PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

WAN design with demand, hop limit and path diversity constraints

Vincent Ho

Feb. 2008

Orientador de Tesis: Dr. Hector Cancela
Supervisor: Dr. Hector Cancela

WAN design with demand, hop limit and path diversity constraints
Vincent Ho

ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 08-01
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República

Montevideo, Uruguay, Febrero de 2008

Abstract

This thesis presents a challenging telecommunication network design problem, in which the network designer has to assure not only connectivity, but also some degree of performance guarantee and survivability to the solution.
We introduce a topological and capacity design, with node-pair constraints of heterogeneous bounds. These constraints take into account the throughput, the number of node-disjoint paths, and the maximum number of hops. Moreover, the capacity to place on each link is chosen from a pool of available cable types, each with its maximum capacity and fixed cost, in other words the cost function is stepwise with respect to capacity usage.

The solution space of the problem in study is composed by each one of the node-pair demands. An in-depth combinatorial study was done, leading to the development of a new coding for genetic algorithms. The last part of this study focuses on the testing of the proposed genetic algorithms as a solver for the problem in study.

Keywords: topology, capacity, network design, node disjoint paths, hop limit, multi-path routing, genetic algorithms

Resumen

En esta tesis presentamos un problema de diseño de redes de telecomunicación, en el cual se considera junto a la conectividad, también cierta medida de garantía de performance y tolerancia a falla. El objetivo es diseñar topología y capacitad, donde se requiere caminos de nodo disjuntos que no excedan una cantidad máxima de saltos para llevar las demandas entre pares de nodos. Las restricciones son de parámetros heterogéneos, y para cada arista se admite únicamente un conjunto de cables a instalar. Estos últimos tienen capacitad máxima asociado y costo fijo de instalación, en otras palabras, la función de costo respecto al uso de capacitad es una función de escalones.

Las soluciones de este problema son compuestas por caminos nodos disjuntos para cada una de las demandas. Se realizó un estudio combinatorio sobre el espacio de solución y se diseñó una nueva codificación en los algoritmos genéticos. La última parte de este trabajo se centra en la experimentación del algoritmo genético propuesto para la resolución del problema de interés.

Palabras claves: topology, capacity, network design, node disjoint paths, hop limit, multi-path routing, genetic algorithms

Acknowlegements

La realización de este trabajo de maestría contó con el apoyo de una beca financiada por la CAP (Comisión Académica de Postgrado de la Universidad de la República).

Durante el estudio de la maestría, entré en contacto con una cantidad de profesores y colegas. En primer lugar agradezco a Hector Cancela orientador/supervisor de la tesis quien me ha brindado apoyo y orientación en innumerables puntos cruciales a lo largo de la maestría. Es gracias a su paciencia, dedicación e inspiración que logré completar este trabajo.

En el transcurso de estos últimos años, he tomado una cantidad de cursos en áreas interesantes, agradezco a los profesores involucrados por aportar a mi formación: Eduardo Canale, Hector Cancela, José Eduardo Díaz, Enrique Ferreira, Graciela Ferreira, André Fonseca, Antonio López, Sergio Nesmachnow, Franco Robledo, Carlos Testuri y Alfredo Viola.

Sergio Nesmachnow es quien lleva el crédito de haberme introducido al área de computación de alto performance y los algoritmos genéticos. También debo mencionar a los cohabitantes de la oficina con quienes he pasado estos años: Andrés Almansa, Gerardo Ares, Nelson Calero, Pablo Ezzatti, Eduardo Fernández, Tomas Laurenzo, Antonio López, Martín Pedemonte.

El grupo de investigación operativa, donde se encuentra gran parte de mis profesores, colegas y amigos. A través tanto de sus integrantes como de los eventos que ha organizado, he podido entrar en contacto con otros investigadores en el mundo. Un especial agradecimiento a: Hector Cancela, Graciela Ferreira, Antonio Mauttone, Alfredo Olivera, Franco Robledo, Carlos Testuri, María Urquhart y Omar Viera.

Luis Sierra y Álvaro Tasistro quienes me abrieron las puertas al mundo académico, donde comencé como asistente en uno de sus cursos LPI (Lógica de la Programación Imperativa), y de este modo conocí a los otros integrantes de INCO. Así mismo, debo agradecer a los profesores junto a quienes he dictado cursos durante la maestría. Del curso LPI: Alberto Pardo, Luis Sierra y Álvaro Tasistro. Del curso Redes de Computadoras: Juan Diego Ferré, Gabriel Gesto, Carlos Martínez, Federico Rodríguez, Felipe Zipitría. Y del curso Computación I: Guadalupe Artigas, Pablo Ezzatti, Juan González, Daniel Gómez, Martín Pedemonte, Leticia Pérez.

Las bibliotecarias y secretarias de InCo quienes han brindado un soporte imprescindible a la investigación: Lucyla Alonso, Laura Bermudez, Joseline Cortazzo y Mabel Seroubian. El equipo de administradores de InCo quienes nos han dado soporte técnico y material: Julian Adib, Alejandro Blanco, Juan Diego Ferré, Marcelo Rodríguez, Marcos Orfila, Jorge Sotuyo, Felipe Zipitría.

Y por último, en forma algo genérica pero no menos importante agradezco a mis colegas de InCo por las ayudas directas e indirectas que recibí. A mis familiares y amigos quienes me han dado un apoyo moral continuo durante todos estos años.

Contents

Chapter 1

Introduction

Telecommunication is defined as the transmission of signals over distance for the purpose of communication. In the history, some of the most remarkable and popular devices of telecommunication include optical telegraphs like smoke signals in the ancient times, electrical telegraphs since 1700s, telephone since 1870s, radiotelegraphy since 1890s, and Internet nowadays. Internet in particular has its early form dating back to the 1970s as ARPANET, and in only a few decades, it evolved to the "nervous system of commerce".

Another of the most important invention of the last century is the electronic computer. From its first prototypes to the modern personal computer and supercomputers, it took once again, only a few decades. The fact is that nowadays computers and networks of computers have become important and even essential in many different professions. There exist a huge and ever growing number of examples in which traditional ways of communication like telephone, letters, multimedia broadcast are being replaced by modern alternatives over a computer network. And many other new possibilities have opened with it, like chat, teleconference, electronic auction, e-business, massive multiplayer game, etc.

The extensive importance of computer networks in our society and its widespread growth brings a number of problems in planning and administration, problems that optimisation can address. The objective of this thesis is to address a WAN design problem including some of these real-life characteristics (which augment the complexity and the difficulties of solving the problem). The approach consists in designing a genetic algorithms (G.A.) based solution method and to study some of its characteristics.

Among the existing global search techniques, we found evolutionary computation interesting, and in particular we center the study on genetic algorithms, which have proved to yield good results in many different $\mathcal{N}P$-hard combinatorial problems. The drawback of the usage of heuristics or metaheuristics is that by definition they don't guarantee the optimality of the final result. Although this sounds terrible, in reality, there are many optimisation problems for which no exact solution method is known other than the brute force approaches like exhaustive exploration, those approaches are only practical for problems with small solution space. Furthermore, there are many other problems for which, the search of a feasible solution is already a difficult sub-problem, without even considering optimisation over the entire solution space. Unfortunately, many problems with important real life application fall in those categories, and for those problems in which we do not know any efficient solution method, we can only sacrifice the optimality and trade it for solutions of good quality, as measured by empirical results or guaranteed with high probability for some theoretical results.

The rest of this thesis begins with Chapter 2 where we discuss some of the most usual goals in WAN designing together with some well known problems and solutions. Then later on, in the same chapter we define an ambitious problem that includes service quality and reliability. Afterwards in Chapter 3 we introduce genetic algorithms as the chosen solver, together with GGEngine a genetic algorithms library that we developed and which allows us to solve problems with a minimum effort in implementation. In Chapter 4 the experiments and results are presented, and we conclude in Chapter 5 with a summary and conclusions from the obtained results.

Among the appendices we have Appendix A written as a standalone article, where we present in detail the enumeration of node-disjoint paths with hop-limit, which is one of the principal components in the coding for our genetic algorithms. Appendix B is a standalone technical report with an overview and user manual of GGEngine v2.0 -the library we developed and used in this thesis-. Then in Appendix C we present the format of the problem's input file and in Appendix D details of the basic cases in the test suite.

Chapter 2

Wide Area Network design problems

2.1 Introduction

Data communication or computer networks are naturally modelled as graphs, where routers and hosts are represented by nodes, while cables interconnecting them are represented by edges (or arcs for unidirectional communications).

Graph theory has its origins date back to 1736 when Leonhard Euler modelled the Königsberg bridge problem with the Eulerian circuit. More than two hundred years later, in 1969, the first computer network ARPANET gave birth to Internet, which has been growing rapidly. This interesting area of application has been a significant boost in the growth of graph theory as an important area in mathematics.

The size of a communication network brings problems to its design and management and Internet has been growing to a planet wide size. A sign of its growth is the address shortage which appeared in only three decades after the birth of Internet. To delay the problem, Internet Assigned Number Authority (IANA) was forced to apply the current policies: Classless Inter-Domain Routing (CIDR), and reclamation of prior allocated addresses[Tanenbaum 03]. According to a recent report [Hain 05], the amount of free address in IPv4 can last a few years more (two to five depending on consumption), and meanwhile IPv6 is being designed and deployed as its successor with a wider address space among other improvements. Another clear sign is mentioned among others in a report of the AT&T Labs [Coffman 01], which states that the monthly traffic on the backbone in U.S. grew from 1 terabytes in 1990, to 20,000-30,000 terabytes in 2000.

Many of the problems in network design and management are transformed into classical combinatorial optimisation problems in graphs. Nowadays only a few of them can be solved in polynomial time (belonging to class P). But many others have not been solved in such an efficient way, for example, those that belong to $\mathcal{NP}$-complete, $\mathcal{NP}$-hard or beyond are considered intractable for medium or even small problem instances. The network design problem studied in this thesis belongs to the $\mathcal{NP}$-hard class, for this kind of problems instead of exact algorithms, many researchers propose heuristics which trade in the certainty of obtaining the optimal solution for a better response time, and that's also our approach. Before talking about our specific problem, we introduce some related and simplified problem in the following sections, and show briefly how they can be treated.

2.1.1 Definitions, Notations and Conventions

We adopt a classical graph notation similar to the one employed in [Diestel 00, Weisstein 05].

By $G(V, E)$ we denote an undirected graph which is a pair of sets V and E, such that V represents a non empty set of vertices , and E represents the set of edges that are drawn between pairs of vertices. When talking about digraph (directed graph) $G(V, A)$ we use the term arcs to refer to ordered pair of nodes, and A is the set of arcs. In the context of communication network designing, vertices are also called nodes or sites, while edges and arcs are being called full duplex links and unidirectional

$G(V, E)$
set V
set E
set A

links respectively. Unless stated otherwise, we only work with graph without self loops and multiple arcs, because in the context of communication network design, multiple arcs are usually represented by a single arc with the aggregated properties; and loops (arc from a node to itself) are not a valid design. The cardinal number of a set S is referred to by $|S|$. | |

A path is a sequence of nodes, as we use path to model the route followed by a stream of packets in a network, so unless stated otherwise, path doesn't contain cycles. A source-target (s-t) path is one that begins with a given source node s and ends with a given target node t. Paths that don't share any intermediate nodes are called node-disjoint paths. Paths that don't share any edges are called edge-disjoint paths (in an undirected graph). A set of node-disjoint $s-t$ paths with cardinality K is referred by s-t K-path, or K-path in short . s-t path | s-t K-path | K-path

A non-empty graph is called connected if any two of its nodes are linked by a path. A graph $G(V, E)$ is called k-connected for $k \in N$, if $|V| > k$ and $G(V, E)/X$ is connected for every set $X \in V$ with $|X| < k$, in other words, there are at least k node-disjoint paths between each pair of nodes. k-connected

2.1.2 About Complexity

The following is a brief summary of some important classes of problems in the theory of complexity.

A decision problem Π is one whose solution is either positive (yes) or negative (no), formally it consists of a set of problem instances I_Π, and a subset of positive instances $Y_\Pi \subseteq I_\Pi$. The $\mathcal{N}P$ class stands for those decision problems that can be solved in polynomial time by a non-deterministic Turing machine, additionally any positive instance can be verified in polynomial time in a deterministic Turing machine. The P class stands for those decision problems that can be solved in polynomial time by a deterministic machine. And it is well known that $P \subseteq \mathcal{N}P$. $\mathcal{N}P$ | P

The $\mathcal{N}P$-completeness was designed to be applied only to decision problems [Garey 79], and when the problem of interest is not a decision but an optimisation problem, there are two common guidelines. The first one is to recast it into a decision problem, and study the related decision problem. There are many different ways of recasting, and in those cases where the optimisation problem can be used to solve the decision problem, a relation in their complexity can be established. A second alternative is to extend the definition of $\mathcal{N}P$ over optimisation problems like in [Crescenzi 99, Crescenzi 00]. An optimisation problem consists of a set of problem instances I_Π, a set of feasible solutions sol_Π, an objective function m_Π and a goal which can be either max or min. Then the $\mathcal{N}PO$ class is all those optimisation problems such that: $\mathcal{N}PO$

- Problem instances belonging to the set I_Π are recognisable in polynomial time.
- For any instance $x \in I_\Pi$, $sol_\Pi(x)$ denotes the set of feasible solutions of x, those solutions are short[1] and for every short solution[2] y, it is decidable in polynomial time whether $y \in sol_\Pi(x)$.
- Given an instance $x \in I_\Pi$ and a feasible solution $y \in sol_\Pi(x)$, $m_\Pi(x, y)$ denotes the value of that solution, and it is computable in polynomial time.

Informally a problem is in $\mathcal{N}PO$ if its solutions can be checked and evaluated in polynomial time. To solve an optimisation problem is to find (depending on the goal) a max or min feasible solution according to the measure function. An optimisation problem belongs to the PO class when it is solvable in polynomial time. PO

There are also some interesting classes of problems defined for counting problems. For instance, $\#P$ is the class of functions f such that there exists a non-deterministic polynomial time Turing machine M with the property that $f(x)$ is the number of accepting computation paths of M on input x (extracted from [Allender 99]). In other words, those are functions that their total number of accepting paths can be counted with a non-deterministic Turing machine in polynomial time. Clearly, any function in $\#P$ has its corresponding decision problem in $\mathcal{N}P$.

Let C be a complexity class, a problem Π belongs to the C**-hard** class if any problem in C is reducible to it, in other words, Π is at least as hard as any problem in C. If a problem is in both C-hard and C classes, then it is said to be in the C**-complete** class (the hardest among C, but not beyond). Refer to [Garey 79] for an introduction to the theory of complexity. -hard | -complete

[1] A polynomial p exists such that for any $y \in sol_\Pi(x)$, $|y| \leq p(|x|)$

[2] A "short solution" is any y, such that $|y| \leq p(|x|)$.

2.1.3 Network Design Problems

In the remaining of this chapter, we present a few computer and communication network design or redesign problems, each of them are classical example of specific design goals. Those are questions like "how to do something in the best way", or "how to do it with the lowest cost", given certain conditions or constraints. In Section 2.2 we present some problems in the category of Topological Design; in Section 2.3 we present some others about Capacity Planning; so finally in Section 2.4 the particular problem studied in this thesis is presented.

2.2 Topological Design

2.2.1 Hierarchy: Different kinds of data communication networks

A topology design specifies which hosts are connected to which other hosts. So the question here is: "between which hosts shall we install a cable to fulfil the given requirement?" While a capacity design specifies the type or kind of cables to use between each pair of nodes in a topology. There are many kinds of networks, and associated requirements, one possible categorisation is in WAN/-MAN/LAN.

