran our experiments on a macOS Mojave Version 10.14.5 system for the Local Machine, CentOS Linux 7 for the Pascal, and Red Hat Enterprise Linux 8.3 for the Raj Server. But, there is no specific OS requirement.

Raj Server is Marquette's centrally managed HPC cluster to promote research and scholarship. The Raj Server brought online in April 2021. Raj Server has three main components: the nodes (or servers/computers), the storage system and the network. Raj has several classes of nodes including a login node, a management node, general compute nodes, large memory compute nodes, massive memory compute nodes, GPU compute nodes and AI/ML nodes.[6] The general compute nodes are the backbone of the system providing resources for traditional CPU intensive computations. We use the general compute nodes to perform our program in our project.

The primary method for scheduling and executing tasks on Raj is Slurm (Simple Linux Utility for Resource Management). Because Raj is a campus-wide resource with a limited size of CPUs, users are not allowed to run programs or conduct simulations directly on compute nodes as they might on a private cluster such as Pascal Server.

Table 5.3: Hardware Configuration

	Local	Pascal Server	Raj Server
Processor	Inter Core I5	Xeon(R) Gold 5118	AMD EPYC 7702
Processor Speed	3.1 GHz	2.3GHz	2.0GHz
Total Number of Cores	2	48	128

5.3 Path Solver Experiments Result

We deploy three different versions of the routing path solver in Julia, as we have described in Section 4.1 on three different running environments. In the deployment, we evaluate the execution time and memory usage. We also assess the performance for two parallel versions when the number of cores is 4,8,16, and 32. In this section, we visualize all results in figures.

5.3.1 Serial Version Result

This section shows the general routing path solver with the serial implementation with Julia build-in array results on the different running environments. Figure 5.1 displays execution time and memory usage results on the local machine in the first six test cases. Figure 5.2 presents execution time and memory usage results on Pascal Server with a maximum topology of 2281 ASes. Figure 5.3 shows execution time and memory usage results on Raj Server with a maximum topology of 3281 ASes.

These three figures indicated that execution time and memory usage increase as the network size increases, no matter which running environment is used.

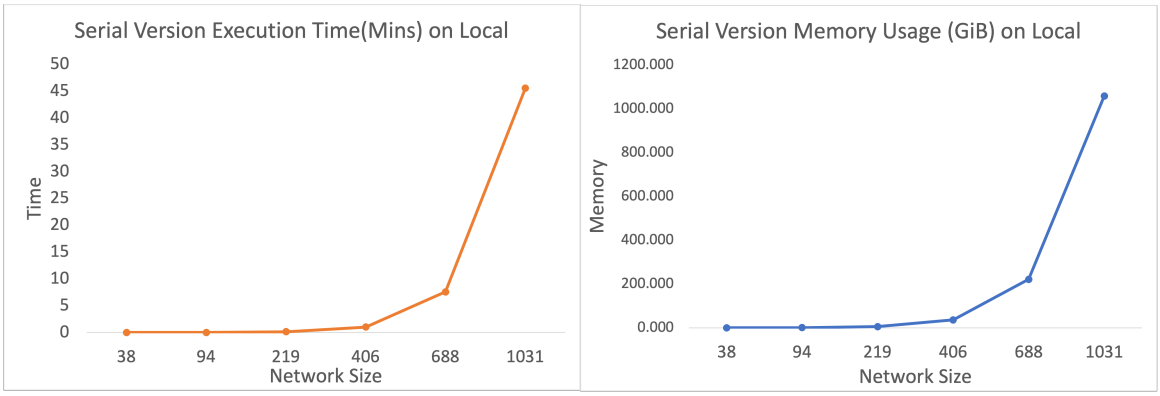


Figure 5.1: Serial Version Result on Local Machine.

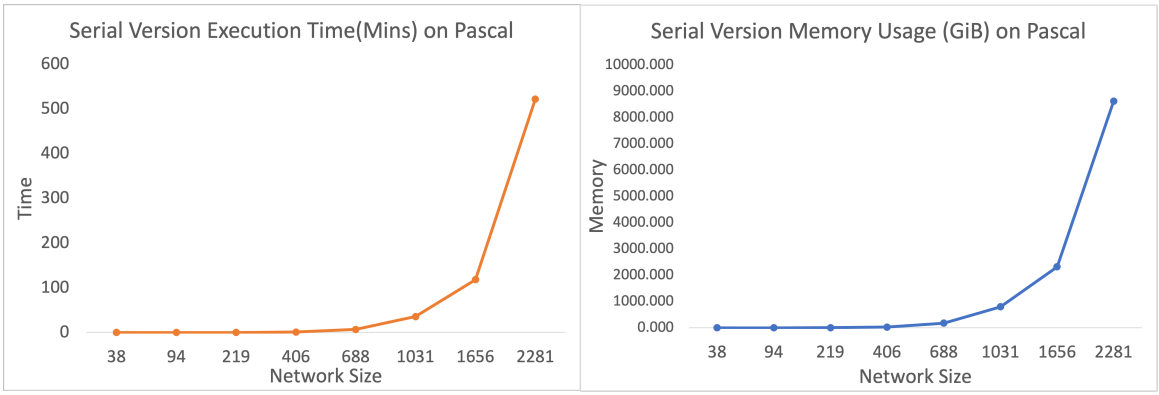


Figure 5.2: Serial Version Result on Pascal Server.

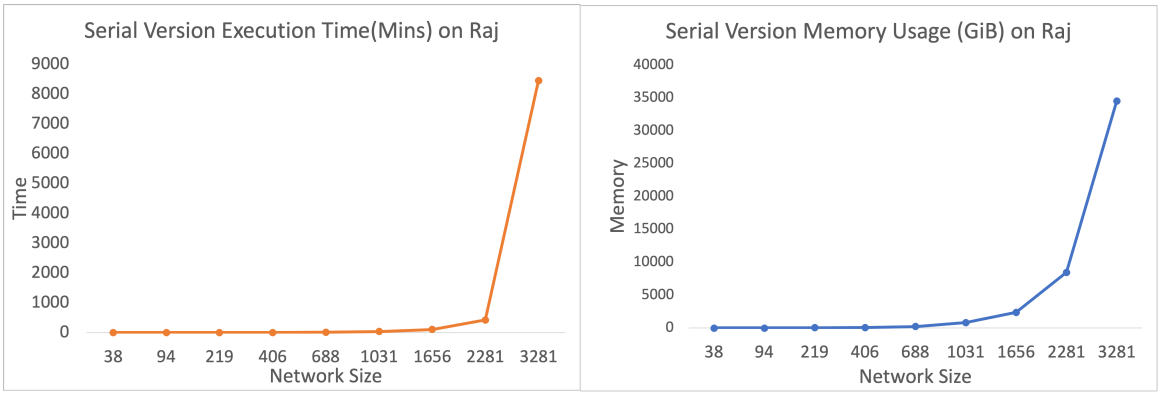


Figure 5.3: Serial Version Result on Raj Server.

5.3.2 Parallel Version Result

This section shows the general routing path solver in the parallel implementation with Julia build-in array results on the different running environments with 4,8,16, and 32 cores. Figure 5.4 displays execution time and memory usage results on the local machine in the first six test cases. Figure 5.5 and Figure 5.6 present execution time and memory usage results on Pascal Server and Raj Server with a maximum topology of 4844 ASes.

With these figures, we can see that memory usage is linearly related to network size and core usage no matter which running environment is used. Execution time result on Pascal Server and Raj Server has a linear relationship with network size and core usage, but not on a local machine.

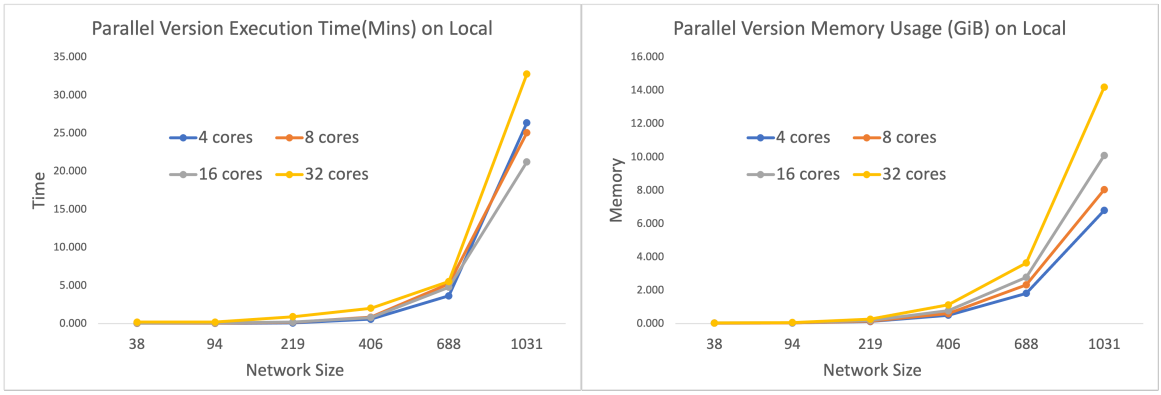


Figure 5.4: Parallel Version Result on Local Machine.