Local Area Network[3] (LAN) usually is located in a single building, because they are based on hardware with great capacity but supports only a limited distance, see Table 5. The most popular technology used in LAN is Ethernet (10/100BaseT), FDDI sometimes is used as a LAN backbone or to interconnect high traffic demand hosts. The goal in a LAN is the interconnection between a number of hosts (varies from two to thousands), which may also include an entrance point to Internet, some typical topologies are tree, ring, star, or hybrids of them; path diversity is not as critical as the MAN or WAN counterparts.

Metropolitan Area Network[4] (MAN) is a network that covers an area as large as a city, like an Internet Service Provider (ISP), or a private enterprise with a citywide network. The goal in this kind of network is to interconnect a number of LAN represented by their entrance points, and a few out-going routers which communicates to other networks like neighbour ISP, or even the Internet. The installation of cables here is more expensive, and path diversity is usually a requirement, because the number of involved host is much larger than the LAN counterpart. A single node in this case may represent thousands of LAN hosts. Typical topologies in this case are ring or mesh as the backbone (carrying aggregated traffic), and trees connecting it to groups of access points.

Wide Area Network[5] (WAN) is a country or continent wide network, as an example: Network Service Provider (NSP), banking networks, airline reservation network, etc. The goal is to interconnect different networks across a countrywide area. In this case, the installation cost and bandwidth of cables are extremely high (like burying long distance underground or submarine cables, satellite communications, etc.). Topologies for WAN require higher path diversity, because its nodes are transit nodes or collapsed MAN (representing a big number of customers). In this case, reliability is not only desirable, but necessary.

Sometimes when designing MAN or LAN topology, the whole internet backbone is collapsed into a single special node, so the designer can focus on the access network. Other times when designing WAN topology, different MAN are collapsed into different nodes, so the designer can focus on the WAN level requirement. But why the three levels are treated separately? It's because nodes of a certain level can only connect to nodes of other levels in a limited way, and network design problems can be simplified by dividing the network into different levels.

[3] Also called local access network, sharing the same acronym.

[4] Also called inter-office/switching centre networks, interconnecting switching centres/local exchanges/central offices, see [Balakrishnan 91].

[5] Also called long-distance/toll/inter-city network that connects city pairs through gateway/point-of-presence nodes, see [Balakrishnan 91].

Type of Link	bandwidth (bit per second)	Maximum length (meter)
10Base2 (Thin Coaxial)	10 000 000	< 18
10Base5 (Thick Coaxial)	10 000 000	< 500
10BaseT	10 000 000	< 100
100BaseT	100 000 000	< 100
FDDI	100 000 000	< 2000
Optic Fibre	1 000 000 000	< 1000

Table 2.1: **Some popular LAN cabling**

2.2.2 To achieve Connectivity

An interconnected communication network can be modelled as a connected graph. The problem of connected network design has many applications in telecommunication and transportation network. The following two problems have their applications in real life LAN designs.

MST: Minimum Spanning-Tree problem.

Given:

- an undirected graph, denoted as $G(V, E)$
- a non-negative real function representing the cost of each edge denoted as $C : E \rightarrow R^+$

Decision variables:

- edges from E to integrate a sub-graph

Objective:

- find the lowest cost sub-graph in G

Constraint:

- each node in V is connected

MST

If the cost function is positive, then the result is always a tree. There exist a number of polynomial time algorithms for MST like those of Prim, Kruskal, Boruôvka, or Chazelle. To the best of our knowledge, the best algorithm for this problem nowadays is the one proposed by Bernard Chazelle[Chazelle 00], it's a deterministic polynomial time algorithm with $O(|E|\alpha(|E|, |V|)) \approx O(|E|)$ where α is the classical inverse[6] of Ackermann's function, which grows with a nearly constant order. Moreover it can also solve MST with arbitrary edge costs [Chazelle 00]. An application of MST is to determine the lowest cost cabling for an access network, in which all hosts are already fixed: workstations in an office, clients of an ISP in a city, etc.

[6]It's not really the strict inverse of Ackermann's function, but a function that grows so slowly as Ackermann's function grows quickly.

SPG: Steiner tree Problem in Graph

Given:

- an undirected graph $G(V, E)$
- a non-negative real function $C(e)$ representing the cost of each edge $C : E \rightarrow R^+$
- a subset of nodes called terminal nodes, denoted as V_T such that $V_T \subset V$

Decision variables:

- nodes from V and edges from E to integrate a sub-graph

Objective:

- find the lowest cost sub-graph in G

Constraint:

- each node in V_T is connected.

SPG

For SPG, no polynomial algorithm is known nowadays. The fact that SPG is $\mathcal{NP}O$-complete ([Garey 79, Karp 72] implies that the decision version is $\mathcal{NP}$-complete[7]) makes it impossible to be solved efficiently in a modern computer, unless the classes P is equal to $\mathcal{NP}$ (a still open one million dollar question [Clay Math. Inst. 05]).

Comparing MST and SPG, we can see that the only difference is in the set of nodes to be connected, keeping some of them as Steiner nodes (optional nodes) can transform an easy problem into a difficult one. The SPG received much attention since its formulation, and there are polynomial time exact algorithm for some special variants like: Hwang's algorithm for full Steiner trees in an Euclidean plane (a linear order time algorithm) [Hwang 86]; There are also some approximation algorithm like the one from [Robins 00] which achieved a performance ratio of $1 + \frac{ln(3)}{2} \approx 1.55$. But for large problems heuristics are used, examples are GRASP [Martins 98], Genetic Algorithms [Esbensen 95], Tabu Search [Ribeiro 00], etc.

But what happened to the solution space at the moment when we added those Steiner nodes? An optimal solution is just a minimum spanning tree in a set of optimal nodes S, as the construction of a MST is not hard, the difficulty of this problem is in the selection of an optimal set of Steiner nodes. Note that being a solution, it must contain all the terminal nodes V_T, let's analyse the required comparisons when an exhaustive scheme is used:

Let $n = |V| - |V_T|$, if the exact number of Steiner nodes in the optimal solution is i and we know it, than we only have to check $\binom{n}{i}$ possible combinations of Steiner nodes. But as i is unknown, we have to compare $S_n = \sum_{i=0}^{i=n} \binom{n}{i}$ solutions to get an optimal one.

Adding one extra Steiner node to a given SPG changes its solution space[8] from S_n to $S_{n+1} = 2S_n = 2^{n+1}$. The day when someone finds out how to identify non-optimal Steiner nodes efficiently, the SPG can be solved with the same efficiency by discarding nodes one by one, as each removal reduces the solution space to a half. On the other hand, notice that in SPG the solution space grows exponentially with the parameter n, this kind of huge solution space is very common in combinatorial optimisation problems.

Both problems are particular cases of our design problem, this is discussed in detail in Section 2.4.5.

[7] For SPG, both the optimisation and search problem can be reduced to the decision problem with a polynomial algorithm.

[8] In Section A.3.1, we present a structure called binomial-tree, with one that has n nodes on the first level, S_n is just the total number of nodes. Comparing it with a $n + 1$ one, the stated equation can be obtained easily.

2.2.3 To achieve Reliability/Survivability

There are many different hardware and software components involved in a computer network, and many of them are susceptible to failures. A cable between two computers can be affected by an electromagnetic interference; any program may contain bugs, etc. It's impossible to construct a flawless network with our current technology, but we still can coexist and tolerate a certain level of errors. Like the way how each entity in the different layers of the OSI model has to implement redundancy and retransmission, a network designer must add redundancy to his/her design so an eventual failure in one or multiple components does not bring unacceptable side effects. This requirement is called survivability[9], the ability to maintain a certain degree[10] of network service in the event of a component failure.

In the scenery of topological design, redundancy is implemented as additional links added to offer route diversity. But each such link increases the total cost, so we can not add them indiscriminately. Notice that graphs with path diversity factor equals to two or greater have ring or mesh topology, while the minimally connected graphs have tree topology. Adding links to a tree may not lead to the global optima in the former problem, but is a valid heuristic.

Problems in this category may contain a node-disjoint or a link disjoint path requirement. The former is a stronger condition, because its solutions are always solutions of its counterpart, but not inversely. As a result, optimal solutions in node-disjoint formulations have higher cost than the link disjoint counterpart. If there are K link disjoint routes between two nodes, then such a design can survive $(K-1)$ link failures; if there are K node-disjoint routes between two nodes (notation: K-path), then such design can survive not only $(K-1)$ link failures, but also $(K-1)$ node failures (without counting the source and target nodes).

There are many classical optimisation problems in this theme, refer to [Grötschel 95] for a complete survey of researches previous to 1995. Another particular case of our network design problem is the following:

GSP-NC: Generalised Steiner Problem with Node Connectivity constraints

Given:

- an undirected graph $G(V, E)$
- a non-negative real function $C(e)$ representing the cost of each edge $C : E \rightarrow R^+$
- a subset of nodes called terminal nodes, denoted as V_T such that $V_T \subset V$
- for each pair of nodes in V_T, a natural number specifying the number of node-disjoint paths required between them

GSP-NC

Decision variables:

- nodes from V and edges from E to integrate a sub-graph

Objective:

- find the lowest cost sub-graph in G

Constraint:

- between each pair of nodes in V_T the required number of node-disjoint paths is present

Note that the resulting sub-graph need not be connected. GSP-NC is a more general version

[9] Something is reliable if it works most of the time; something is survivable if it works even when some component is broken.

[10] If a node is down, then traffic directed to it may unavoidably interrupted, but disconnection in its neighbours can be avoided with redundancy.

of SPG, as the latter is an instance in which all node-pair connectivity requirement is set to one. This problem is known to be $\mathcal{NP}O$-hard, as it generalises SPG, which is a $\mathcal{NP}O$-complete problem. Belonging to the $\mathcal{NP}O$-complete class means that at least one instance of it, is very hard, but we can see that there also exist some easy instances. If in V_T there are only two terminal nodes, it is the problem of finding the set of shortest node-disjoint paths in a graph, which is solvable in polynomial time with algorithms by [Bhandari 99] (an explanation of those algorithms can also be found at Appendix C of [Pióro 04]).

Another similar problem is the GSP-EC, in which the node-disjoint requirement is replaced by an edge-disjoint requirement. Both GSP problems were introduced by Krarup, and are known as $\mathcal{NP}O$-hard [Robledo 00, Westbrook 95]. In [Agrawal 95] an approximation algorithm for solving the GSP-EC problem within $2\lceil log_2(r+1)\rceil$ of the optimal cost is presented, where r is the biggest connectivity requirement. In [Nesmachnow 04] a parallel genetic algorithm is proposed for GSP-EC together with a survey of the problem. One of the application of this problem is in the design of MAN/WAN topology with fixed access nodes and optional transit nodes.

2.2.4 Adding Performance Constraints into a Topological Design

Another aim of telecommunication network designing is to achieve good quality of service, which means not only high throughput, but also low delay. The following is a brief summary, then we discuss it further in Section 2.3.5, as this goal is strongly related to capacity planning. The theme of this subsection is network topological design problem with performance requirements.

The more capacity assigned to the links (and nodes) of a particular route, the lower will be the response time we get for packets travelling in it. Adding capacity improves the performance, but it also increases the cost of the design, an alternative is to shorten the length of routes. Long routes (paths) in a topology not only worsen the performance, but also increase the probability of failure as they involve more components. In a topological design, hop limit in paths or diameter length in graphs help to reduce delay. The traditional approach for network design divides the problem into stages, where topological design is only one of them. When the performance is an objective, some performance constraint must be added into the topological design stage. If the routes between nodes are unknown, the usual option is diameter limit. But if those routes are also a decision variable, then hop-limit is a more suitable choice, because it guarantees the bound in path length while graph diameter limit doesn't[11].

[11] For shortest path routing, graph diameter does bound the path length, but many other routing algorithms like in load based routing, longer paths may exist.

WMP: Weighted Multi-demand Problem

Given:

- an undirected graph $G(V, E)$
- a non-negative function $C(e)$ representing the cost of each edge $e \in E$, $C : E \to R^+$
- a set of demands D, which contains (s_i, t_i, K_i, H_i) as elements. Where $s_i, t_i \in V$ are the i-th source and target nodes. $K_i, H_i \leq |V|$ are two natural numbers representing the node-disjoint diversification factor and the hop limit for this pair of nodes.

Decision variables: WMP

- nodes from V and edges from E to integrate a sub-graph

Objective:

- find the lowest cost sub-graph in G

Constraint:

- between each pair of $s - t$ nodes in D the required number of node-disjoint paths is present and the hop-limit constraints are respected.

The WMP is shown to be in $\mathcal{NP}O$-complete in [Bley 97] (originally in [Itai 82]), even in the simplified version with single $s - t$ pair (WSP) the problem remains in $\mathcal{NP}O$-complete. Together WSP with this problem arises the question: "is there any feasible solution for a given problem instance?", such a feasibility problem belongs to the $\mathcal{NP}$-complete class for both WSP and WMP. It means that it is hard even to know if the set of feasible solutions is empty or not for some instance of WMP. However, if all hop-limits are less than or equal to four, the feasibility problem can be solved in polynomial time, by dividing it into different single $s - t$ feasibility problems.

The WSP can be solved in polynomial time for the following special cases:

- $K = 1$. Modified Dijkstra with hop-limit [Bley 97, Pióro 04].
- G is a complete graph, $C(e)$ verifies the triangle inequality. It implies $H \leq 2$ which is trivial.
- $C(e)$ verifies the triangle inequality and $H \leq 4$ [Bley 97, Itai 82].
- $H = |V| - 1$. The case without hop-limit can be solved with an algorithm for "shortest set of disjoint paths" explained in [Pióro 04], in a few words we can describe it as an iterative algorithm that in each step transforms the graph and applies a modified variant of Dijkstra's algorithm which admits negative weights.

In [Bley 97] an exact algorithm and two heuristics are proposed for WMP. The exact algorithm begins with all the edges of the given graph, and then drops links in a branch and bound fashion. The first heuristic begins with an initial feasible solution (given by an augmentation heuristic), and then drops redundant links. And the last constructive heuristic is composed by three phases, solving the connectivity, feasibility, and then finally dropping redundant links.

In [Kumar 93][12] a Genetic Algorithm based network design problem with diameter constraint appeared. Later in [Huang 97] a 3-connected graph design with diameter limit, for complete graph was solved also with a Genetic Algorithms based approach.

In [Cahn 98] a survey of more network design problems and associated solution techniques are presented, including problems with limit in the maximum or average number of hop combined with routing algorithms. The heuristic techniques they proposed for these two problems are culling (not suitable when the probability of generating a feasible solution is low) and augmentation.

[12]Some details are missing, like the correction mechanism in case when a non-feasible individual appears.

2.3 Capacity Planning

2.3.1 About Routing

There is a close relationship between capacity planning and the routing protocol in use, a network designed to work optimally with one routing algorithm may not work as well for others. The difference in using one routing algorithm or another is essentially in the selection of path to satisfy a particular pair wise demand, which plays a very important role in the capacity planning stage. For this reason network designers must take into account how the underlying routing protocol works in order to plan the capacity consistently.

In general, a routing algorithm takes a topology as input, and decides which route to use for interconnecting each pair of sites. In operation, a protocol is designed so that each router knows to which other it should forward a packet, for it to arrive to the destination. Most routing protocols are adaptive, upon a change in the topology they update the routing table in each node to work around it, something that a static protocol does not provide. Some of them even consider the workload in the routing, to avoid overwhelming an already congested route. It is not necessary to use the same routing protocol in the whole network, for example, Internet can be viewed as a interconnection of many Autonomous Systems and those AS are managed by different entities using their chosen Interior Gateway Protocol (like OSPF, RIP, IS-IS). Border Gateway Protocol version 4 is used to establishes the interconnection between AS nowadays [Rekhter 06, Meyer 06].

AS
IGP
BGP

The behaviour of some routing algorithm can be predicted easily, like the case of shortest path algorithm. For more complicated routing, a network simulator like NS2 can be used to evaluate and predict the behaviour of a draft network, even during the capacity design stage.

2.3.2 Multi-Path Routing in the future

For a network to survive component failures, it is necessary to add backup components to increase its reliability, but backup components also increase the total cost of a network, whereas single path routing networks can only use one of the multiple paths at a time. Moreover, the performance of one fully loaded path is much worse than two half-loaded paths because of the effect of congestion. Therefore, in network designs with multiple paths, it is more effective to use them all at once with the help of multi-path routing. Nowadays Internet uses only single path routing protocols, because key concepts of multi-path routing are still in development and still not standardised. An example is that TCP consider a connection is congested when packets arrive out of order, so sending packets of the same TCP connection through different paths may lead to a false alarm. Most protocols in Internet were designed for single path routing, therefore the implementation of the multi-path alternative has to work around many difficulties. A work around for the previous example is to send packets of a same TCP connection through a same path, and add a mechanism to assign paths to TCP connections. Even when multi-path routing is not yet popular, it can be implemented in an AS nowadays, in [Lee 04] is a survey of some alternatives applicable to MPLS through traffic engineering(for information about MPLS-TE check [Awduche 99]). Because multi-path routing is a more efficient and consistent way of using survivable network designs, it would not be a surprise if it replaces single path routing in the future.

MPLS-TE

2.3.3 Throughput as an input of the problem

Throughput is the amount of data transferred in the network from a specific node to another, divided by the time taken to transfer it, usually expressed in bits or Bytes per second (bps or Bps). For network designing, it is an aggregation of data transferable form one node to another, divided by time unit.

throughput

When planning capacity for a network, the required pair wise throughput is an important input. However, how can a network engineer know the minimum throughput required for a pair of nodes? If there is already a physical network, this can be done by adding traffic monitors in routers to give an insight into the current resource usage, and then, make projections to predict the growth. Very often, the resource usage of a network is not constant, it usually present a certain pattern in

a day-of-week or time-of-day basis with peaks and troughs. Depending on the deviation, sometimes the average is also a good representative. However, to maintain a good throughput even during the peak utilisation, the peak value should be used as the requirement. But when designing a network in which the peak hour is different for different pairs (for example, when the physical distance covers different time zone or when users of very different behaviours are present), consider the peak usage of all of them simultaneously may lead to an underused and expensive network, therefore further optimisation is possible there. For capacity planning problems, the throughput is considered an input data given by some expert, so the problem's focus is to define what capacities to place in which links so the requirements are satisfied.

2.3.4 Capacity Planning Problem

Capacity is the maximum possible data transfer rate of a communication channel under ideal conditions. It can be associated with a link or a node, but in all the problems presented in this thesis it is associated with links exclusively[13]. capacity

Remember the minimum spanning tree problem (MST), which is solvable in polynomial time. There is another very similar problem that includes capacity design, namely capacitated minimum spanning tree problem (CMST).

CMST: Capacitated Minimum Spanning-tree Problem.

Given:

- an undirected graph, denoted as $G(V, E)$
- a central server node $s \in V$
- a cost or length associated to each edge $L : E \rightarrow R^+$
- a maximum capacity associated to each edge $C : E \rightarrow Z^+$
- a demand $d_i \geq 0$ associated to each node in $V/\{s\}$, to be sent to the server node

Decision variables: CMST

- edges from E to integrate a tree

Objective:

- find a tree in G with the shortest total length

Constraint:

- each node in V is connected
- the capacity in each edge can carry the aggregated traffic in all paths passing through it

CMST is shown to be $\mathcal{NP}O$-complete in [Papadimitriou 78][14], and integer demand in nodes can be transformed to an equivalent problem with unit demand (and zero cost between some links). The associated decision problem remains $\mathcal{NP}$-complete even when the capacity is uniform in general, see ND5 from Section A.2.1 in [Garey 79]. A classic heuristic for this problem is Esau-Williams' algorithm [Cahn 98] from 1966, it is a $O(|V|^2 log(|V|))$ algorithm. Another relevant work is [Jothi 04] whose authors present a summary of previous approaches to this problem, together with a modification to Esau-Williams' algorithm with a claim of 1.6% average improvement in solution quality

[13] Because for a directed problem, node related capacity can be dealt with by splitting each node into incoming and outgoing nodes, and place a capacitated arc (directed link) between them. Moreover undirected problem can be transformed into directed problem with additional constraints to re-establish the undirected nature.

[14] More precisely, the decision problem is shown to be $\mathcal{NP}$-complete, and as feasibility can be verified in polynomial time, the optimization is $\mathcal{NP}O$-complete.

(and up to 17% in some experiments) without increasing the worst case run-time.

Let's consider a more general version of CMST, in which the capacity can be chosen among a range of possibilities, and the cost varies along with the capacity (say cost is positive and linear with capacity). We can call this problem CPMST, which stands for Capacity Planning in Minimum Spanning Tree. It is more interesting because it features the dimensioning of capacity in edges. CPMST

But what happens when we added capacity planning into the problem? Well, the cost in each edge is no longer fixed, and it now depends on the structure of the tree. This fact invalidates the usage of all the mentioned algorithms for MST to solve this problem. Notice that, if the topology (a tree) is fixed, it only requires $O(|V|)$ operations[15] to determine the minimum capacity. We can say that the difficulty of this capacity dimensioning problem is not so related to the capacity, but to the variable cost.

For network design in general, [Gerla 77] presents a survey of capacity design techniques of the 70's, those are traditional approaches. The scheme is to divide the network design problem into different stages, namely topological design, capacity design, and routing. Moreover, the capacity and routing problem can be merged into the so called capacity and flow assignment problem, in which both the capacity and the routes to use are design variables. The capacity planning stage takes a fixed topology as input and includes some constraints, for example performance constraints. To model performance, the abovementioned work managed to apply queuing theory by introducing a number of assumptions to simplify the problem, like specific packet arrival distribution, the independence assumption, etc. Additional insight can be found in queuing network literatures or in the article [Kleinrock 02].

2.3.5 Performance in Capacity Planning

We will concentrate on one important measure of the performance of a computer network, namely the delay for information to travel from one node to another. Node to node delay in a network is affected by:

- propagation delay (property of cable or type of link to use);
- number of hops (depends on the route to be used);
- utilization or congestion in components (related strongly to the capacity of the components involved);
- retransmissions due to packet losses in components (reliability of cable or type of link).

We can include performance guarantees into a network design by choosing low propagation and high reliability links, short paths in routing, or by adjusting the capacity of different components to reduce congestion. As a result, we can say that performance is closely related to the selection of types of links to use, namely capacity planning.

Before optimising the design with performance constraints, we have to model performance somehow. One alternative is the usage of queuing theory [Gerla 77, Kleinrock 02, Robertazzi 00] which requires some assumptions to simplify the analysis and make the problem treatable. Another alternative is to incorporate a network simulator to simulate traffic and evaluate performance, moreover, simulators can overcome some of those simplifications for example by using empirical data. During the initial steps of this study, we evaluated the usage of the network simulator NS2 as an oracle or black box for performance evaluation, because it was one of the possible directions of the thesis. But later on, we developed a coding of K-paths, in which the hop-limit constraint can be included. As a result, we decided to use hop-limit as a performance guarantee in our problem.

There are a number of ways to limit the number of hops in the routes to use, for example, by limiting the diameter of the graph (a topology related constraint, suitable for shortest path routing),

[15] First mark and list all the leaves with $|V|$ operations, then for each leaf set the capacity of the edge leading to it (according to the demand) and increase the demand of the father node, also mark and add the father node to the end of the leaves list (if it isn't already marked). Each node will enter and be processed in the list only once, totalling $O(|V|)$ operations.

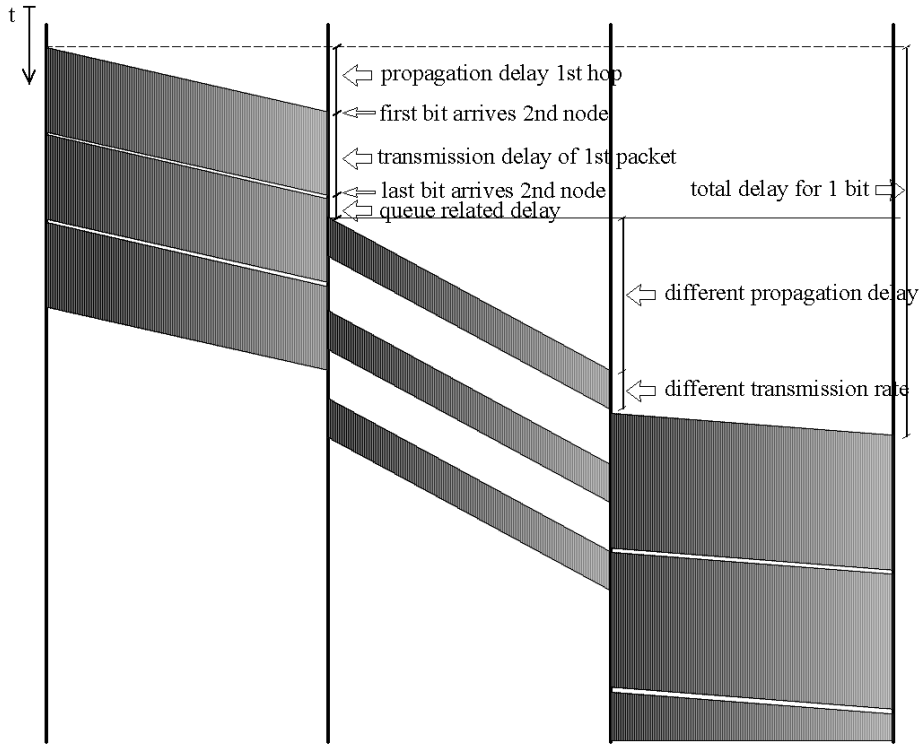


Figure 2.1: Picture showing how the different types of delay contribute to the total delay in a packet switching network.

by limiting the average hop length of the paths (no guarantee for the worst case), or by using explicit limits for each path. Both of the two last cases depend on the routing algorithm in use, and path-wise limit is a more general constraint.

2.4 Our Problem: TC-DHK

2.4.1 Introduction

Previously we introduced some important goals and problems in network communication design. Our problem of interest arises in a communication network design with pair wise connectivity, throughput, reliability and performance requirements. To settle down these requirements, we consider as a reliability requirement the existence of node-pair node-disjoint paths, which is a way to survive link and node failures. To plan the capacity for a network, we have to understand the available underlying routing protocol, or chose the most suitable one to use. Adaptive routing protocols are popular because in the event of a component failure, they re-establish the routes trying to maintain the connectivity. Moreover, if the designer knows exactly which paths are optimal between each pair of sites, then a more suitable alternative is a fixed multi-path routing protocol, as it uses all the optimal routes, operating in a more efficient way. Therefore, we chose multi-path routing protocol. About performance, a simple way to exclude routes with high response time is by using hop-limit. Furthermore, this restriction also bounds the number of components in a path, together with its probability of failure. We consider discrete capacity and cost for possible cables to use in a link, while this feature is more difficult to deal with than continuous capacity and cost, it is a more realistic assumption.

TC-DHK: Topology and Capacity design with throughput demand, hop-limit and path diversity constraints

Given:

- a directed graph, denoted as $G(V, A)$
- a set of demands D, which contains $(s_i, t_i, K_i, H_i, f_i)$. Where $s_i, t_i \in V$ are the i-th source and target nodes. $K_i, H_i < |V|$ are two natural numbers representing the node-disjoint diversification factor and the hop limit for this pair of nodes. And a positive real number f_i as the required throughput for each of the K_i paths.
- an array of available types of cables T, which contains for each type t the pair (m_t, ξ_t), where m_t is the capacity of the t-th type and ξ_t is its cost.
- a binary matrix $U_{|A|\times|T|}$ with ones in the position (a, t) if the t-th type is available for the a-th arc[a]; and zeros otherwise.

Decision variables:

TC-DHK

- nodes from V and arcs from A to integrate a sub-graph G_{sol}
- a type of cable to use for each selected arc in G_{sol}
- paths to use for each node-pair in D

Objective:

- find the lowest cost sub-graph in G

Constraints:

- each pair of node in D is connected with the required number of node-disjoint paths with no more than the specified number of hops.
- the type of cable used in each arc in G_{sol} has enough capacity to carry the aggregated flow of all paths using it, and is indicated as a valid option by U.

[a]Formally, A is a set and the a-th arc in A makes no sense. We consider an implicit order among elements of A and take the a-th arc according to that ordering.

2.4.2 Problem Formulation 1

With a mathematical formulation, we can specify the problem in an unambiguous way. Using the same notation as in [Pióro 04], let us consider the following Multi-commodity formulation of our network design problem: *Given a set of possible paths for each demand, referred by P_d:*

indices

v	$= 1, 2, \ldots, \|V\|$	nodes
a	$= 1, 2, \ldots, \|A\|$	arcs (directed links)
t	$= 1, 2, \ldots, \|T\|$	types of cables to be placed in arcs
d	$= 1, 2, \ldots, \|D\|$	demands
p_d	$= 1, 2, \ldots, \|P_d\|$	index of candidate paths to fulfil demand d

constants

δ_{ap_d}	$= 1$, if arc a belongs to the path p_d fulfilling demand d; 0, otherwise
α_{vp_d}	$= 1$, if node v belongs to the path p_d fulfilling demand d; 0, otherwise
s_d	source node for demand d
t_d	target node for demand d

f_d volume of demand d
k_d diversity factor (or local node connectivity between s_d and t_d) for demand d
ξ_{at} cost of the t-th type of cable on arc a
m_{at} capacity of the t-th type of cable on arc a
H_d maximum number of hops for demand d

variables
x_{p_d} flow fulfilling demand d allocated on the path p_d (continuous non-negative)
u_{p_d} binary variable indicating whether the path p_d is chosen for demand d
u_{at} binary variable indicating whether the cable of type t is installed on arc a

objective
minimize $F = \sum_a \sum_t \xi_{at} u_{at}$

constraints

$\sum_{p_d} x_{p_d} = k_d f_d$	d=1, 2, ..., $\|D\|$	(satisfy demands)
$x_{p_d} = u_{p_d} f_d$	p=1, 2, ..., $\|P_d\|$; d=1, 2, ..., $\|D\|$	(diversity constraints)
$\sum_d \sum_p \delta_{ap_d} x_{p_d} \leq \sum_t m_{at} u_{at}$	a=1, 2, ..., $\|A\|$	(sufficient capacities)
$\sum_t u_{at} \leq 1$	a=1, 2, ..., $\|A\|$	(at-most-one)
$\sum_v u_{p_d} \alpha_{vp_d} \leq H_d + 1$	d=1, 2, ..., $\|D\|$; p=1, 2, ..., $\|P_d\|$	(hop-limit)
$\sum_{p_d} \alpha_{vp_d} u_{p_d} \leq 1$	$\begin{cases} d = 1, 2, \ldots, \|D\|; \\ v = 1, 2, \ldots, \|V\|; \\ v \neq s_d; v \neq t_d \end{cases}$	(node-disjoint)