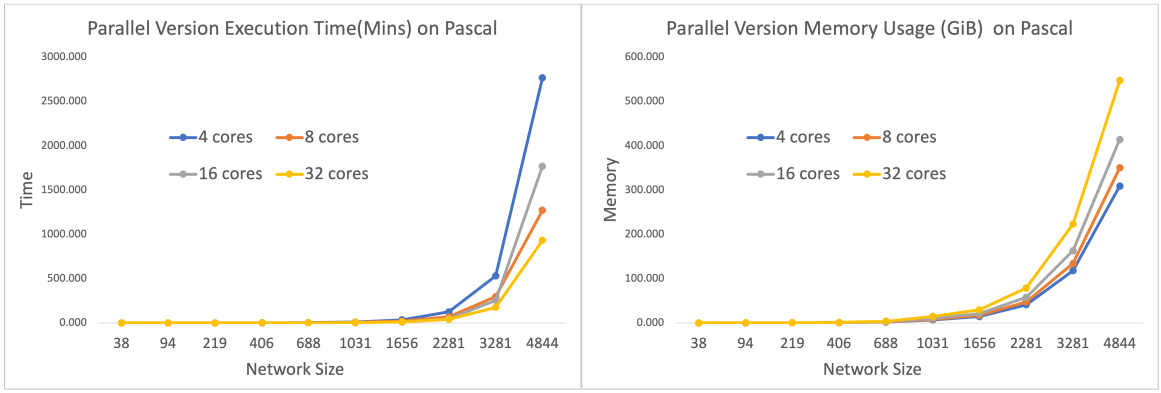


Figure 5.5: Parallel Version Result on Pascal Server.

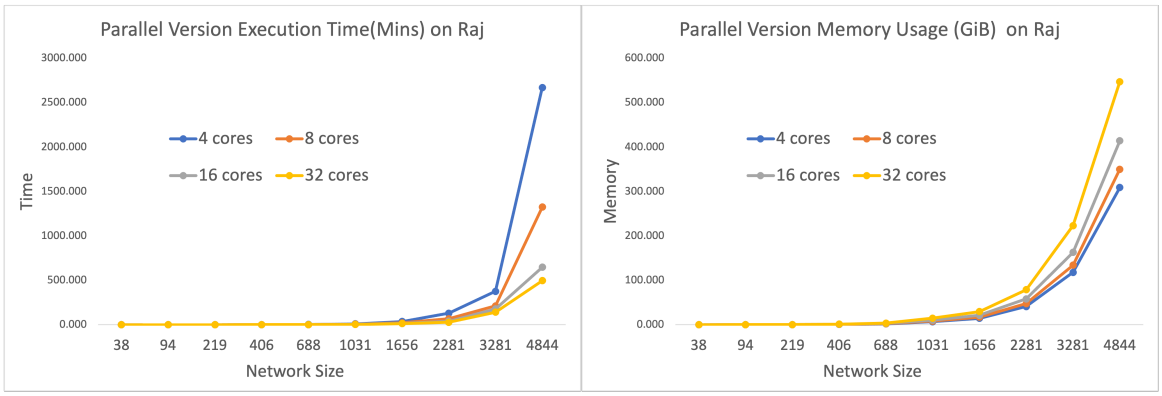


Figure 5.6: Parallel Version Result on Raj Server.

5.3.3 Shared-Array Version Result

This section shows the general routing path solver in the parallel implementation with Julia Shared-Array results on the different running environments with 4,8,16, and 32 cores. Figure 5.7 displays execution time and memory usage results on the local machine in the first five test cases. Figure 5.8 and Figure 5.9 present execution time and memory usage results on Pascal Server and Raj Server with a maximum topology of 4844 ASes.

From Figure 5.7 we can easily see that the best results are achieved when 4 cores are used, and the worst effects are obtained when 32 cores are used. This is an interesting result, and we will discuss this situation in section 6.2.

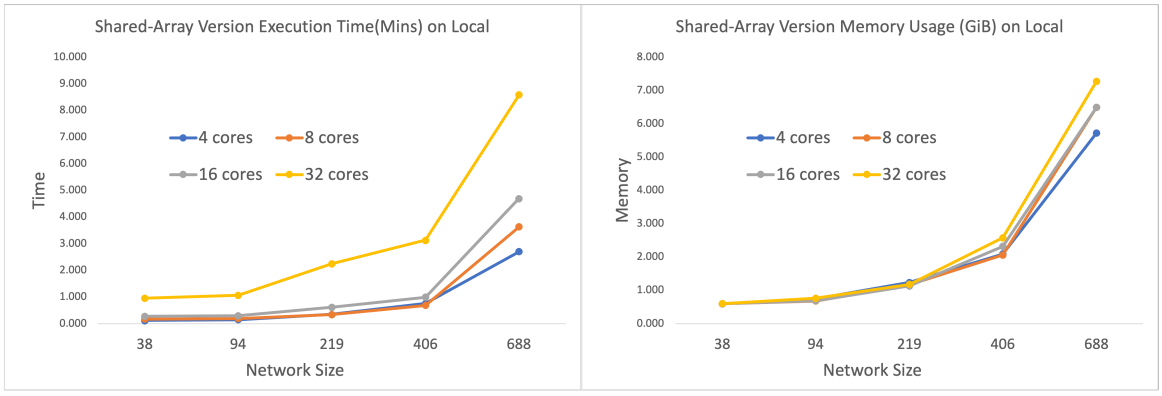


Figure 5.7: Shared-Array Version Result on Local Machine.

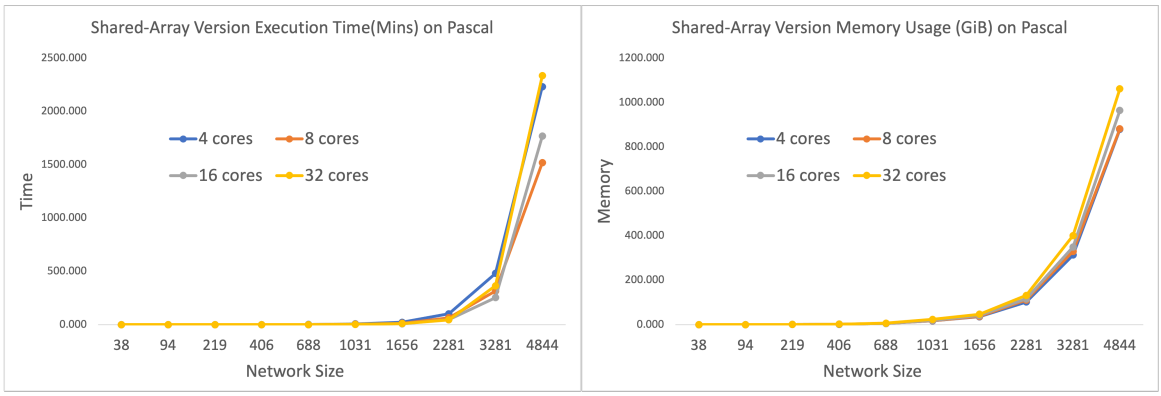


Figure 5.8: Shared-Array Version Result on Pascal Server.

Figure 5.9: Shared-Array Version Result on Raj Server.

5.4 Router Level Simulation Result

From the Figure 5.5, Figure 5.6, Figure 5.8, Figure 5.9 and Table B.5, Table B.7, Table B.11, Table B.13, it is easy to find the parallel implementation using the Julia built-in array performed significantly better than the shared-array parallel implementation on Raj Server with 16 cores or 32 cores. We will visualize the compare result in following Section 6.

So, in this section, we performed the router level simulation on Raj Server with 16 cores using the Julia built-in array. The execution times with the first five test cases are shown in Table 5.4. We also compared the result between the router level path solver and general path solver and visualized the result in Figure 5.10.