2.4.3 Problem Formulation 2

We can simplify and rewrite the previous formulation as follows: *Given ${}^H P_d$ a set of hop limited $s-t$ K-paths for each demand $d \in D$:*

indices

v	$= 1, 2, \ldots, \|V\|$	nodes
a	$= 1, 2, \ldots, \|A\|$	arcs (directed links)
t	$= 1, 2, \ldots, \|T\|$	types of cables to be placed in arcs
d	$= 1, 2, \ldots, \|D\|$	demands
p'_d	$= 1, 2, \ldots, \|{}^H P_d\|$	index of candidate $s-t$ K-path to fulfil demand d

constants
$\delta_{ap'_d}$ =1, if arc a belongs to the K_d-path p'_d fulfilling demand d; 0, otherwise
$\alpha_{vp'_d}$ =1, if node v belongs to the K_d-path p'_d fulfilling demand d; 0, otherwise
s_d source node for demand d
t_d target node for demand d
k_d diversity factor (node connectivity between s_d and t_d) for demand d, it is implicit in the set ${}^H P_d$
f_d volume of demand d
ξ_{at} cost of the t-th type of cable on arc a
m_{at} capacity of the t-th type of cable on arc a
H_d maximum number of hops for demand d, it is implicit in the set ${}^H P_d$

variables
$u_{p'_d}$ binary variable indicating whether the K_d-path p'_d is chosen for demand d
u_{at} binary variable indicating whether the cable of type t is installed on arc a

objective
minimize $F = \sum_a \sum_t \xi_{at} u_{at}$

constraints

$$\sum_d \sum_{p'_d} \delta_{ap'_d} u_{p'_d} f_d \leq \sum_t m_{at} u_{at} \quad a=1,\, 2,\, \ldots,\, |A| \qquad \text{(sufficient capacities)}$$
$$\sum_{p'_d} u_{p'_d} = 1 \quad d=1,\, 2,\, \ldots,\, |D| \qquad \text{(one set of paths)}$$
$$\sum_t u_{at} \leq 1 \quad a=1,\, 2,\, \ldots,\, |A| \qquad \text{(at-most-one)}$$

Let us analyse the complexity of the last formulation: It has $(\sum_d |^H P_d|) + |A||T|$ variables, and $2|A|$ constraints. When $G(V, A)$ is a complete graph, it has $(\sum_d |^H P_d|)+(|V|^2-|V|)|T|$ variables and $2(|V|^2 - |V|)$ constraints, where $|^H P_d|$ is the number of hop limited K-path to be considered for the d-th demand. The complexity increases with $|^H P_d|$, and when considering all the possible K-paths it can be as big as $\sum_{l=k_d}^{l=V} \binom{V}{l} \frac{l!}{k_d!} f(l, k_d, H_d)$, where $f(l, k_d, H_d)$ is an integer function with $\binom{l}{k_d}$ as its upper bound (a detailed study is presented in Section A.4.1). This variable space is astronomical for most solution methods, but formulated this way the hop limitation and path diversity are respected implicitly, moreover it has only $O(|A|)$ constraints. In Chapter 3 we propose an approach to this problem using Genetic Algorithms, which are reported in different literatures to be able to search for good solutions even for problems with huge decision space size.

The feasibility of a proposed solution can be checked and evaluated in polynomial time and any instances of this problem are recognisable also in polynomial time, so it is in $\mathcal{NPO}$. Moreover, it is in $\mathcal{NPO}$-hard for being more general than some $\mathcal{NPO}$-complete problems; therefore, it is a $\mathcal{NPO}$-complete problem.

Observation: A possible intermediate formulation is to consider a set of $s-t$ K-paths for demand d instead of $^H P_d$, and leave the hop-limit condition as a constraint. That is a worse alternative as it has more constraints and $|^H P_d|$ is also bigger.

Observation: There exists an analogue problem where instead of arcs there are links. This variant is also $\mathcal{NPO}$-complete by a similar argument, and it is a general version of WMP and GSP-NC.

2.4.4 Related Work

In the previous section we showed a number of problems strongly related to TC-DHK, our network design problem, in particular WMP resembles ours most. WMP in [Bley 97] is just TC-DHK without the throughput constraint and capacity design (refer to page 11 for WMP related works and a list of easy simplified cases).

More recently, [Botton 07] presented a work with a similar problem in which the only difference with TC-DHK is that they have an edge-disjoint instead of node-disjoint constraint, and a linear cost for capacity usage. They use an exact approach, working with an extended formulation. The reported computational time is very small comparing to our problem, and it is remarkable how the difficulty of the problem may change as some constraints are removed or relaxed.

The study of complexity in the different particular cases, gives an insight into the complexity of our network design problem, as it is a general version of a number of $\mathcal{NPO}$-complete problems, the $\mathcal{NPO}$-hardiness is a direct result. For instance WMP and WSP are proven to be $\mathcal{NP}$-complete in [Itai 82], both are particular cases of TC-DHK. Therefore, to conclude that we have a $\mathcal{NPO}$-complete problem, we only have to prove that it is in $\mathcal{NPO}$, but the latter is trivial[16].

[16] Problem instances are recognisable, feasibility check is verifiable, and solutions values are computable all in polynomial time.

2.4.5 Particular Cases

In the following, we explain how some problems in the previous section are particular cases of the one studied in this thesis (or the analogue edge version problem).

MST: given $G(V, E)$ and $C : E \rightarrow R^+$. The edge version problem can be obtained as follows.

- Maintain the same graph $G(V, E)$;
- for each pair of nodes $(s_i, t_i) \in \{V, V\}$, create $(s_i, t_i, k_i = 1, H_i = |V| - 1, f_i = 1)$ and insert it into D;
- for each $e \in E$, create $(m_e = |V|, \xi_e = C(e))$ and place it in the e-th position of T;
- set $U = I$ the identity matrix.

SPG: given $G(V, E)$, $V_T \subset V$ and $C : E \rightarrow R^+$. The edge version problem can be obtained as follows.

- Maintain the same graph $G(V, E)$;
- for each pair of nodes $(s_i, t_i) \in \{V_T, V_T\}$, create $(s_i, t_i, k_i = 1, H_i = |V| - 1, f_i = 1)$ and insert it into D;
- for each $e \in E$, create $(m_e = |V|, \xi_e = C(e))$ and place it in the e-th position of T;
- set $U = I$ the identity matrix.

GSP-NC: given $G(V, E)$, $V_T \subset V$, $C : E \rightarrow R^+$, and $R_{|V_T| \times |V_T|}$ the connectivity requirement matrix. An edge version problem can be obtained as follows.

- Maintain the same graph $G(V, E)$;
- for each pair of nodes $(s_i, t_i) \in \{V_T, V_T\}$, create $(s_i, t_i, k_i = R(s_i, t_i), H_i = |V| - 1, f_i = 1)$ and insert it into D;
- for each $e \in E$, create $(m_e = |V|, \xi_e = C(e))$ and place it in the e-th position of T;
- set $U = I$ the identity matrix.

WMP: given $G(V, E)$, $C : E \rightarrow R^+$, and $D' = \{s_i, t_i, k_i, H_i\}$ the requirement set. An edge version problem can be obtained as follows.

- Maintain the same graph $G(V, E)$;
- for each pair of demand in $(s_i, t_i, k_i, H_i) \in D$, create $(s_i, t_i, k_i, H_i, f_i = 1)$ and insert it into D;
- for each $e \in E$, create $(m_e = |V|, \xi_e = C(e))$ and place it in the e-th position of T;
- set $U = I$ the identity matrix.

Chapter 3

Solving with Genetic Algorithms

3.1 Introduction

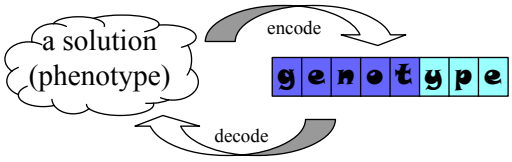
Genetic Algorithms is a meta-heuristic invented in the 1960's for solving optimisation problems, inspired on the natural selection mechanism described in Darwin's theory of evolution. With some degree of abstraction it tries to simulate the process of evolution, in which all existing life forms are descendants by generation of pre-existing forms or species, which undergo modifications. GA

When species are placed in an environment, those individuals or varieties that fit better in it have a greater chance to survive and produce offspring, in this way, individuals in the following generation carry fitter characteristics. Time after time, offspring appear with peculiarities not present in their progenitors, such an event is called mutation and it is a key component for the introduction of new varieties. In brief, natural selection acts by preservation and accumulation of beneficial variations of species in a given environment[Darwin 59].

GA has its origins back to the 1960's, and most notably when John Holland proposed explicitly a set of genetic operators together with the theory of schemes. Meanwhile, other evolutionary related strategies were also being developed like evolutionary strategy (Ingo Rechenberg, 1965) and evolutionary programming (L.J. Fogel, A.J. Owens and M.J. Walsh, 1966). For more details in the history of GA, refer for example to [Marczyk 04].

The following is the list of components common to every GA, they can be implemented in many different ways yielding different GA.

- The **coding** consists of the encoding/decoding operations; to encode is to transform a solution into a genotype, with its internal structure divided into genes. As genetic operators work with genes of a genotype, the coding together with its genetic operators play an important role in GA. Good coding may lead GA to solve a problem efficiently, while a bad one may mislead the search and result in a total failure. Decode is the reverse operation of encode, when applied to a genotype, it yields the original phenotype or solution of the problem.



  Examples of coding are: encode natural numbers as an array of bits; encode graph as array of bits representing the presence or absence of a certain component; encode a Hamilton cycle as a special permutation[1] of natural numbers representing cycles, etc.

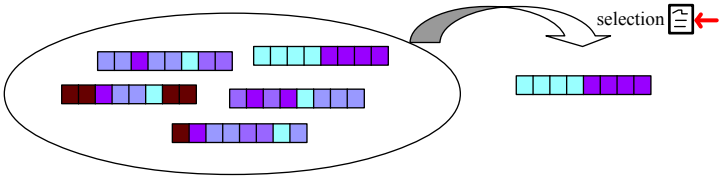
- A **fitness function** defines a virtual environment, it is used to evaluate phenotypes and determine their suitability in the environment. It is closely related to the objective function

[1] To represent a cycle, the use of permutation will lead to repeated representation, but by removing an fixed element and imposing that two positions have a given order we will end up with a permutation-like coding for cycles.

of the underlying optimisation problem, and usually is defined as a real non-negative function (fit solutions get high values).

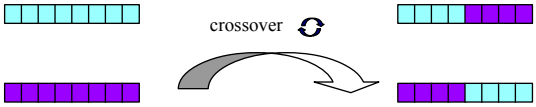
$$f : \{phenotype\} \rightarrow R^{+}$$

- The **selection strategy** specifies the way how the mating pool is conformed, by choosing individuals from the previous generation. It uses the fitness function to evaluate individuals and applies a certain strategy so that individuals with good fitness are more likely to survive then otherwise.

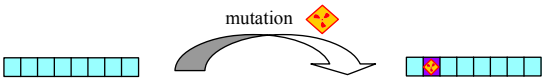


Examples of selection strategies are: roulette selection, ranking selection, tournament, etc.

- The **crossover operator** is the genetic operator that combines a number of progenitors to form offspring. By using genes that are present in different "good" parents, it tries to produce better or fitter offspring. Some example are: one-point crossover, N-point crossover, uniform crossover, etc. An issue about coding and genetic operators is the feasibility constraint, very often, combining feasible genotypes yields unfeasible ones. Three possible alternatives are available. The first one is to choose a coding that never decodes to an unfeasible solution; although this sounds simple it is not always possible, as the structure of some problems are very complex[2]. A second alternative is to implement correction or repair mechanism for unfeasibility. The third one is to allow the existence of unfeasible individuals, then mark them as unsuitable solutions with a penalty in the fitness function. Each of these alternatives has its advantages and disadvantages, depending on the problem in question, one may be more suitable than the others.



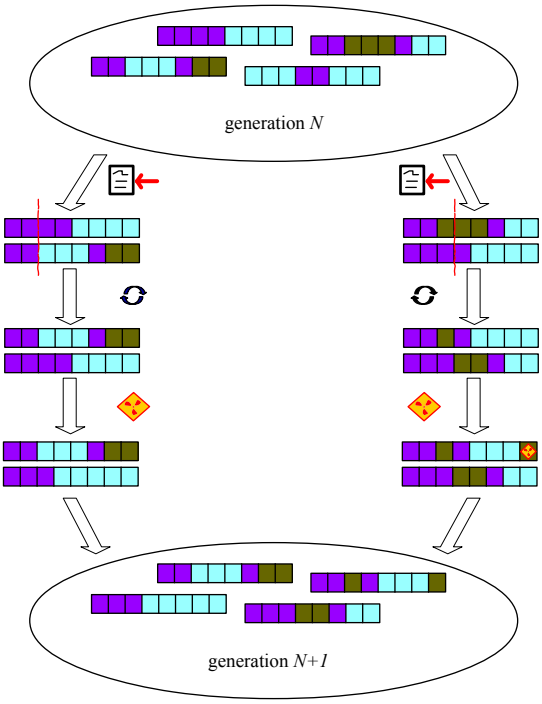
- The **mutation operator** is the genetic operator that introduces variation to the individuals of a population, and it is a key source of new genetic material in a GA. Without mutation, the simulated evolution may converge to a very limited set of genes, which is very undesirable as it leaves the rest of the solution space unexplored. To improve the efficiency of GA as a solver, it must explore the solution space as much as possible. On the other hand, a high mutation rate may lead to the destruction of good genes, and degrades the GA to some sort of random search. As a result, the mutation rate is an important parameter in GA; moreover, to fix it correctly, the number of genes per genotype should be taken into consideration, because the probability of a genotype surviving mutation depends directly on its length and the mutation rate.



Genetic Algorithms are stochastic search strategies that work with a **population of solutions**, and evolve it generation by generation with the help of genetic operators. To find good solutions for an optimisation problem, first, a set of solutions is encoded into a set of genotypes, and it forms the generation zero. Random generation or heuristics may yield this initial set of solutions. Then

[2]There are problems in which checking the feasibility of a single solution is easy, but to know if there is any solution is very hard (belonging to $\mathcal{NP}$-hard).

iteratively, the algorithm filters out and replaces members with low fitness. The surviving individuals form a mating pool, in which the crossover and mutation operations are applied to obtain the next generation.



3.2 Designing GA for our problem

In Section 2.4.2 the problem of interest is presented in detail, but before we proceed with the coding and design, note that the problem was defined for digraphs, and from now on we work only with digraphs. We are working with directed arc capacity and directed flow. However, it is possible to define an analogue problem in which the link capacity is undirected or duplex[3]. We will explain their difference and the required changes in the design when applying our solution method to this analogue problem later in Section 3.3.5.

Studying the alternate formulation in Section 2.4.3 we can see that the main decision variables are "the set of paths to use for each pair of $s-t$ nodes". Furthermore, once the set of paths to use is chosen, the type of cable can also be fixed by applying a min-cost polynomial time algorithm.

3.2.1 Traditional coding

We are dealing with a very complex problem, for which we could not find any previous work using genetic algorithms, therefore as a first step we analyse the traditional coding for other similar problems, to evaluate their suitability for our problem.

Graph structure coding: In our network design problem, one of the outputs is the structure of the graph. Putting aside the goal of obtaining the explicit routes to use, it is a pure topology and capacity design problem with a number of constraints. The traditional coding in GA for designing graph structure or topology, is an array of arcs (and/or nodes). It consists of an array of bits representing the presence or absence of each component of the graph; it also can be an array of natural numbers representing the absence or a particular choice (read hardware with different capacities

[3]In problems with undirected link capacity, the sum of flows in both directions of a link can not surpass an undirected (duplex) capacity.

and costs) for a particular arc (and/or node). As an example, in [Ahn 01] a GA based topology reconfiguration is proposed, it uses multi-dimensional array to encode graph topology. This kind of coding is very short and natural, but not quite suitable when there are unfeasible combinations. Unfortunately, our problem has capacity, path diversity and hop-limit constraints. And in general, the size of our problem's feasible solution space can be very small compared to the whole set of all combinations. Furthermore, even without the capacity constraints, to check the feasibility of path diversity together with hop-limit constraints (feasibility problem of WMP) is a $\mathcal{N}P$-hard problem in general. In other words, to use this kind of coding will include unfeasible solutions and we don't even have an efficient way to determine feasibility, therefore, we conclude that graph structure alone is not suitable for our problem.