Table 5.4: Router Level Simulation Result

Case	Network Size	Execution Time (Sec)	Router Table Generation Time (Sec)
1	38	2.31	0.698
2	94	2.73	0.702
3	219	7.00	0.842
4	406	28.50	0.767
5	688	141.00	0.848

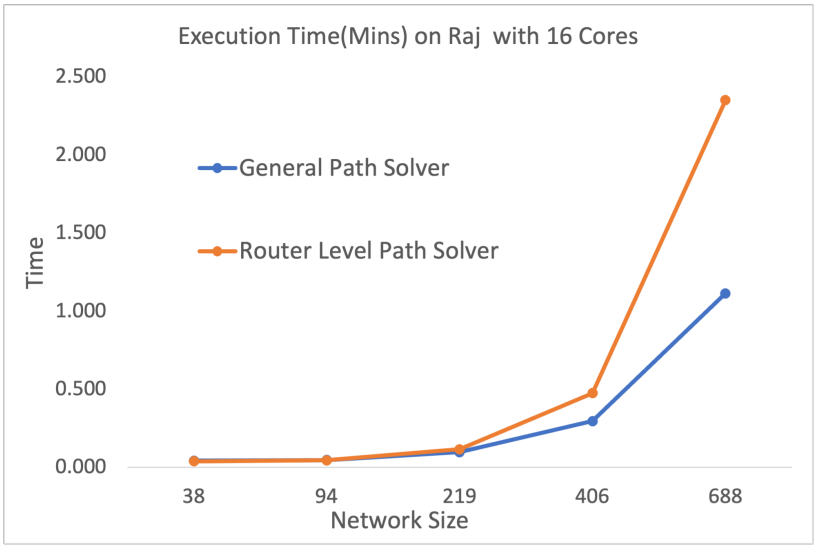


Figure 5.10: Compared with General Path Solver.

5.5 Path Prefix Attack Result

We analyzed the network depicted in Figure 5.11 to find the most vulnerable nodes for a path prefix attack. For each of the nodes of the topology, which are represented by one row in Table 5.5, we run seven experiments in which one of the other nodes of the topology performed a path prefix attack impersonating it. We then measured how many nodes changed their best path selection to select the malicious node instead of the legitimate one. The aggregate result from all seven experiments represented by one row in Table 5.5 is reported in the last column.

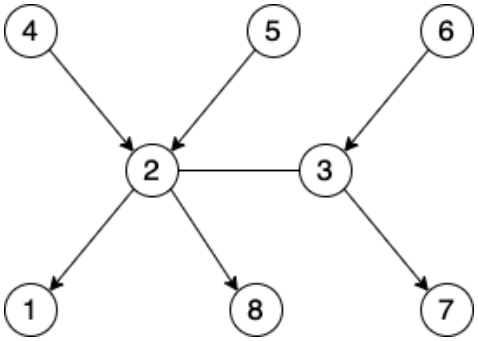


Figure 5.11: Eight node topology.

Table 5.5: Eight Node Network Change in Best Path

	Without Malicious Node Total Path	With Malicious Node Total Path	Total Best Path Change	Percentage of Best Path Change
1	336	372	15	41.6%
2	336	372	11	30.5%
3	280	310	16	53.3%
4	168	186	14	77.7%
5	168	186	14	77.7%
6	112	124	11	91.6%
7	280	310	18	60.0%
8	336	372	15	41.6%

We also analyzed the first three small network models described in Table 5.1. The execution times and attack statistics result are shown in Table 5.6.

Table 5.6: Attack Simulation Result

Network Size	Execution Time(Sec)	Toal of Attack	Atack Succcessful	Atack Succcessful Percentage
38	2.31	1406	796	0.567
94	804	8742	8399	0.960
219	96370	47742	47166	0.988

In the path prefix attack simulation process, we also analyzed path prefix attack based on the AS type. The following Figure 5.12, Figure 5.13, and Figure 5.14 show the origin attack statistics based on the AS type.

Network Size: 38
9×4 DataFrame

Row	Name String	Total_Attack Int64	Attack_Successful Int64	Attack_Successful_Percentage Float64
1	Tier_1 Attack	37	36	0.972973
2	Tier_X Attack	185	184	0.994595
3	Stubs_att Attack	1184	576	0.486486
4	Tier_1 Attack Tier_X	5	4	0.8
5	Tier_1 Attack Stubs	32	32	1.0
6	Tier_X Attack Tier_1	5	5	1.0
7	Tier_X Attack Stubs	160	160	1.0
8	Stubs Attack Tier_1	32	32	1.0
9	Stubs Attack Tier_X	160	160	1.0

Figure 5.12: Origin Attack Statistics with 38 Node Network.

Network Size: 94
9×4 DataFrame

Row	Name String	Total_Attack Int64	Attack_Successful Int64	Attack_Successful_Percentage Float64
1	Tier_1 Attack	465	465	1.0
2	Tier_X Attack	930	930	1.0
3	Stubs_att Attack	7347	7004	0.953314
4	Tier_1 Attack Tier_X	50	50	1.0
5	Tier_1 Attack Stubs	395	395	1.0
6	Tier_X Attack Tier_1	50	50	1.0
7	Tier_X Attack Stubs	790	790	1.0
8	Stubs Attack Tier_1	395	395	1.0
9	Stubs Attack Tier_X	790	790	1.0

Figure 5.13: Origin Attack Statistics with 94 Node Network.

Network Size: 219
9×4 DataFrame

Row	Name String	Total_Attack Int64	Attack_Successful Int64	Attack_Successful_Percentage Float64
1	Tier_1 Attack	2180	2180	1.0
2	Tier_X Attack	5450	5450	1.0
3	Stubs_att Attack	40112	39536	0.98564
4	Tier_1 Attack Tier_X	250	250	1.0
5	Tier_1 Attack Stubs	1840	1840	1.0
6	Tier_X Attack Tier_1	250	250	1.0
7	Tier_X Attack Stubs	4600	4600	1.0
8	Stubs Attack Tier_1	1840	1840	1.0
9	Stubs Attack Tier_X	4600	4600	1.0

Figure 5.14: Origin Attack Statistics with 219 Node Network.

5.6 Monte Carlo Method Result

In this section, we performed our All Pairs Monte Carlo Method described in Section 3.6.2 with the network model described in Table 5.2. The execution times and estimated number of paths are shown in Table 5.7 and Table 5.8.

Table 5.7: Monte Carlo Simulation Execution Time Result

Case	Network Size	Density	LM Method Execution Time(Sec)	All Pairs Monte Carlo Execution Time(Sec)
1	4	0.375	1.00	0.69
2	4	0.75	0.70	0.73
3	8	0.375	0.397	0.736
4	8	0.875	1.18	0.94
5	12	0.354	0.69	1.33
6	12	0.868	139111	228
7	16	0.340	2292	54.9
8	16	0.719	303910	321

Table 5.8: Monte Carlo Simulation Number of Path Result

Case	Network Size	Density	Actual Number of Path	Estimating Number of Path	Estimating Accuracy
1	4	0.375	12	11.98	99.9%
2	4	0.75	60	60	100%
3	8	0.375	554	554.84	99.8%
4	8	0.875	109592	109592.00	100%
5	12	0.354	58543	58475.95	99.8%
6	12	0.868	748355761	748458193	99.9%
7	16	0.340	59499502	59633331	99.7%
8	16	0.719	135305299183	135293363754	99.9%

Chapter6
SIMULATION RESULT DISCUSSION

This chapter compares different version path solver results with the same running environments and discusses single version path solver in different running environments. This section used parallel performance metrics to analyze and investigate the A* algorithm's performance and scale-ability.

The parallel execution time is defined as the time that elapses from the moment a parallel computation starts to when the last processor finishes execution. So, we use T_s to represent the serial execution time and T_p to represent the parallel execution time.

The speedup is defined as the ratio of the serial run time of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on p processors.

6.1 Compare Different Running Environments

According to the results in Section 5.3, we visualized result in Figure 6.1, Figure 6.2 and Figure 6.3.

From Figure 6.1, we can find that Raj Server has better performance than Pascal Server and Local at serial version. Raj Server performs better and better as the network size grows. This result also reflects the hardware configuration in Table 5.3.

Figure 6.2 and Figure 6.3 show that Raj Server does not always have better performance than Pascal Server and Local. Pascal Server performs better at parallel and shared-array versions with 4 and 8 cores. The efficiency of the Pascal Server decreases significantly with increased cores usage and network size. Raj Server has better performance at 16 and 32 cores with a large network.

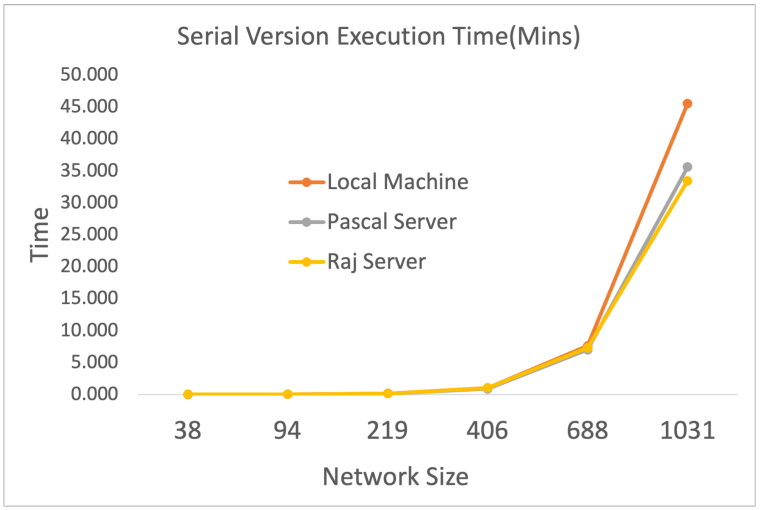


Figure 6.1: Serial Version Execution Time(Mins).

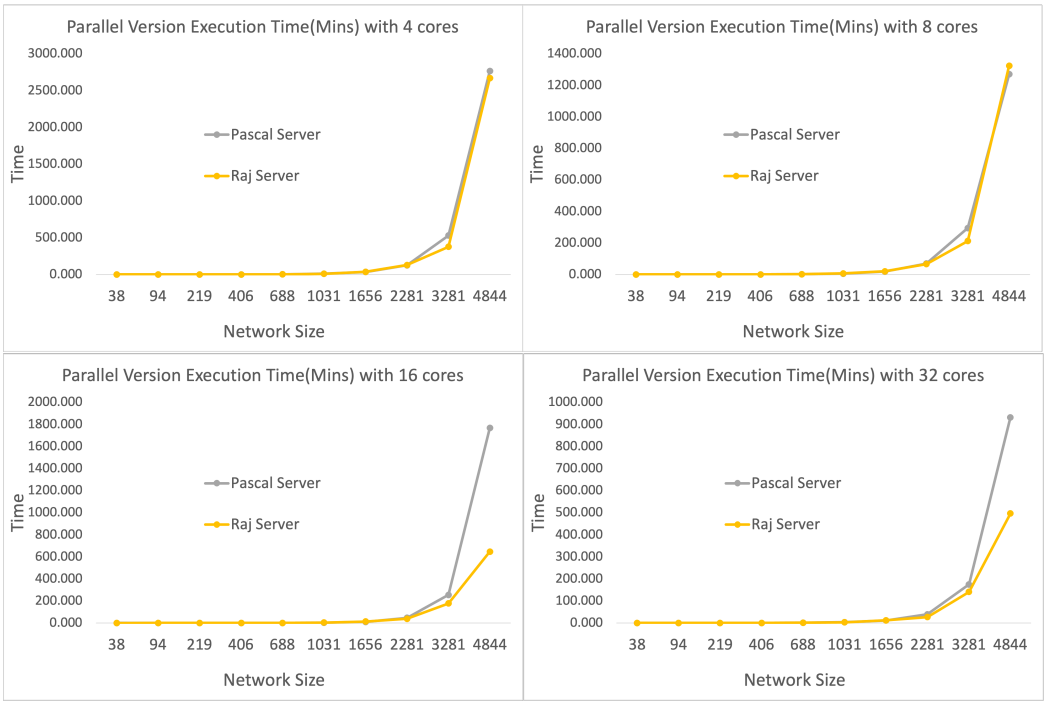


Figure 6.2: Parallel Version Execution Time(Mins) with different cores.

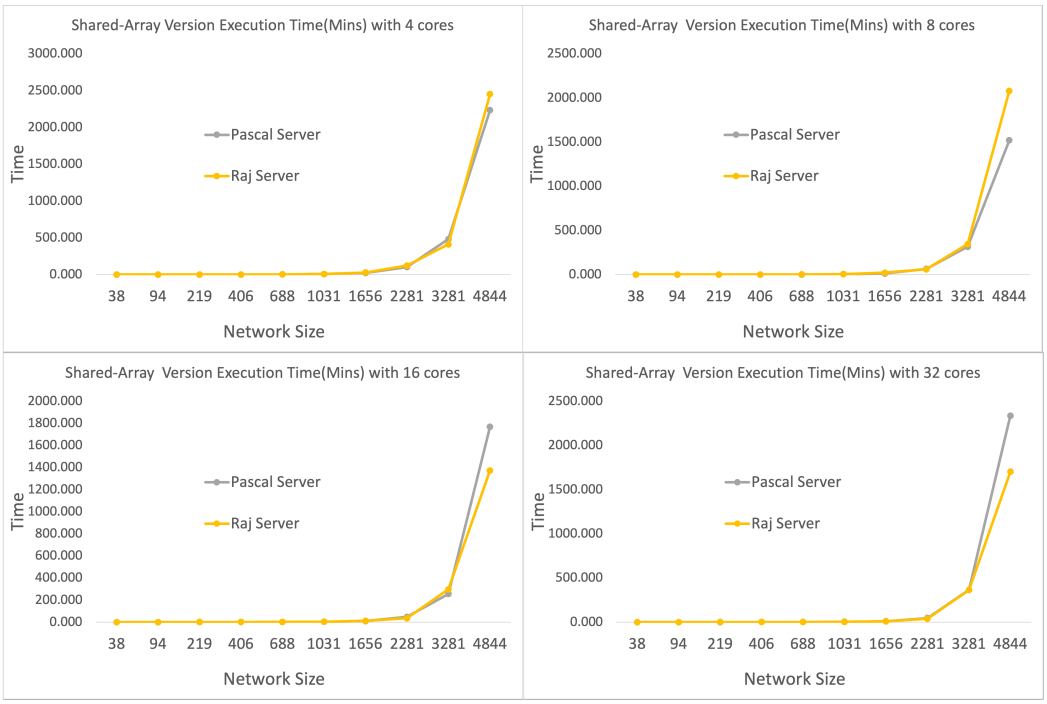


Figure 6.3: Shared-Array Version Execution Time(Mins) with different cores.

6.2 Two Parallel Version on Different Environments

According to the results in Section 5.3.2 and Section 5.3.3, we visualized two parallel version result in Figure 6.4 and Figure 6.5

Figure 6.4 shows the parallel version with Julia build-in array results on different running environments. Typically, the execution time decreases as increases of the number of cores. In this figure, we prove this point.

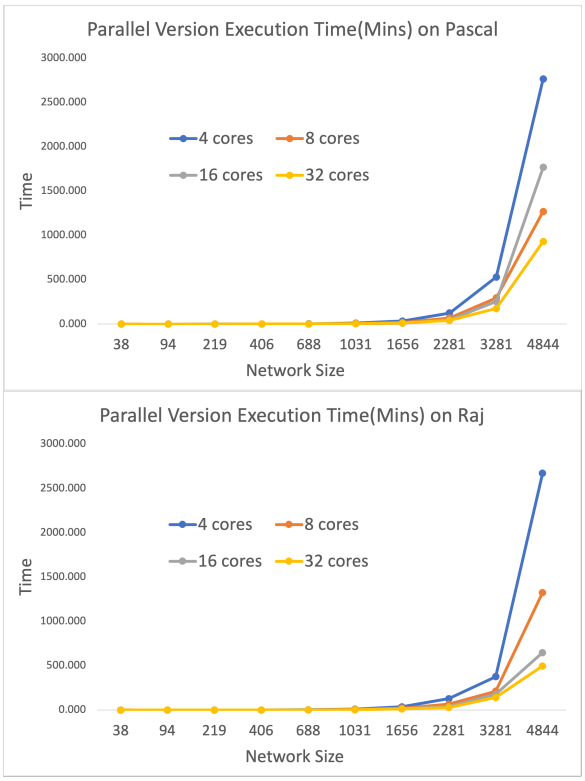


Figure 6.4: Parallel Version Execution Time(Mins).