Path coding: In our problem, if the routes for each $s-t$ node-pair are explicit, the feasibility check is trivial for both path diversity and hop-limit constraints. Some traditional coding to encode path are list of nodes or arcs. In [Nesmachnow 04, Calegari 02] the authors studied different alternatives and proposed some coding based on list of paths for the general Steiner problem with edge connectivity constraint (GSP-EC). To use this kind of coding, the genetic operators must include correction mechanism to avoid loops and maintain feasibility for the node-disjoint, hop-limit constraints.

All the previous codings have some serious problems when applied to our problem, so we devoted a part of this work to develop a new coding as a work around.

3.2.2 The proposed coding: to encode H-limited K-paths

What we need is a coding that can somehow simplify the feasibility check, and maintain the feasibility after the application of genetic operators without the overhead of complicated correction mechanisms. In other words, we want to codify solutions in our solution space without adding unfeasible solutions or bias (repeated representation) in the coding. It is to transform the solution space (K_i-path with hop limit H_i between each pairs of nodes s_i and t_i) using bijection to some more simple structure. In this way, each solution in the encoded space is a feasible solution according to the hop-limit, path diversity constraints and therefore no correction mechanism is required. From the K-paths to use and the throughput of each $s-t$ node-pair, the capacity constraint can be checked in polynomial time $O(|A|)$. Furthermore, if the K-paths of each $s-t$ node-pair is fixed, then the optimal topology and capacity for the graph can be determined easily, by building the sub-graph which contains only the nodes and arcs presented in those K-paths; computing over each arc the accumulated flow of the different K-paths that use it; and assigning as arc type the cheapest type of cable that can carry this accumulated flow.

***H*-limited *K*-path coding:** Two problems arise with this approach, in the first place, it still contains unfeasible solutions: when the accumulated flow assigned to an arc, from different demands, is larger than the largest available capacity, then it becomes an unfeasible solution. The second problem, "is there an efficient way to encode all the possible $s-t$ K-paths with hop-limit?". The former is very common in GA, it can be solved with penalty in the fitness function or with correction in genetic operators. And in the particular case of TC-DHK, we can simply require that the input problem has a large enough capacity on each one of the edges. As for the second issue, we proved that it is not possible for general graphs, because to be able to enumerate hop limited K-paths efficiently is to be able to solve the feasibility problem of WSP efficiently, which is proven to be $\mathcal{N}P$-complete in [Bley 97, Itai 82]. Even worse, a result from [Allender 99] states that the function that counts the total number of simple s-t paths in a generic directed path is a $\#P$-complete function, i.e., it is among the hardest counting function in the class $\#P$. Unless $\mathcal{N}P = P$, we will not be able to enumerate hop limited K-path for generic graphs, and it seems that generic graph is the primary source of the difficulty. By limiting the problem to complete graphs, we managed to develop a coding, which is one of the main results of the present thesis.

The proposed encode/decode functions are Θ_5 / Θ_5^{-1} from Appendix A, the latter is written as a standalone document including all the developed structures, combinatorial analysis and intermediate steps for the design of the bijection between hop limited K-paths and range of naturals.

With Θ_5 we can map H-limited K-paths in a complete graph into natural numbers, the latter can in turn be mapped into binary strings. But for the TC-DHK network design problem Section 2.4 we have as solution not one but a set of $|D|$ H-limited K-paths. At this point we have two immediate alternatives, one is to take the Cartesian product of the $|D|$ solution spaces and map solution in this giant space into number or binary string; the other is to treat each H-limited K-path as a component and represent the entire solution space as list of numbers. Through the remaining of this work, the former is named type I while the later type II. type I/II

Together with the two codings, we have to design the paired genetic operators. After investing so much effort into the development of the coding, one of the immediate result is that no correction mechanism is required here to re-establish feasibility of the different constraints.

Coding type I: A giant number representing a set of $|D|$ H-limited K-paths, that ends up being a string of bits. The simplicity of this representation is that it resembles the simple genetic algorithms and we can use the traditional operators.

- Selection: roulette selection (standard proportional selection);
- Crossover: uniform crossover;
- Mutation: uniform mutation.

After applying crossover and mutation, it is possible that the result falls outside the range of the solution space, because the size of the solution space usually is not a power of two (not all combination are feasible). To solve this problem, we detect and replace overruns with a random feasible solution (uniform distribution). For size that surpasses barely a power of two, this may lead to almost half of the representations being redundant and therefore require re-sampling, increasing in this way the randomness of the genetic algorithm. It is a weakness common to all bit string representations.
A clarification for the last statement: in the worst scenery when almost half of the representations require re-sampling, the number of re-sampling in practice is much lower than the redundant portion suggests. In fact the local search aspect of the G.A. will greatly reduce this number. Small mutation rate is another factor that helps to avoid re-sampling, because to trigger a re-sampling, it is necessary to have a mutation to occur in some particular positions.

Coding type II: Although the idea to represent H-limited K-path as numbers is new, coding using list of numbers is not. It is one of the traditional coding in the usage of genetic algorithms, therefore we don't have to invent new operators neither in this case.

- Selection: roulette selection (standard proportional selection);
- Crossover: uniform selection;
- Mutation: uniform mutation.

We choose those operators in an attempt to be as close as possible to the "simple genetic algorithms", the only difference a priori is that we replaced the one point crossover with the uniform crossover. The latter is reported as outperform the former in many different problems by different authors, and it is very understandable. One of the reasons is that one point crossover is more disruptive for long schemes in coding using list. Notice that the results from the crossover and mutation operators in this coding are always feasible. The mutation operator when applied to a gene, will always generate a result in the feasible range.

A question arises here: is one of the alternatives more suitable than the other? Comparisons can be found in the next chapter, among other experiments.

Fitness function: The traditional definition of fitness, is a function F_{fit} : phenotype $\rightarrow \mathbb{R}$, which gives high values to fit phenotypes. In a min-cost optimisation problem, a good candidate is the cost function. But we face here a dilemma, should we maintain the traditional definition of high-value-fit-phenotype or should we adapt our own library to admit the inverse relation, which is more natural for min-cost optimisation? We chose the latter and therefore we define the fitness function of our G.A. as the cost function of the problem, which takes a given graph (topology and capacity are

obtained directly from the H-limited K-path of each of the $|D|$ demands in $O(|D||V|)$ operations) and returns the total cost of all the chosen cables in all the presented arcs.

3.2.3 The particular implementation

We have two types of solution coding, type I represents solutions using binary strings (extremely long strings), type II represents using list of big numbers.

During the experiment we used the basic GGEngine v1.0 and extended it with a number of functionalities, some of the new additions didn't yield better results for this problem, but for those that did, we retained them as new features in the second version of GGEngine (the technical report on GGEngine v2.0 can be found in Appendix B). The main difference between our proposal and a traditional G.A. are the following:

- customised mechanism of fitness scaling,
- dynamic mutation rate,
- usage of elitism.

All those improvements and their parameters are result of a series of undocumented trials, in the search of mechanisms that can improve both the original performance in solution quality and run time. Those experiments were performed with the small test cases, using different random seeds, and we included only those mechanisms that dominated the others after running a fixed number of generations.

Customised fitness scaling: Traditionally, the fitness function is defined as a measure that matches fit individuals to high values. We have a minimisation problem and the fitness scaling not only scale the function, but also inverse the direction of the objective. The fitness scaling function in use resembles the classical linear scaling [Goldberg 89]

$f_{scaled} = a \times f + b$ $\qquad$ a, b are coefficients, f the raw fitness, f_{scaled} the scaled fitness

If we consider the population of a given generation as samples in the solution space, then we have their associated costs (fitness values) in a range $[f_{min}, f_{max}]$. The idea is to determine the coefficients a, b to transform that range into a scaled range linearly. At the beginning of the G.A. to promote genetic diversity we choose the scaled range to be very small ([99,100]), and then it expands slowly generation after generation. Moreover, the stopping condition in use is by fixed number of generation, and after half of the generations are passed, the expansion rate is increased, and it stops after reaching a target range ([10,100]). The objective of the dynamic expansion rate is to reach the target range a fraction of generations before the end.

Dynamic mutation rate and elitism: In the classic G.A., mutation rate is fixed. Mutation is an important genetic operator in the introduction of variation, associated to the need of variation is the premature convergence problem. If the population converges to a suboptimal solution (which can be very far away from the optima in terms of objective value, and genetic information), then the crossover operator is useless in the introduction of new information. It leaves the mutation to be the only source of variation, which in turn depends strongly on the mutation rate. For small mutation rates, although the probability of mutating is non-zero, it may take a lot of generations before a better individual appears, but then it may surely converge in the next few generations. On the other hand, high mutation rates are too disruptive, in fact they degrade G.A. to random search. We can also observe this from the theorem of schemes[Goldberg 89], if the probability of disruption is too high, the lower bound for the number of copies that match a given scheme reduces to zero.
So we propose to have dynamic mutation rates, when convergence is detected, we increase the mutation rate, because if the proportion of best individuals exceeds a certain critical limit, without the intervention of high mutation rate, the roulette selection will ensure complete convergence. We decided to use the proportion of best individuals to control the mutation rate, we also included elitism to conserve the best-so-far individuals of the evolution. After some trials, we found that the best strategy for dynamic mutation is to make the changes smoothly, and maintain the mutation rate

fixed at least a number of generations (five generations). We define a range in which the mutation rate can vary and make small increases when the proportion of best surpass the limit of 10% of the population, and small decreases otherwise. Changes in the mutation rate are maintained for at least five generations.

Beside those additional mechanisms, we have a traditional G.A., where the coding represents a set of H-limited K-path, in either type I or II. During the evaluation of fitness, the path is decoded and an auxiliary graph is constructed to accumulate the flow assigned to each arc. Then capacity is assigned to each arc, by taking the most economic arc type (we discard all the dominated capacity-cost arc types in pre-processing, leaving this step trivial). The sum over all the assigned capacity is the cost or raw fitness of the solution.

3.3 Implementation

3.3.1 Other genetic algorithm engines

The skeleton or core of genetic algorithms usually is invariant among different instances, and it forms a big portion of reusable codes. To encourage the experimentation of genetic algorithm and to reduce the effort of implementation when it is applied to solve a problem, there are a number of genetic algorithms engines in the open source community, among the pioneers there is genesis (1990, by John Grefenstette[Grefenstette 90]), based on them there are developments like dgenesis (1993, by Erick Cantu-Paz[Cantus-Paz 94]), among many others. There are also other frameworks like the MALLBA Project [Univ. de Málaga 07](nowadays, by the universities of Malaga, La Laguna and Barcelona) which offers a list of solvers for the user to choose from, including G.A. as one of the options.

3.3.2 GGEngine - Generic Genetic Engine

In 2002 we developed the first version of GGEngine (Generic Genetic Engine) [Ho 02] a C library, with the aim to offer a light weight and flexible solver for researchers of our team. The objective is to have a solver that can be extended later to support the different distributed and parallel computing infrastructures available in our laboratory. This implementation offers a large list of popular codings and genetic operators, reducing the required implementation when solving a problem. Moreover, if both the generic operators and structures required for coding are already implemented, then extra implementation required when applying G.A. to solve a problem is reduced to

- a definition of the phenotype;
- the encoding function: phenotype $\rightarrow$ genotype;
- the decoding function: genotype $\rightarrow$ phenotype;
- the fitness function: phenotype $\rightarrow \mathbb{R}$.

The following list presents a summary of the features in GGEngine v1.0:

- genes coding: binary, integer, float, and user extensible;
- chromosome type: list of genes (of any type);
- selection operator: roulette (standard proportional selection), tournament, ranking, and user defined selection;
- crossover operator: one, two and n point crossover [Goldberg 89], uniform crossover [Syswerda 89], biased crossover, and user defined crossover;
- mutation operator: uniform mutation, bit representation specified uniform mutation, and user defined mutation;

- no recompilation required when changing coding or genetic operators (user defined components requires stand alone compilation and then linking).

After the first release of GGEngine, we received reports from users (primary from colleagues in our faculty) with suggestions to improve the performance and the addition of new features, like fitness scaling, different implementations of fitness caching and modification to the implementation of mutation operators (replacing multiple Bernoulli trials with binomial trials). In the ongoing version, all these changes are applied, together with other features that we introduced during the development of the present thesis, like the option of elitism, dynamic mutation rate, new random generator. Being an ongoing development there are still features in the to-do list, like support for parallelism, among others.

3.3.3 The GMP library

To implement the coding of H-limited K-path proposed in the previous sections, the standard arithmetic operators in C will limit the representation to small graphs, therefore it is necessary the usage of library that implements those operations for larger numbers. After a brief search, we came across GMP (GNU Multiple Precision Library, http://gmplib.org/), a GNU C library that offers implementation of arithmetic for big integer and reported to be one of the fastest implementations (at least among free libraries). Moreover, it is included in many distribution of Linux, and supports (optimised over) a large number of hardware platforms. The version used during the beginning of the development was 4.1.4 and later updated to 4.2.1.

3.3.4 Pseudo Random Number Generator

The generation of random numbers or more strictly pseudo random numbers is an important component of a genetic algorithms engine. In the current version of GGEngine, two separate random generators are used. The first one is provided by the dependant library GMP, it uses a linear congruent method to generate numbers, and long bit length numbers are formed by concatenating multiple results.

The second internal generator is used for all non-GMP related data types. In our search of good modern uniform generators, we came across the Mersenne Twister [Matsumoto 98]. And in the web site [Matsumoto 06] of its author, we located an implementation (mt19937ar) which is used as our internal generator. The Mersenne Twister is a 32 bits integer generator with a period of $2^{19937} - 1$, it requires 624 words of memory and its speed is reported to be comparable to other generators.

For a valid statistical study, it is necessary to alter the sequence of random number in use. It can be done by initialising the generator with different seeds, in our case two different seeds are needed as there are two generators. The linear congruent algorithm generates numbers with a formula like $X_{n+1} = (AX_n + B)\ mod\ C$, where A, B, C are constants and the first X_0 is the seed. To generate different sequence, one way is to choose seeds that are far apart from each other according to the last formula. In the case of Mersenne Twister, seeds with small Hamming distance or similar bit patterns may produce shifted sequences. Note that in the case of linear congruent method, any pair of seeds produce shifted sequences. We used seven pairs of seeds in our study, their values are presented in Table 3.3.4.

3.3.5 For the problem with duplex capacity

The design and implementation shown in this chapter was over the problem formulation in Section 2.4.3, which requires that:

$$\sum_d \sum_{p_d} \delta_{ap_d} f_d u_{p_d} \leq \sum_t m_{at} u_{at} \qquad a=1, 2, \ldots, |A| \qquad \text{(sufficient capacities)}$$

In each arc, the aggregated flow from different demands must be smaller than the assigned capacity. The capacity in question is directed, or arc related. A possible variant is to consider the duplex channels, in which the capacity is undirected or link related. Then the aggregated flow should be the sum over flows on both directions.

Our proposed solution method can be applied to this variant with a small modification in the stage of capacity assignment during the calculation of fitness. For each link, the capacity to assign is the most economic one that can hold the aggregated flow from both directions.