Figure 6.5 shows the parallel version with Julia shared-array results on different running environments. In the figure, we find that on Pascal Server and Raj Server, when the network size is 4844, the shared-array version with 32 cores speed more time than other cores. We call this phenomenon overhead. To understand this overhead phenomenon, we compute speedup for shared-array results.

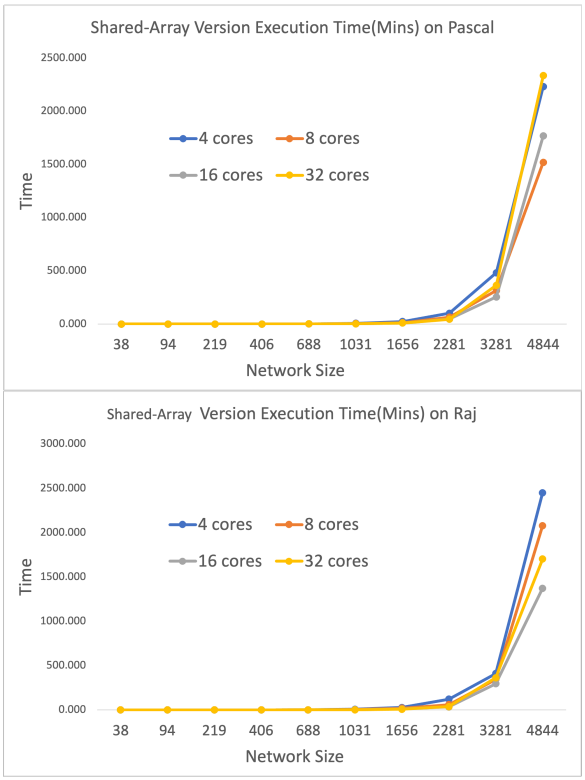


Figure 6.5: Shared-Array Version Execution Time(Mins).

Figure 6.6 shows speedup result with 16 cores and 32 cores . The speedup depend on problem size and number of processes. Typically, the speed up increases as increases of problem size and number of processes. When the maximum speedup is achieved, the speedup will decrease as number of processes increase. And we call this is overhead.

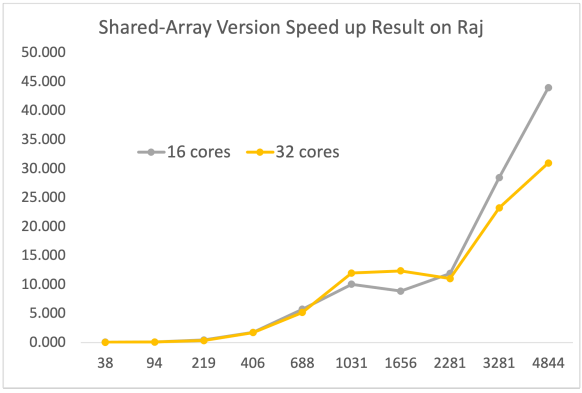


Figure 6.6: Computed Speedup.

We find the overhead phenomenon is obvious on Raj Server at network size is 3281 and 4844 , the overhead arises because shared memory has to be used for communication and synchronization.

In the Figure 6.7, it is clear that the sharp increase in memory usage at network size is 3281 and 4844.

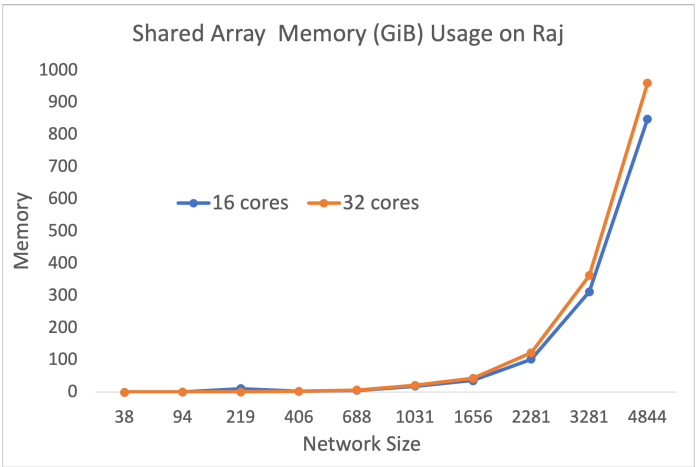


Figure 6.7: Shared Array Memory Usage on Raj

6.3 Two Parallel Version on Same Environments

In Section 6.2, we visualized two parallel version results on different running environments. So, in this section, we will compare two parallel version results in the same environments.

Figure 6.8 and Figure 6.9 visualized two parallel version execution time compare on the Pascal Server and Raj Server.

From Figure 6.8 and Figure 6.9, we find that only with 4 cores at a network size is 4844, shared-array parallel implementation has better performance. So, we can say that the parallel implementation using the Julia built-in array performed significantly better than the shared-array parallel implementation.

The reason for this situation is because of Bit-type. In Section 4.1.2, we discussed the Bit type and Shared-Array data structure. The original Shared-Array takes ad-

vantage of the shared memory parallel programming model across all processes, but this advantage is not shown in our method.

To use Shared-Array, we declared a new Bit-type data structure *Path*, and we need to do a type convert in every single iteration. That processes cost more resources

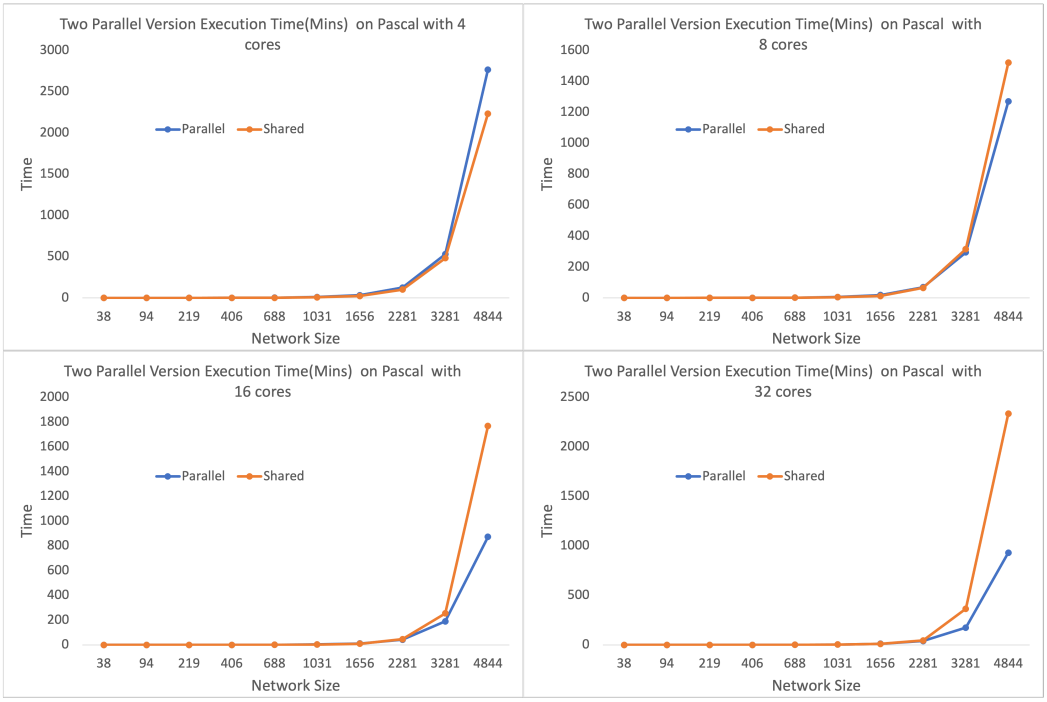


Figure 6.8: Two Parallel Execution Time(Mins) Compare on Pascal.

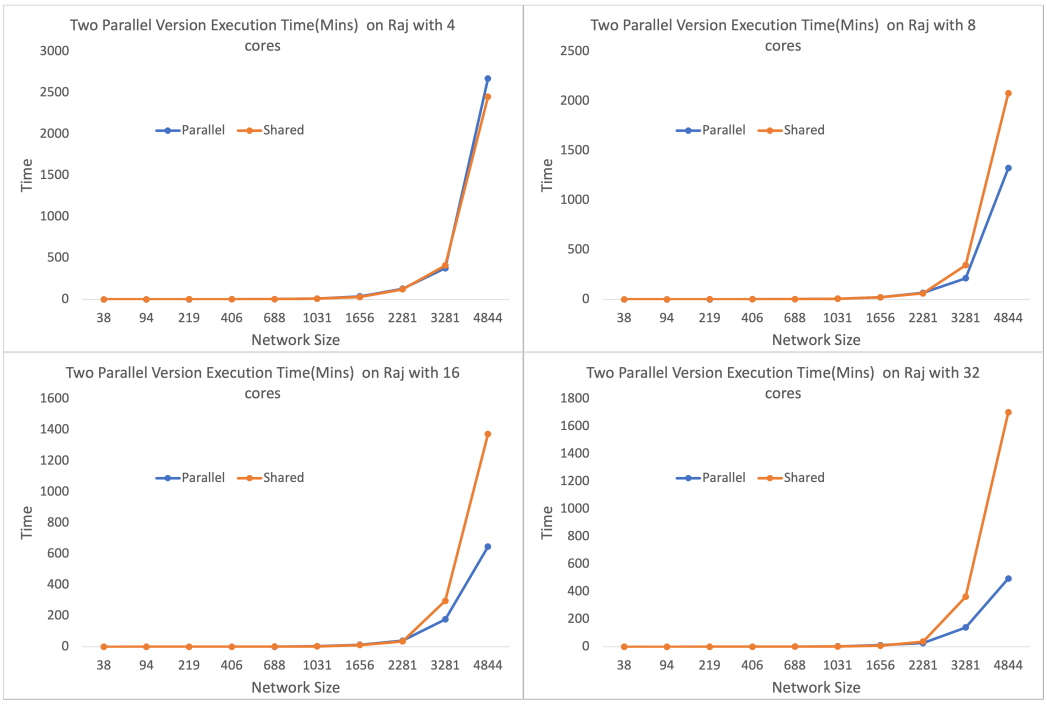


Figure 6.9: Two Parallel Execution Time(Mins) Compare on Raj.

Chapter7
CONCLUSION

In this project, we have presented a new BGP simulator and its implementation. An experiment has been presented to show the performance with different test cases and three different implementations. However, the experiment has not shown an advantage of Shared-Array in Julia programming language.

7.1 Conclusion

To sum up, during this project, we have created an efficient routing simulator for policy-based protocols that enumerates not only the best but all candidate paths between all nodes of the topology. A path enumeration algorithm - A* algorithm was created and used in this project. From the result, the new algorithm proves its reliability and efficiency.

We have experimented with three different versions of the implementation and identified the one with the best overall performance in the Julia programming language. The Julia built-in array's parallel implementation performed significantly better than the shared-array parallel implementation and serial implementation.

We have performed AS level simulations with a maximum of 4844 nodes on three different running environments with three different implementation versions. We identified the one with the best overall performance for experiments from the result.

We have also performed router level simulations in a maximum topology of 1031 ASes many of which have 3-5 routers each. To maximize the performance of router-level simulations, we choose the Raj Server as the running environment. We were implementing a parallel version with the Julia built-in array method with 16 cores. During the router level simulations process, we generated a router table with all

possible paths and identified paths with its business policy.

We have run experiments exploring the effect of a path prefix attack at the best path selections of the nodes. We have also performed path prefix attack simulation with a maximum topology of 219 ASes and calculated the attack success percentage for each network topology. We also analyzed path prefix attacks based on the AS type.

Finally, we have run Monte Carlo Method experiments with 6 test cases and compared our All Pairs Monte Carlo method with the A* algorithm.

7.2 Future Work

As mentioned during this project, several features could be refined in this project if there was more time. First of all, we plan to improve the performance of our algorithm further to improve scalability. We will experiment on clusters with larger networks and real-world cases.

Furthermore, try to run an experiment with more diverse sets of business policies. In this project, we only run an experiment with a simple twob business relationship policy: Valley-free and loop. We also performed three business policies at router level simulations to make path selection.

Finally, We further plan to expand our model such that it is able to represent greater policy diversity within an AS.

AppendixA

Adjacency Matrices of Test Cases

Case 1: (size = 4, Low Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Case 2: (size = 4, High Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Case 3: (size = 8, Low Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Case 4: (size = 8, High Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Case 5: (size = 12, Low Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Case 6: (size = 12, High Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0&1&1&1&1&1&1&0&1&1&1&1\\ 1&0&1&1&1&1&1&1&1&1&1&1\\ 1&1&0&1&1&1&1&1&1&0&1&1\\ 1&1&1&0&1&1&1&1&1&1&0&1\\ 1&1&1&0&0&1&1&1&0&1&1&1\\ 1&1&1&1&1&0&1&1&1&1&1&0\\ 1&1&1&1&1&1&0&1&1&1&1&1\\ 1&1&1&1&1&1&1&0&1&1&1&1\\ 1&1&1&1&1&1&1&1&0&1&1&1\\ 1&1&1&1&1&1&1&1&1&0&1&0\\ 1&1&1&1&1&1&1&1&1&1&0&0 \end{pmatrix}$$

Case 7: (size = 16, Low Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0&1&1&0&0&1&0&0&1&0&0&0&0&0&0&0\\ 0&0&1&1&0&0&0&1&1&0&0&1&0&1&1&1\\ 1&0&0&0&0&0&0&0&0&1&1&0&0&0&0&1\\ 1&1&0&0&1&1&1&1&0&1&1&0&1&0&0&1\\ 0&1&0&0&0&0&1&0&0&1&1&0&0&0&1&0\\ 1&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0\\ 0&0&0&1&0&0&0&0&1&0&0&0&0&1&0&1\\ 1&0&0&0&0&0&0&0&0&1&1&0&0&1&0&0\\ 1&1&0&0&1&0&0&0&0&0&1&0&0&1&0&0\\ 1&1&0&0&1&1&0&0&1&0&0&0&1&0&1&0\\ 0&0&1&0&0&0&0&0&0&1&0&0&0&0&0&1\\ 0&0&1&1&1&1&0&1&0&0&1&0&1&1&0&1\\ 1&1&0&0&1&1&0&0&0&0&0&0&0&1&0&0\\ 1&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1\\ 1&1&0&1&0&0&0&1&0&0&1&0&0&0&0&0 \end{pmatrix}$$

Case 8: (size = 16, High Density)

$$Adjacency\ Matrix\ A\ =\ \begin{pmatrix} 0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1\\ 0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1\\ 0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1\\ 0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1\\ 1&1&1&1&0&1&1&1&1&1&1&0&0&0&0&0\\ 1&1&1&1&1&0&0&0&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&0&0&0&1&1&1&1&1&1&1&1\\ 1&1&1&1&1&0&0&0&1&1&1&0&0&1&1&1\\ 1&1&1&1&1&0&0&1&0&0&1&1&1&1&0&0\\ 1&1&1&1&1&1&1&1&1&0&0&0&1&0&1&1\\ 1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&1\\ 1&1&1&1&1&1&1&0&1&0&1&0&1&1&0&1\\ 1&1&1&1&1&1&1&1&0&1&0&0&0&0&1&0\\ 1&1&1&1&0&1&1&1&1&0&0&1&0&0&1&1\\ 1&1&1&1&0&1&1&1&0&0&0&1&1&0&0&1\\ 1&1&1&1&0&1&1&1&0&1&1&1&1&1&1&0 \end{pmatrix}$$

AppendixB

Simulation Results Tables

B.1 Serial Version Result

This section shows routing path solver in the serial version with Julia build-in array results on the different running environments. Table B.1 and Table B.2 displays execution time and memory usage result with its running environment with first 6 test case.

Table B.1: Execution Time in Seconds

Case	Network Size	Local	Pascal Server	Raj Server
1	38	0.685	0.723	0.968
2	94	1.220	1.100	1.560
3	219	8.400	8.210	8.980
4	406	61.400	53.600	62.10
5	688	455.000	421.000	438.000
6	1031	2733.000	2136.000	2006.000

Table B.2: Memory Usage in GiB

Case	Network Size	Local	Pascal Server	Raj Server
1	38	0.114	0.048	0.069
2	94	0.503	0.417	0.412
3	219	5.200	4.450	4.440
4	406	36.300	29.800	29.600
5	688	221.000	177.000	192.000
6	1031	1058.000	800.000	789.000

B.2 Parallel Version Result

This section shows routing path solver in the parallel version with Julia build-in array results on the different running environments. Table B.3 and Table B.4 shows the parallel first 6 test cases result on local machine with 4,8,16 and 32 cores. In the Table B.5 and Table B.6, we display parallel result with 4,8,16 and 32 cores for all test cases on Pascal Server. Raj server result with all test cases is shown in Table B.7 and Table B.8.