Seeds used in experiments

Group	MT seed	GMP seed
A	00000000	00000000000000000000000000000000000000000000000000000000000000000000000000000001
B	ABC01F23	c8bbdd822a45386dc74be73c34e71e75bf791993f37f08cc282592ee498a4335
C	F2E3DCA3	e84d98bba8c4b9956dbf54dc40daf4e480fe4c9e05642033d3b32848d236a25b
D	4EF5A126	f539876331a07f46599dbb2795c2641fbf4b51507ac9d255ac73c0db4c5ce0ba
E	622FE1C0	b86587be451a15ab2585eb8fb3dc79473677ca2af76254cf3f6ffbbfe686546a
F	93F4C6B1	d78a6ad6fdb350485079f225f7796f384255757e4d7b910cb83055265d03da8b
G	03F46B89	43e1635a78b12507e3212187d4b03d16bfbb0ff02f4b5f0899892fc988a29667

Table 3.1: "nr.dat" is an input file for GGEngine and it should contain the seeds as integers, but the same data is displayed here in HEX as it's much shorter.

Chapter 4

Experimental results

4.1 Introduction

A number of questions are posed and scattered over the previous chapters, find here a brief summary which serves as introduction to the chapter.

The main question is about the performance of the proposed G.A. using the coding Θ_5 / Θ_5^{-1} from Appendix A to solve the problem of interest. Being G.A. the chosen solution method, we have to face the problem of parameter fixing. One of the first steps is to study and determine relationships between properties in the problem instance and parameters in the algorithm. We have to check a number of parameters in G.A., in the short list we have: population size, mating pool size, crossover rate, mutation rate, number of generations. We run experiments with relatively few generations for most of the tests in this part of the study. We also have two variations of the proposed coding, is one of them more suitable then the other? We planed a series of test to address these questions.

To be able to compare the different solution methods and/or parameter settings, first we have to select or define the measures to use; we can find in Section 4.2.3 a survey and the selection of measures. Another fundamental piece in any experiment is the test suite to use, therefore, we will dedicate Section 4.2.1 to detail it and Section 4.2.2 presents the computational infrastructure used for obtaining the numerical results.

Section 4.3 presents all the experiments and their results. It is divided into several subsections. First of all, we concentrate on parameter fixing (Section 4.3.1), then we proceed to compare the two alternative codings to evaluate their differences (Section 4.3.2). Afterwards the proposed method is compared against random search (Section 4.3.4). Finally, we present the results of using pre-processing as an alternative to improve the performance for the method, specially needed in the case of medium and large instances (Section 4.3.5).

4.2 Experimental design

4.2.1 About the test cases: basic & composite cases

The suite of test cases consists of basic cases with known optimal solution and composite cases which are compositions of basic cases. As source of basic cases, we considered the possibility of converting known solutions of similar problems (add the missing inputs in a biased way to retain known solution), design graphs with special structure (with known optima by construction), or generation of random problems (have to run exhaustive search for optima). We implement only the last one of them, even when it was the most cpu-time consuming among all the alternatives, because it yields instances that are more natural -without extremely biased costs-. So the basic cases (A to G, see the details in Appendix D) consist of complete graphs with a set of capacity-cost types, those were assigned with uniform distribution between each pair of nodes. An additional (or fictional) link with unlimited capacity is added to ensure feasibility, the cost for this exceptional link type is linear with capacity usage, opposite to the other discrete link types. This fictional link is used to penalise the solutions that excess the maximum capacity limit in some link, notice that for problems in which

the maximum capacities in every link are enough to carry the total demand, this fictional link will never participate and therefore will be redundant. In some undocumented experiments, we used high discrete cost/capacity for the fictional link, but later, we realised that linear cost is a better alternative, because the discrete one forms some very hard to escape local optima (an example is shown in 4.1).

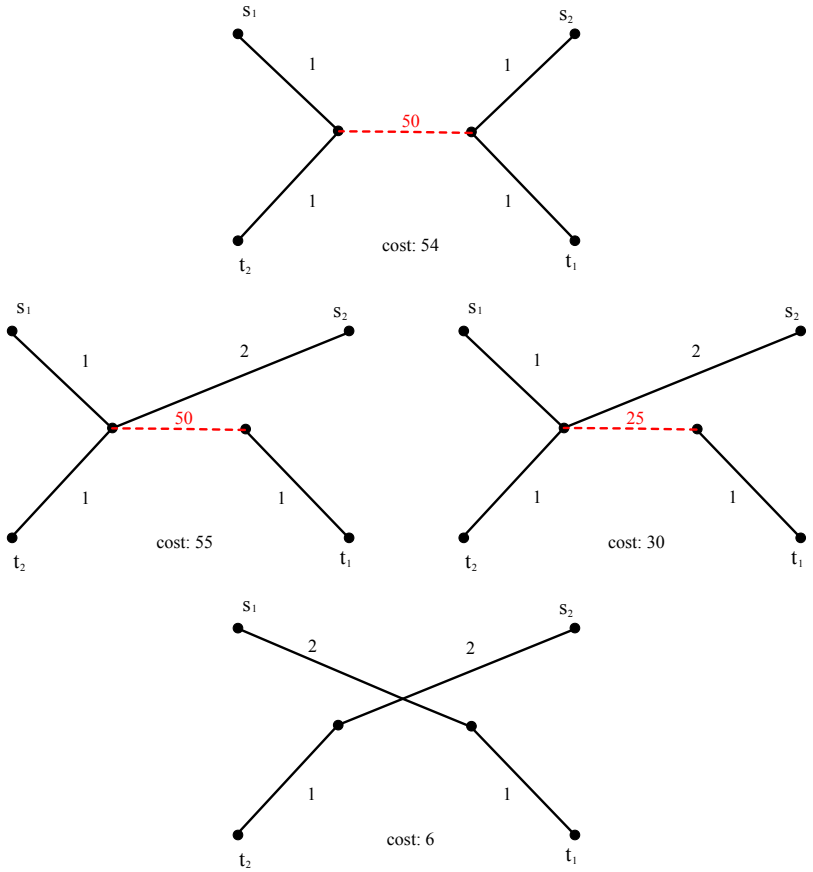


Figure 4.1: Here we show an example in a sub-graph with 2 demand pairs, and an already saturated dotted link. The top figure shows both demands using the dotted link, which is more expensive then the bottom-most figure. On the left we use the discrete cost/capacity for the fictional link, in which the removal of one demand does not decrease the cost. While on the right, we use a linear cost/capacity, therefore the removal of one demand decreases the link cost. In this example we show that the discrete model forms some local optima that are not present in the linear one.

The usage of exhaustive search in the construction of exact solutions limits the number of generated cases and their size, therefore we have to construct bigger instances by composition, in this way we have bigger test cases with known solutions. All the basic cases can be found in Appendix D, they are presented with the input file's format, a summary of their properties can be found in Table 21

Composite cases are constructed by extending a basic (or composite) case of N_1 nodes with another basic case of N_2 nodes, it will result in a complete graph of $(N_1 + N_2 - 1)$ nodes. We want the solutions of the resulting problem to be connected, that's why we require that one node must be shared by some demand pairs $s - t$ of both problems. On the other hand, we want the size of the composite problem to be as big as possible, it means that all the other nodes in the original problems are transformed into disjoint nodes. When more then two cases participate in the composition, there are three different ways to do it: by using different joint node each time, by using the same joint node each time, or hybrid of the two former (see Fig. 4.2). There will not be any demand other than the ones appearing in the basic components, and the available capacity type outside those basic

components is limited to the high cost fictional capacity link. Moreover, we set the cost of those links with fictional capacity in such a way that the usage of any of them will imply a total cost higher than the known optima.

Although the optimal solutions for this style of composition are graphs with pivoting nodes, which is a particular and undesirable structure for WAN, we decide to use them anyway, because they offer a good challenge to the solver, and the complexity is not reduced[1]. The optima cost in the composition is the sum of the optima costs in its components, and the total number of optima solutions is the size of the Cartesian product over its components' optima solutions sets. In Table 21 and Table 4.1 we show the huge size of the solution space of these composite problems.

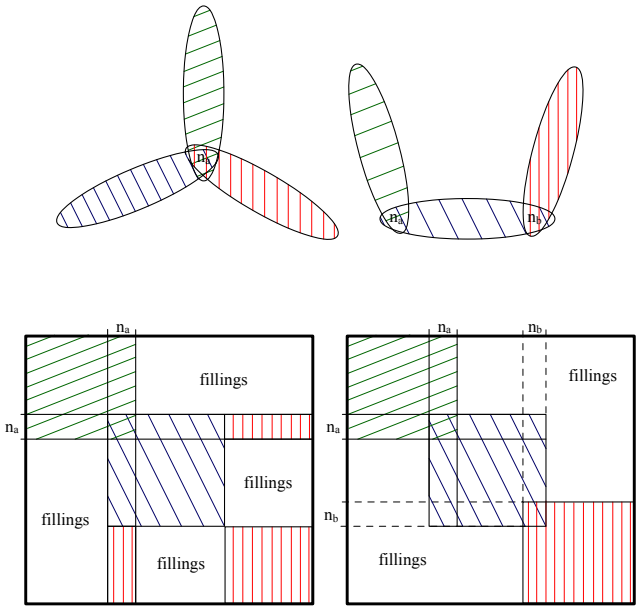


Figure 4.2: The two variants of composition and their matrix representation (hybrid of them is also possible when more than 3 components participate in the composition). The problem has matrices for available edge types, and demand (source to destination throughput, path diversity, hop-limit), the "fillings" for the demand related matrices are all zero, while for the available link type it is limited to the fictional capacity type.

G.A. is strongly based on random number generation, when studying a random algorithm, it is necessary the usage of different seeds for its pseudo-random generator, to increase the reliability and reduce the luck factor. To evaluate the proposed G.A. we selected seven different seeds and replicated each run seven times, once for each seed. Seven may be considered small for a strict statistical study, but increasing it further means also increasing the total runtime for the experiment, so we decided to keep it small for now. Find in Table 3.3.4 the seven different seeds used in the experiment.

[1] The size of the solution space is invariant with the costs, it means that if we lower the cost of some capacity type, we will have exactly the same space to explore, although, the number of local optima or their depth may be changed.

Properties of basic and composite cases

	$\|V\|$	$\|D\|$	opt. cost	# of opt. sol.	solution space size
A	6	6	43	7	788050640
B	6	6	55	3	327688400
C	6	5	70	3	430941160
D	6	6	142	7	304538000
E	7	4	64	5	714918000
F	7	4	55	4	737256000
G	7	3	43	5	376856000

	$\|V\|$	$\|D\|$	opt. cost	# of opt. sol.	solution space size
AA	11	12	86	49	8.8×10^{39}
BB	11	12	110	9	1.5×10^{44}
CC	11	10	140	9	2.8×10^{50}
DD	11	12	284	49	7.0×10^{57}
EE	13	8	128	15	9.0×10^{46}
FF	13	8	110	16	5.1×10^{42}
GG	13	6	86	25	5.0×10^{49}
AAA	16	18	129	343	4.2×10^{78}
BBB	16	18	165	27	4.6×10^{93}
CCC	16	15	210	27	2.1×10^{111}
DDD	16	18	426	343	1.3×10^{125}
EEE	19	12	192	125	3.7×10^{105}
FFF	19	12	165	64	4.7×10^{90}
GGG	19	9	129	125	2.6×10^{115}

Solution space size (the exact big number)

	0987654321098765432109876543210987654321098765432109876543210987654321098765432109876543210987654321098765432109876543210987654321
AA	875454603924737281449963388655571984000
BB	151367119745118756031542650295667101696000000
CC	28211812565229960299468929209482156803401252864000
DD	6985777610356914043526609879578500671439561004880625000000
EE	900923932279489664246061529629129146284857600000
FF	506923348905469138335196420101889034496000
GG	503315273527436234541125835408753027226075009600000
AAA	41629009630241036533274894129349305088064908556254376652106232105468345808716250
BBB	46496251598164805084269949619001383450621317537690616001006594963853466437950648664032416093750
CCC	20641787748049573191951417673608372160988768355402265914020866581418397410585136156133674182384925876303774708750
DDD	12739373112978729805503221282678809544955547919223353502541037827122821861307110671548030688757226698414842536388967634960937500
EEE	365780561326105127254736867370691929298595494084855385153348483268444398854561997502144870717849600000000000
FFF	472367773109213248773495016408393354319957818973675442818355796576068718827088011224678400000
GGG	26269863520287316617055704700597293808626321458521485005938880939440591911997539557671484649750683584041930719232000

Table 4.1: The solution space size is calculated by taking the product of the size for each individual demand.

Hostname	CPU type	RAM capacity
crysnife	AMD Duron 1.1GHz	512MB
homePC	Intel Pentium IV 2.0GHz	256MB
lorien	Intel Xeon 2.4GHz	1GB
smgPC	Intel Pentium IV 3.06GHz (HT)	1GB
hydraNode	AMD Opteron 880 2.4GHz (x8)	32GB

Table 4.2: Hardware features of test platforms.

4.2.2 Infrastructure for testing

During the development of the experiment, different computers become available to us, they have different features. Some remarkable ones in order of appearance are crysknife, homePC, lorien, smgPC, hydraNode, see Table 4.2.

The construction of basic case and their exact solutions were performed with crysnife, homePC and lorien; while the general testing were done with all of them in different period of the testing. We used different operating system in them, all of them are distributions of linux, ranging from Redhat, Fedora-Core, Suse, to HP XC. To admit the coexistence of test results that come from such a heterogeneous set of platforms, we decided not to take into account the traditional measure: run-time. Instead of it, we searched for another more suitable measure to express delay or CPU consumption. After making a profile of the code, we discovered that a huge portion of the time is used in the evaluation of fitness, this allows us to replace easily the runtime by the number of evaluations. We have to remark that the counting of function calls instead of runtime is not a new idea, many other researchers have been using it in the past in different scenes. To give an idea of the conversion of evaluation to the traditional time, it takes 40,000 sec to perform in hydraNode a complete test suit consisting of all the cases AA-GG and AAA-GGG, with 500×60,000 evaluations in each case. In other words, it takes in average 1 sec for 12,750 evaluations to be done in a single processor of hydraNode. The drawback of this approach is that we had to disable fitness caching, to gain a complete control over the number of evaluations to perform. This left-aside feature can save more than 30% of the evaluations in some cases, which could be directly translated into a reduction of runtime.

Another result obtained from the profiling is that the memory requirement is relatively low in all those experiments; for composition of up to 3 basic cases, it requires only about 50MB of maximum memory. Therefore, no swap memory was used during the experiment.

4.2.3 Measures

Before being able to evaluate the results, we have to choose or define some measures for the evaluation of solution quality. Measures defined for a maximisation problems can be transformed and used for minimisation ones easily, from now on we will use the minimisation version for all the measures in discussion. As a first approach we can consider some simple measures like the average best, the best, and the standard deviation from the results obtained for a same problem instance, using different seeds in the pseudo-random generator. These three measures can tell how consistent is the solution method, or in other words, how likely does it obtain similar results using different seeds. We cannot tell if the method gives a good solution or not, without comparing these results against (the cost in) a $\{best\ known\ solution\}$, which can be a known optima or a bound. Further refinement is possible by using the $\{best\ known\ solution\}$ of a problem instance to normalise the measure somehow.

While choosing measures to use, we observed many problems with some traditional measures, those problems are related to how easy is to cheat and bias the results.