Table B.3: Execution Time in Seconds on Local Machine

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	3.020	2.430	4.450	12.700
2	94	2.620	3.020	5.250	13.100
3	219	6.100	11.700	12.300	55.500
4	406	36.200	51.900	48.700	122.000
5	688	220.000	314.000	288.000	333.000
6	1031	1583.000	1505.000	1275.000	1967.000

Table B.4: Memory Usage in GiB on Local Machine

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	0.038	0.038	0.040	0.042
2	94	0.047	0.051	0.059	0.073
3	219	0.129	0.153	0.193	0.269
4	406	0.511	0.606	0.784	1.130
5	688	1.820	2.330	2.790	3.650
6	1031	6.800	8.050	10.100	14.200

Table B.5: Execution Time in Seconds on Pascal Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	1.920	1.870	2.140	2.060
2	94	2.270	2.190	2.170	2.640
3	219	6.060	5.050	5.560	7.270
4	406	23.300	18.200	16.700	23.000
5	688	135.000	84.900	65.100	69.700
6	1031	607.000	324.000	218.000	226.000
7	1656	2014.000	1103.000	685.000	692.000
8	2281	7469.000	4140.000	2539.000	2388.000
9	3281	31788.000	17678.000	11391.000	10490.000
10	4844	165927.000	76166.000	52394.000	55874.000

Table B.6: Memory Usage in GiB on Pascal Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	0.036	0.039	0.040	0.043
2	94	0.049	0.053	0.058	0.073
3	219	0.128	0.154	0.193	0.274
4	406	0.485	0.620	0.778	1.130
5	688	1.980	2.280	2.910	3.630
6	1031	6.760	7.960	10.100	14.600
7	1656	14.500	16.900	21.100	29.600
8	2281	40.900	47.600	57.900	78.700
9	3281	118.000	134.000	163.000	223.000
10	4844	308.000	350.000	414.000	547.000

Table B.7: Execution Time in Seconds on Raj Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	2.470	2.460	2.590	2.730
2	94	2.690	2.670	2.780	2.990
3	219	6.470	5.690	5.830	8.150
4	406	28.6	19.700	17.800	24.800
5	688	132.000	88.200	66.800	79.700
6	1031	573.000	385.000	228.000	207.000
7	1656	2130.000	1221.000	818.000	716.000
8	2281	7704.000	3963.000	2343.000	1625.000
9	3281	22599.000	12720.000	10641.000	8447.000
10	4844	160185.000	79419.000	38795.000	29786.000

Table B.8: Memory Usage in GiB on Raj Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	0.039	0.040	0.041	0.042
2	94	0.048	0.052	0.060	0.068
3	219	0.128	0.161	0.195	0.276
4	406	0.483	0.611	0.781	1.070
5	688	1.940	2.180	2.630	3.690
6	1031	6.540	8.190	10.900	14.700
7	1656	14.400	17.200	21.600	30.500
8	2281	40.600	49.400	57.400	78.600
9	3281	118.000	134.000	163.000	223.000
10	4844	331.000	348.000	416.000	546.000

B.3 Shared-Array Version Result

This section shows routing path solver in the parallel version with Julia Shared-Array results on the different running environments. Table B.9 and Table B.10 shows the parallel first 5 test cases result on local machine with 4,8,16 and 32 cores. Pascal server shared-array result with all test cases is shown in Table B.11 and Table B.12. In the Table B.13 and Table B.14, we display shared-array result with 4,8,16 and 32 cores for all test cases on Raj Server.

Table B.9: Execution Time in Seconds on Local Machine

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	7.360	10.200	16.700	57.800
2	94	8.940	11.600	18.000	64.300
3	219	21.200	20.400	36.900	135.000
4	406	45.600	41.200	59.500	188.000
5	688	162.000	218.000	282.000	515.000

Table B.10: Memory Usage in GiB on Local Machine

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	0.596	0.597	0.599	0.602
2	94	0.745	0.681	0.688	0.767
3	219	1.240	1.170	1.130	1.180
4	406	2.090	2.060	2.320	2.580
5	688	5.720	6.490	6.490	7.270
6	1031	16.900	18.300	Error	Error

Table B.11: Execution Time in Seconds on Pascal Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	9.120	9.100	9.590	10.000
2	94	11.700	10.500	10.800	11.500
3	219	18.200	18.300	18.500	19.400
4	406	35.400	31.800	30.300	31.300
5	688	122.000	83.100	67.200	74.700
6	1031	459.000	298.000	208.000	217.000
7	1656	1390.000	779.000	571.000	592.000
8	2281	6114.000	3912.000	2800.000	2723.000
9	3281	28903.000	18949.000	15276.000	21878.000
10	4844	133964.000	91195.000	106104.000	140145.000

Table B.12: Memory Usage in GiB on Pascal Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	0.521	0.522	0.522	0.525
2	94	0.660	0.603	0.606	0.616
3	219	0.990	1.070	1.100	1.170
4	406	1.970	2.060	2.200	2.480
5	688	5.760	5.620	5.930	7.110
6	1031	17.300	18.200	19.400	23.800
7	1656	35.600	37.000	40.900	47.300
8	2281	102.000	111.000	116.000	132.000
9	3281	314.000	330.000	350.000	401.000
10	4844	879.000	882.000	965.000	1062.000

Table B.13: Execution Time in Seconds on Raj Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	10.800	11.200	10.900	13.100
2	94	14.800	14.100	13.300	13.100
3	219	20.800	19.500	19.800	24.800
4	406	41.400	36.900	35.300	35.500
5	688	135.000	96.900	76.100	84.100
6	1031	530.000	340.000	199.000	167.000
7	1656	1666.000	1249.000	687.000	494.000
8	2281	7332.000	3587.000	2111.000	2281.000
9	3281	24653.000	20695.000	17811.000	21810.000
10	4844	147082.000	124721.000	82347.000	102175.000

Table B.14: Memory Usage in GiB on Raj Server

Case	Network Size	4 cores	8 cores	16 cores	32 cores
1	38	0.550	0.550	0.522	0.554
2	94	0.673	0.677	0.783	0.638
3	219	0.960	0.980	1.070	1.220
4	406	1.820	1.900	2.160	2.320
5	688	4.740	4.950	5.310	6.350
6	1031	15.600	15.500	18.100	20.700
7	1656	30.700	35.300	36.100	43.000
8	2281	88.100	92.300	102.000	122.000
9	3281	278.000	284.000	312.000	362.000
10	4844	774.000	794.000	848.000	960.000

Bibliography

[1] C-BGP. http://c-bgp.sourceforge.net/.

[2] Nist rpki monitor. https://rpki-monitor.antd.nist.gov/.

[3] Resource public key infrastructure (rpki). https://www.apnic.net/get-ip/faqs/rpki/.

[4] Ris raw data. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[5] Routeviews. http://www.routeviews.org/routeviews/.

[6] System architecture. https://www.marquette.edu/high-performance-computing/architecture.php.

[7] What is bgp hijacking? https://www.cloudflare.com/learning/security/glossary/bgp-hijacking/.

[8] A Border Gateway Protocol 4 (BGP-4). RFC 1771, Mar. 1995.

[9] Experts detailed how China Telecom used BGP hijacking to redirect traffic worldwide. https://www.cyberdefensemagazine.com/experts-detailed-how-china-telecom-used-bgp-hijacking-to-redirect-traffic-wor Nov. 2018.

[10] Abhashkumar, A., Gember-Jacobson, A., and Akella, A. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 201–219.

[11] Ballani, H., Francis, P., and Zhang, X. A study of prefix hijacking and interception in the internet. SIGCOMM '07, Association for Computing Machinery, p. 265–276.

[12] Bellovin, S., Bush, R., and Ward, D. Security Requirements for BGP Path Validation. RFC 7353, Aug. 2014.

[13] Bezanson, J., Karpinski, S., B. Shah, V., and Alan, E. Why We Created Julia.

[14] Bezanson, J., Karpinski, S., and Edelman, A. The julia language. `https://julialang.org/benchmarks/`.

[15] Butler, K., Farley, T. R., McDaniel, P., and Rexford, J. A survey of bgp security issues and solutions. *Proceedings of the IEEE 98*, 1 (2010), 100–122.

[16] Butler, K., McDaniel, P., and Aiello, W. Optimizing bgp security by exploiting path stability. In *Proceedings of the 13th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2006), CCS '06, Association for Computing Machinery, p. 298–310.

[17] Cho, S., Fontugne, R., Cho, K., Dainotti, A., and Gill, P. Bgp hijacking classification. In *2019 Network Traffic Measurement and Analysis Conference (TMA)* (2019), pp. 25–32.

[18] Chung, T., Aben, E., Bruijnzeels, T., Chandrasekaran, B., Choffnes, D., Levin, D., Maggs, B. M., Mislove, A., Rijswijk-Deij, R. v., Rula, J., and Sullivan, N. Rpki is coming of age: A longitudinal study of rpki deployment and invalid route origins. In *Proceedings of the Internet Measurement Conference* (New York, NY, USA, 2019), IMC '19, Association for Computing Machinery, p. 406–419.

[19] Cohen, A., Gilad, Y., Herzberg, A., and Schapira, M. One hop for rpki, one giant leap for bgp security. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2015), HotNets-XIV, Association for Computing Machinery.

[20] Cohen, A., Gilad, Y., Herzberg, A., and Schapira, M. Jumpstarting bgp security with path-end validation. In *Proceedings of the 2016 ACM SIGCOMM Conference* (New York, NY, USA, 2016), SIGCOMM '16, Association for Computing Machinery, p. 342–355.

[21] Demchak, C. C., and Shavitt, Y. China's maxim–leave no access point unexploited: The hidden story of china telecom's bgp hijacking. *Military Cyber Affairs 3*, 1 (2018), 7.

[22] Dimitropoulos, X. A., and Riley, G. F. Efficient large-scale bgp simulations. *Computer Networks 50*, 12 (Aug 2006), 2013–2027.

[23] Feamster, N., Winick, J., and Rexford, J. A model of bgp routing for network engineering. *SIGMETRICS Perform. Eval. Rev. 32*, 1 (June 2004), 331–342.

[24] Fu, Y. RPKI Deployment Considerations: Problem Analysis and Alternative Solutions. Tech. rep.

[25] Gao, L. On inferring autonomous system relationships in the internet. *IEEE/ACM Trans. Netw. 9*, 6 (Dec. 2001), 733–745.

[26] Gilad, Y., Cohen, A., Herzberg, A., Schapira, M., and Shulman, H. Are we there yet? on rpki's deployment and security. *IACR Cryptol. ePrint Arch. 2016* (2017), 1010.

[27] Gilad, Y., Hlavacek, T., Herzberg, A., Schapira, M., and Shulman, H. Perfect is the enemy of good: Setting realistic goals for bgp security. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks* (New York, NY, USA, 2018), HotNets '18, Association for Computing Machinery, p. 57–63.

[28] Giotsas, V., and Zhou, S. Valley-free violation in internet routing - analysis based on bgp community data. In *2012 IEEE International Conference on Communications (ICC)* (June 2012), pp. 1193–1197.

[29] Gondran, M., and Minoux, M. *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*, 1 ed. Springer Publishing Company, Incorporated, 2008.

[30] Goodin, D. Citing BGP hijacks and hack attacks, feds want China Telecom out of the US, Apr. 2020.

[31] Haberman, B. K. *Routing Information Verification Tool for Securing Inter-Domain Routing Information*. PhD thesis, USA, 2011. AAI3483282.

[32] Hawkinson, J., and Bates, T. Rfc1930: Guidelines for creation, selection, and registration of an autonomous system (as), 1996.

[33] Hlavacek, T., Herzberg, A., Shulman, H., and Waidner, M. Practical experience: Methodologies for measuring route origin validation. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2018), IEEE, pp. 634–641.

[34] Hunt, C. *TCP IP Network Administration*. O'Reilly amp; Associates, 2002.

[35] Johnson, S. G. Introduction to julia:why are we doing this to you? University Lecture, 2017.

[36] Kwon, C. *Julia Programming for Operations Research*. Changhyun Kwon., 2019.

[37] Kwon, C., and Saba, E. Pathdistribution.jl. `https://github.com/chkwon/PathDistribution.jl`, 2021.

[38] Laub, P. Crude monte carlo. `https://www.youtube.com/watch?v=YLOZ3eb7pqU`.

[39] Levy, M. J. Rpki - the required cryptographic upgrade to bgp routing, Sep 2018.

[40] Lina Ding, Xingwei Wang, Fuliang Li, and Min Huang. A parallel processing method for border gateway protocol update messages. In *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)* (Aug 2015), pp. 2044–2048.

[41] Masip-Bruin, X., Yannuzzi, M., and Siddiqui, M. S. *Inter Domain Routing Security: Securing BGP*, 1 ed. Wiley-IEEE Press, 2014.

[42] McDaniel, P., and Butler, K. Testing large scale bgp security in replayable network environments. In *DETER Community Workshop on Cyber Security Experimentation and Test* (2006).

[43] Mitseva, A., Panchenko, A., and Engel, T. The state of affairs in bgp security: A survey of attacks and defenses. *Computer Communications 124* (2018), 45 – 60.

[44] Mohapatra, P., Scudder, J., Ward, D., Bush, R., and Austein, R. BGP Prefix Origin Validation. RFC 6811, Jan. 2013.

[45] Murphy, S. Bgp security vulnerabilities analysis, 2006.

[46] Oryu, T., and Ishibashi, K. Simulation evaluation of rpki deployment based on cost-benefit analysis. *IEICE Proceedings Series 68*, C1-2 (2021).

[47] Perouli, D., Griffin, T. G., Maennel, O., Fahmy, S., Pelsser, C., Gurney, A., and Phillips, I. Detecting unsafe bgp policies in a flexible world. In *Proceedings of the 2012 20th IEEE International Conference on Network Protocols (ICNP)* (Washington, DC, USA, 2012), ICNP '12, IEEE Computer Society, pp. 1–10.

[48] Premore, B. J., and Nicol, D. M. *An Analysis of Convergence Properties of the Border Gateway Protocol Using Discrete Event Simulation*. PhD thesis, USA, 2003. AAI3097799.

[49] Project, T. J. The julia language, January 2019.

[50] Quoitin, B., and Uhlig, S. Modeling the routing of an autonomous system with c-bgp. *Netwrk. Mag. of Global Internetwkg. 19*, 6 (Nov. 2005), 12–19.

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

[52] Reuter, A., Bush, R., Cunha, I., Katz-Bassett, E., Schmidt, T. C., and Wählisch, M. Towards a rigorous methodology for measuring adoption of rpki route validation and filtering. *ACM SIGCOMM Computer Communication Review 48*, 1 (2018), 19–27.

[53] Roberts, B., and Kroese, D. Estimating the number of s-t paths in a graph. *J. Graph Algorithms Appl. 11* (01 2007), 195–214.

[54] Roughan, M. 2d computational geometry package for julia (programming language), Oct 2014.

[55] Roughan, M., Willinger, W., Maennel, O., Perouli, D., and Bush, R. 10 lessons from 10 years of measuring and modeling the internet's autonomous systems. *IEEE Journal on Selected Areas in Communications 29*, 9 (October 2011), 1810–1821.

[56] Ruan, L., and Susan-Varghese, J. Computing observed autonomous system relationships in the internet. In *Computer Science Technical Reports* (2014).

[57] Sermpezis, P., Kotronis, V., Dainotti, A., and Dimitropoulos, X. A survey among network operators on bgp prefix hijacking. *SIGCOMM Comput. Commun. Rev. 48*, 1 (apr 2018), 64–69.

[58] Sobrinho, J. a. L. Network routing with path vector protocols: Theory and applications. SIGCOMM '03, Association for Computing Machinery, p. 49–60.

[59] Vervier, P.-A., Thonnard, O., and Dacier, M. Mind your blocks: On the stealthiness of malicious bgp hijacks. In *NDSS* (2015).

[60] Wählisch, M., Maennel, O., and Schmidt, T. C. Towards detecting bgp route hijacking using the rpki. *SIGCOMM Comput. Commun. Rev. 42*, 4 (aug 2012), 103–104.

[61] Yeh, W.-C. A new monte carlo method for the network reliability. In *Proceedings of First International Conference on Information Technologies and Applications (ICITA2002)* (2002), Citeseer.

[62] Yoon, S., and Kim, Y. B. A design of network simulation environment using ssfnet. In *2009 First International Conference on Advances in System Simulation* (Sep. 2009), pp. 73–78.

[63] Zhang, Z., Zhang, Y., Hu, Y. C., and Mao, Z. M. Practical defenses against bgp prefix hijacking. In *Proceedings of the 2007 ACM CoNEXT Con-*

ference (New York, NY, USA, 2007), CoNEXT '07, Association for Computing Machinery.