In the first place, the traditional way to compare the results against a $\{best\ known\ solution\}$ is defined as the gap, that is the distance between the result and a $\{best\ known\ solution\}$.

$$\{gap\} = \{given\ solution\} - \{best\ known\ solution\} \qquad \text{gap}$$

The gap alone is just a distance, and the smaller it is, the better is the result. But how small should such distance be, to be acceptable? To answer that question, it is necessary to add something else

in the comparison. Many researchers compare it against the value of the $\{best\ known\ solution\}$, and get the definition of the gap in percentage as:

$$\{gap\%\} = \frac{\{given\ solution\} - \{best\ known\ solution\}}{\{best\ known\ solution\}} \quad \text{gap\%}$$

Now the gap is normalised with the value of the $\{best\ known\ solution\}$, but is it a fair measure? We found a serious problem related to this measure, that is, it is very easy to cheat. By adding or subtracting a fixed cost to the problem, one can change the $\{gap\}$, as in the following example:

> Given an experiment that yields a solution with cost 11, while the optimal cost is 10
>
> $$\frac{11-10}{10} = 0.10 = 10\% \text{ in gap}$$
> $$\frac{(11+C)-(10+C)}{(10+C)} = \frac{11-10}{(10+C)}$$
>
> taking $C = 10$ we get 5%; or $C = 90$ to get 1%
>
> If the experiment is TSP we can add the offset easily by taking a particular node, and add $\frac{C}{2}$ to the cost of all its edges.

By simply adding an offset or fixed global cost, which can be done in a pre-processing or post-processing procedure external to the solution method, the result can become very different. Moreover, this measure doesn't take into account the hardness of the problem instance, if its worst solution's cost is close to the optimal cost, then it will always give good result no matter how bad the solution method is. But then, the gap in percentage can be very good while the method in study is bad. Even when the gap is widely used in many literatures, it is not a fair measure, therefore, we continue our search for a more meaningful measure.

Another traditional measure is "solution quality", the name induces the understanding of high quality corresponds to good solutions, while bad solutions have low ones. And solutions with lower cost have better quality than expensive ones, in a minimisation problem. Some authors define quality in terms of percentage as:

$$\{traditional\ quality\} = \frac{\{optimal\ solution\}}{\{given\ solution\}} \quad \text{traditional quality}$$

When the given solution is optimal, the quality is 100%, and for more expensive solutions the value of quality is lower than 100%. This measure is fine for evaluation of solution, but to evaluate methods, it doesn't take into account the difficulty of the problem nor the work done by the solution method. For instance, if we apply a bad method to solve a very easy problem in which the worst possible solution has high quality, then the final result will always has good quality. On the other hand, for difficult problems, like those that have a big difference between the optima and the second best solution, this measure will be very unfavourable to good solution methods, unless they yield the optima. Once again, the trick of adding a fix cost can also be applied to modify the results of an experiment. For the same test suite and same solution method, subtracting or adding a fixed cost to all solution can easily modify the result of the solution quality.

Once understood the problems with the previous measures, we considered a number of alternatives in an attempt to find a suitable one for our experiments. If we know both the best and worst possible solutions' costs inside the solution space, we can define a measure for the quality of a given solution as:

$$\{quality\} = \frac{\{worst\ solution\} - \{given\ solution\}}{\{worst\ solution\} - \{best\ solution\}} \quad \text{quality}$$

In our network design problem, one may tend to consider the worst possible graph as the complete graph with the highest possible capacities, but it is much more expensive than the worst from the search space. As we want to study the solution quality of our algorithm in a specific search space, we will consider the worst possible solution inside this search space. In which the capacity (and therefore the cost) is set to the minimum necessary for each arc, for a given combination of hop limited K-paths. We can express this measure in percentage, which allows us to study quality without worrying about the magnitude or scale of different problems. Cheating by addition or

subtraction of fixed cost is no longer possible. This measure will yield 100% for an optima solution, and 0% for the worst solution.

But most of the times we do not know the best or worst solution of a problem, in particular, we are working with composite cases where only the best solution is known by construction. So we have to define another measure:

$$\{approximated\ quality\} = \frac{\{worst\ known\ solution\} - \{given\ solution\}}{\{worst\ known\ solution\} - \{best\ known\ solution\}}$$

It is easy to check that $\{approximated\ quality\} \leq \{quality\}$, therefore it is a lower bound for the real quality of the algorithm. From now on, we will call $\{approximated\ quality\}$ as A.Q. in short. A.Q.

We noticed that one can still manipulate the results by adding an extremely expensive solution somehow to the solution space, and then the measure will become meaningless like the formers. To overcome this trick, we can replace the $\{worst\ known\ solution\}$ in the A.Q.'s formula, by the result of a trivial method -best solution from a random search with a certain number of evaluations-. More precisely, we will use the same number of evaluation in the random search as in the proposed G.A., and as $\{best\ known\ solution\}$ we will use the optimal solution's cost. The usage of random search in the comparison has the effect of sampling the solution space, it is an easy way to take into account the hardiness of the problem. The A.Q. defined this way is very robust to cheatings, and it is one of the primary measure applied in out the experiments. To study the cost of a given solution, we define A.Q. respect to random search as:

$$\{A.Q.\ above\ R.S.\} = \frac{\{random\ search\ solution\} - \{given\ solution\}}{\{random\ search\ solution\} - \{known\ optimal\ solution\}}$$

A.Q. above R.S.

Notice that, for methods that perform worse than random search, the resulting A.Q. above R.S. will be negative. Furthermore, the runtime in our experiment is measured by the number of objective evaluations; therefore it is meaningful to make comparison against random search of a certain number of evaluations. Otherwise, the comparison should be based on runtime or other measure of CPU consumption. And finally, a definition problem may present when the cost of the solution obtained with random search (of a certain number of evaluations), matches the optimal cost, but then it is a problem easy enough for random search to solve, in other words, it does not qualify as a test case for the evaluation of solution method.

4.3 Results

4.3.1 Parameter fixing

Before proceeding to experiment with G.A., first, we try to determine the set of parameter to use. This step is important specially after applying modifications to the traditional G.A., the idea is to run a small number of evaluations using different set of parameters and then choose the soundest of them. As there are two encodings implemented, we have to fix the parameters for each one of them. We consider the following short list of parameters to fix, because all the others are already fixed or adjusted dynamically:

- population size;
- mating pool size;
- crossover rate.

To fix the crossover rate, we performed some undocumented testing and the soundest value is 0.7, which means that 30% of the next generation is copied directly from the parents, while 70% of it pass through the crossover operation.

At this point, only two parameters have to be fixed, those are population and mating pool size. We chose three values for each one of them and experimented with all the nine combinations, with the number of evaluations fixed to 5,000,000.

coding type I

There are nine combinations of parameters, and fourteen test cases, therefore we have in total $9 \times 14 = 126$ experiments. Each experiment is run with seven different random seeds, and we register the mean value of their resulting cost. We show in Fig. 4.3 and Fig. 4.4 the graph of comparison for each experiment individually; then in Table 4.3, Table 4.4 and Table 4.5 the comparison of the fourteen problems altogether. We summarise here some of the observations:

- population size equals to 500 yields the worst result in $29/(3 \times 14) = 70\%$ of the cases (Table 4.4). This is caused by the fixed number of evaluations, the number of generations decreases as the size of the population increases.
- mating pool size of 80% yields the worst result in $27/(3 \times 14) = 64\%$ of the cases.
- in G.A. the best solution to report should never appear for the first time in the last generation, moreover, the moment when it first appeared (namely F.A.) is a good indicator of the potential of further improvement. Studying the mean F.A. of the fourteen problems Table 4.5, we can observe that the potential of improvement is poor in general. F.A.
- In Table 4.3 we show the number of times each combinations appeared as the best configuration in a problem. One conclusion is that small populations with small mating pool performed[2] always better than big population with big mating pool. This result is also consistent when we study the Table 4.4.
- we observed that the most consistent configuration for the mating pool size is when it is set to 30% of the total population.

As a result, we fix the configuration to population size equals to 200, mating pool size equals to 60, for this series of composite problems (AA, ..., GGG).

[2]Performance in terms of achieved A.Q., for a small and fixed number of evaluations.

Type I: Best configuration counting

		population size 200	300	500	for fixed P.S.
mating pool size	30%	5	3	1	26
	50%	2	2	0	9
	80%	1	0	0	7
for fixed M.P.S.		26	14	2	

Table 4.3: Each of the fourteen test cases (AA,..,GGG) yields nine combinations of parameters ($14 \times 9 = 126$ experiments in total). In this comparison, we account the best configuration among the mean best cost of 7 different runs for each experiment. The central table shows the frequency a certain combination being the best among the nine possible combinations, for a same test case; the table on the right border compares mating pool size for fixed population size, therefore, each entry is the best configuration among three possible options; the table on the bottom compares the population size for fixed mutation rate.

Type I: Worst configuration counting

		population size 200	300	500	for fixed P.S.
mating pool size	30%	0	0	0	6
	50%	0	0	2	9
	80%	1	4	7	27
for fixed M.P.S.		3	10	29	

Table 4.4: The same as Fig. 4.3, but accounts the worst configuration.

Type I: First appearance of the last solution

Pop. size	mean F.A. in # of generations	total # of generations	F.A. in percentage
200	13850.0	25000	55.40%
300	9262.7	16666	55.58%
500	4814.5	10000	48.14%

Table 4.5: F.A.=first appearance of the best solution. The mean is taken from all the experiments with fixed population size.

Type I: nn compositions

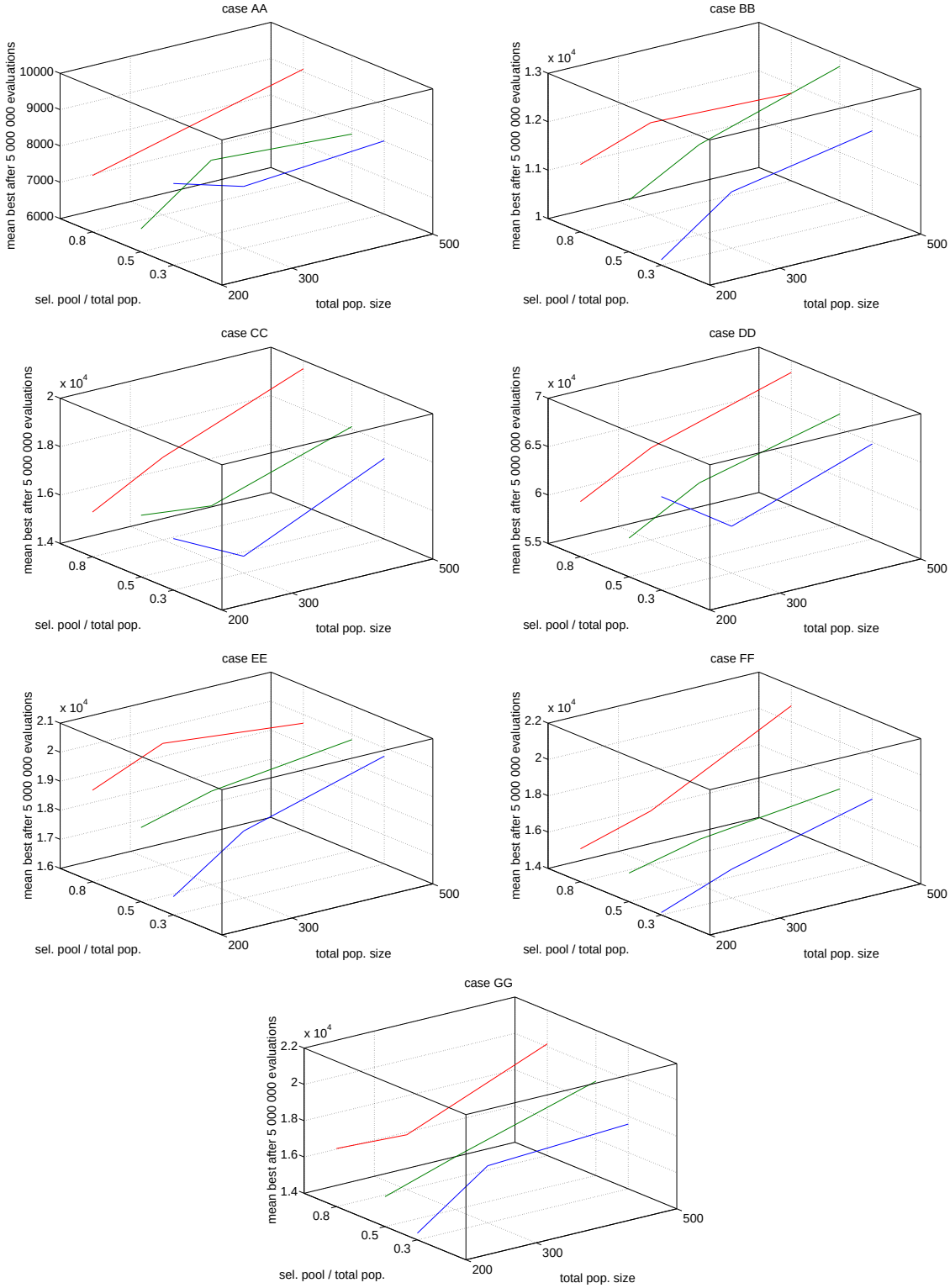


Figure 4.3:

Type I: nnn compositions

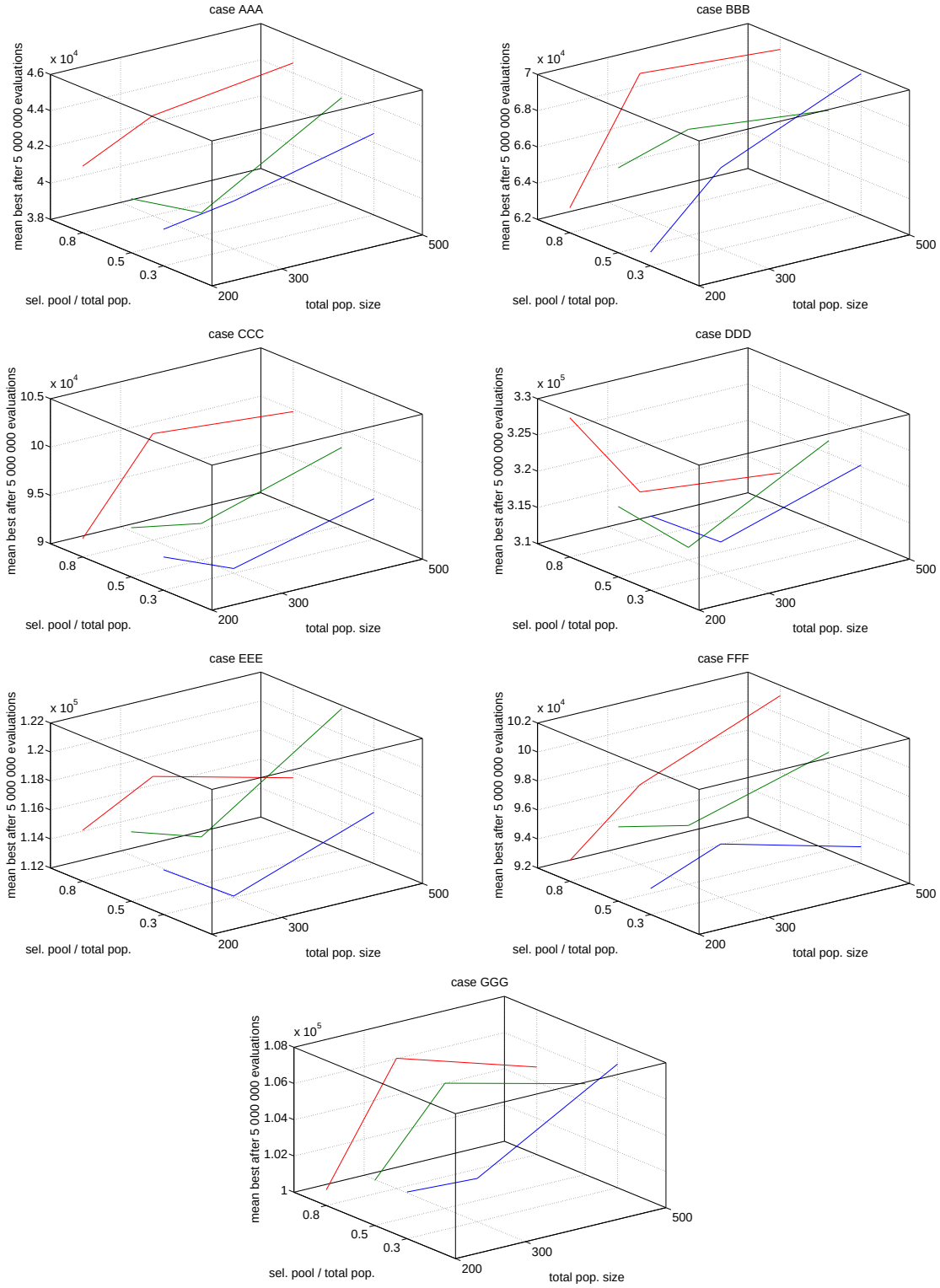


Figure 4.4:

coding type II

Following the same procedure as for type I, we performed $9 \times 14 = 126$ experiments using coding type II. Each experiment is run with seven different random seeds, and we study the mean value of their resulting cost. We show in Fig. 4.5 and Fig. 4.6 the graph of comparison for each experiment individually; then in Table 4.6, Table 4.7 and Table 4.8 the comparison of the fourteen problems altogether. We summarise here some of the observations:

- population size equals to 500 yields the worst result in $30/(3 \times 14) = 71\%$ of the cases (Table 4.7). A causal factor is the fixed number of evaluations, the number of generations decreases as the size of the population increases.
- Studying the mean F.A. of the fourteen problems in Table 4.8, together with the quality of the results, we can conclude that bigger populations have higher potential in solution improvement, but their solution quality improves slower than small populations.
- the F.A. for type II as shown in Table 4.8 are significantly better than the same experiment performed for type I in Table 4.5.
- population size of 200 is the best configuration in $27/(3 \times 14) = 64\%$ of the cases, if we only consider the population size (border table in Table 4.6). This fact alone may lead us to believe that 200 is the most suitable configuration, however, small population size is shown to stall easier than big populations[3]. So we chose the population size to be 300, because it is the second best after 200 and the other alternative 500, is showing too many worst configuration (Table 4.7).
- we observed that the most consistent mating pool size for a population of 300 individuals is to use 50% of the total population (Table 4.6).

As a result, we fix the configuration to population size equals to 300, mating pool size equals to 150, for this series of composite problems (AA, ..., GGG).

[3] A known theoretical result and confirmed in some of our undocumented experiments with larger populations.

Type II: Best configuration counting

		population size 200	300	500	for fixed P.S.
mating pool size	30%	6	0	0	18
	50%	1	2	0	13
	80%	4	1	0	11
for fixed M.P.S.		27	12	3	

Table 4.6: Each of the fourteen test cases (AA,..,GGG) yields nine combinations of parameters ($14 \times 9 = 126$ experiments in total). In this comparison, we account the best configuration among the mean best cost of 7 different runs for each experiment. The central table shows the frequency a certain combination being the best among the nine possible combinations, for a same test case; the table on the right border compares mating pool size for fixed population size, therefore, each entry is the best configuration among three possible options; the table on the bottom compares the population size for fixed mutation rate.

Type II: Worst configuration counting

		population size 200	300	500	for fixed P.S.
mating pool size	30%	0	2	2	14
	50%	0	0	4	14
	80%	0	0	6	14
for fixed M.P.S.		2	10	30	

Table 4.7: The same as Fig. 4.6, but showing the worst case.

Type II: First appearance of the last solution

Pop. size	mean F.A. in # of generations	total # of generations	F.A. in percentage
200	21661	25000	86.7%
300	14722	16666	88.3%
500	8939	10000	89.4%

Table 4.8: F.A.=first appearance of the best solution.

Type II: nn compositions

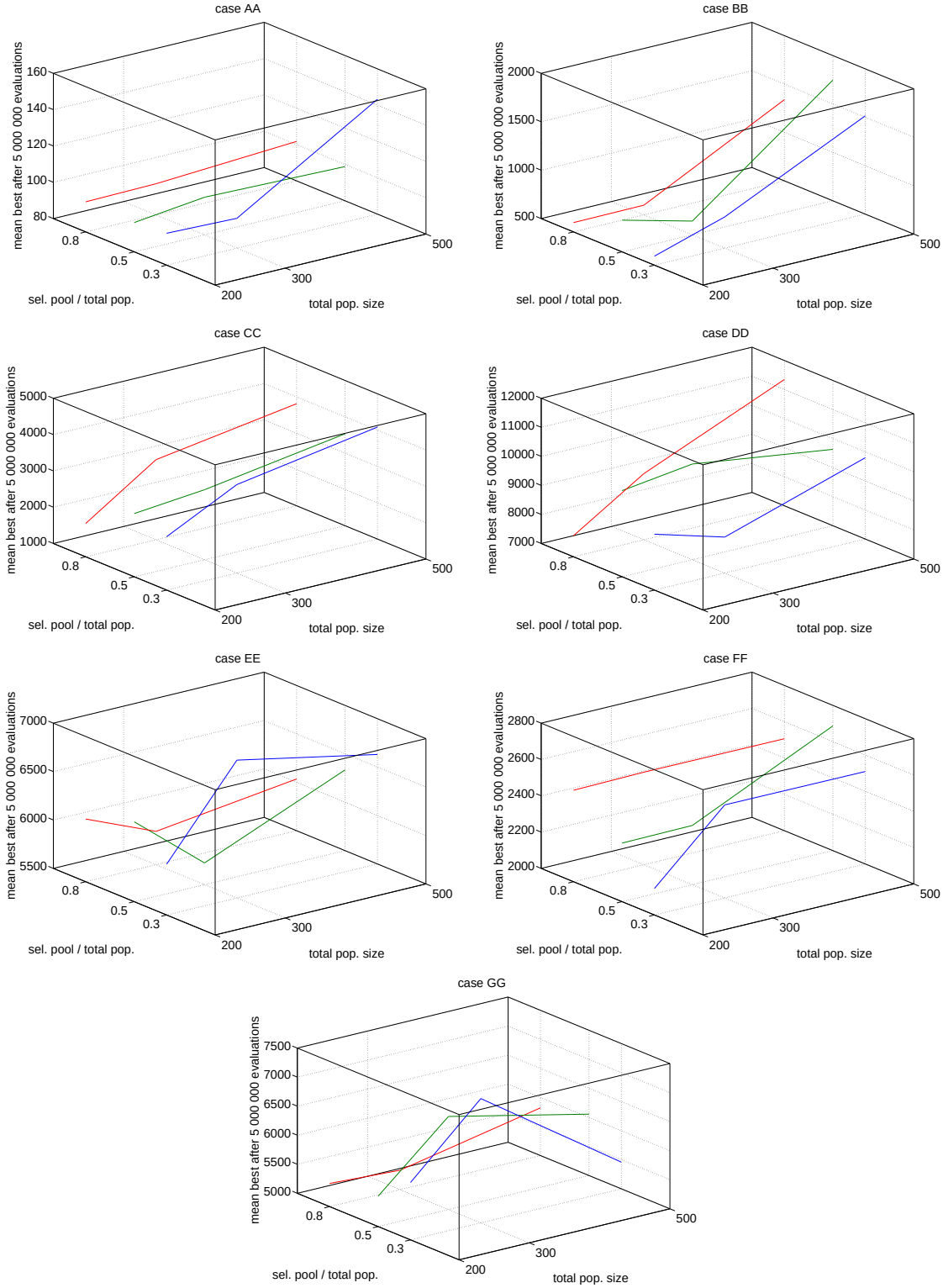


Figure 4.5:

Type II: nnn compositions

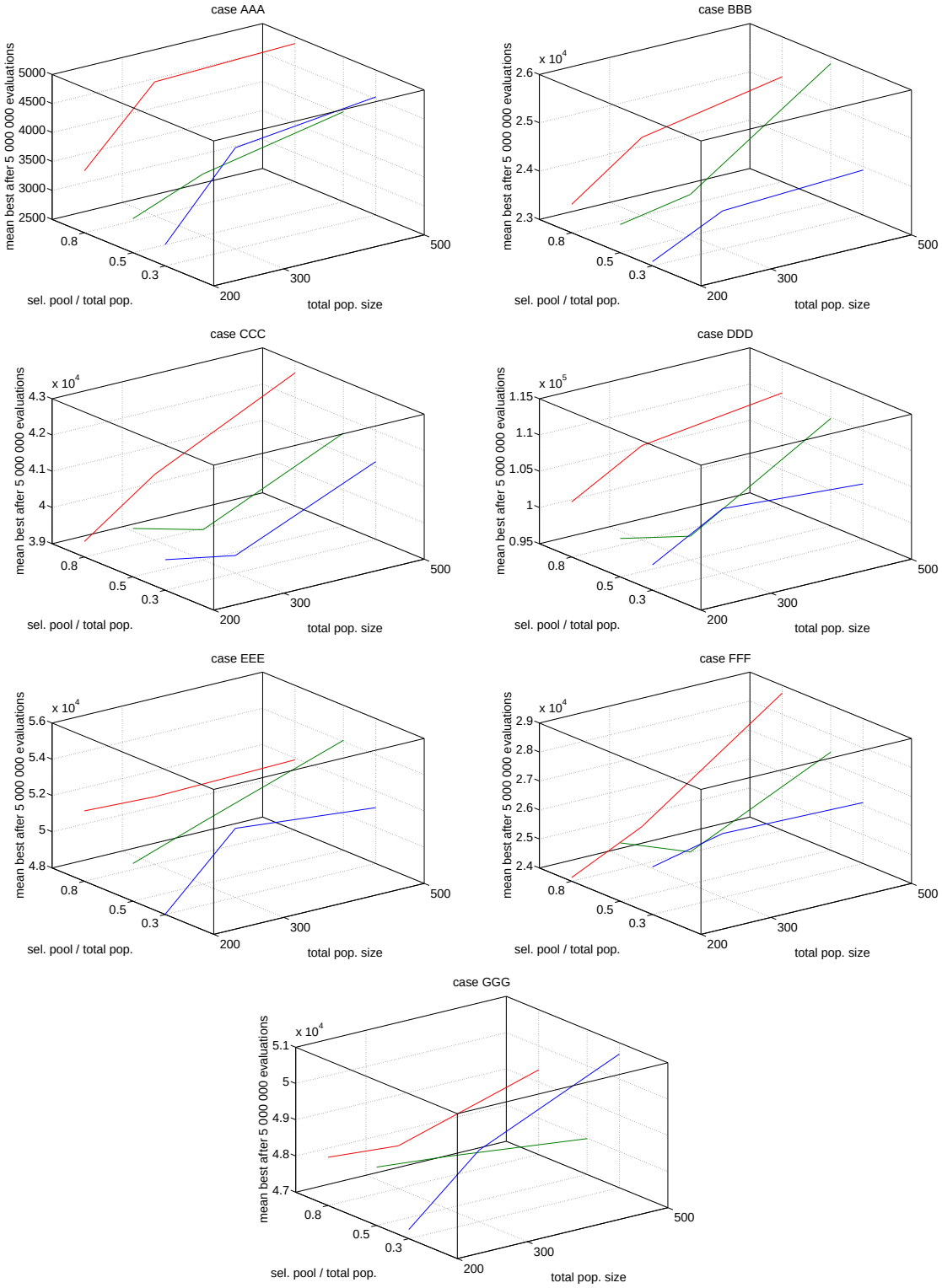


Figure 4.6:

4.3.2 Comparing type I vs type II

Using the same set of collected data, we can perform a first series of comparison between the codings of type I and type II. In this comparison, the number of evaluations was fixed to 5 million and we chose for each problem the best result among the nine combinations of parameters (see Table 4.9 for there values). With best configuration, we refer to the entries that scored in the central table in Table 4.3 and Table 4.7. We can observe a dominance of type II over type I in 100% of the cases. The measure A.Q. above R.S. can tell that coding of type II outperformed type I in more than 50% of quality in almost all the cases. We can also observe that for some cases, type I resulted in less than 10% of A.Q. above R.S., it means that it was only barely better than random search. These facts leaded us to discard type I and concentrate the study on type II.

Type II vs Type I

Case	mean best	
	Type I	Type II
AA	6633.86	96.14
FF	14135.14	2145.86
BB	10111.14	592.86
EE	16631.57	5724.57
GG	14370.71	5333.29
CC	14693.14	1918.71
DD	58996.00	7727.14
AAA	39259.00	2869.71
FFF	93156.00	24129.57
BBB	62760.86	23089.57
EEE	112096.29	48011.00
CCC	90483.71	39427.71
GGG	100862.00	47253.29
DDD	311723.43	98272.14

Case	A.Q. above R.S.	
	Type I	Type II
AA	08.5877%	99.8584%
FF	13.2161%	87.4026%
BB	20.6762%	96.1702%
EE	23.6723%	74.1163%
GG	28.3112%	73.6661%
CC	20.9884%	90.3430%
DD	14.5385%	89.1657%
AAA	10.1203%	93.7047%
FFF	13.1550%	77.6193%
BBB	04.8521%	65.1538%
EEE	06.0417%	59.8497%
CCC	09.7479%	60.7916%
GGG	10.2337%	58.0061%
DDD	10.5680%	71.8899%

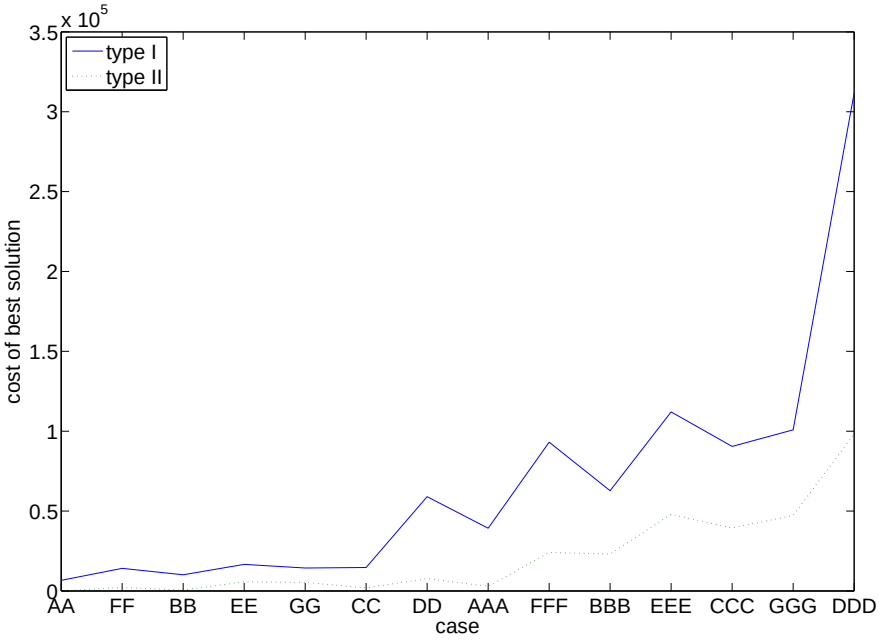


Table 4.9: Number of evaluations fixed at 5 million, the table (upper part) and the graph above shown compare the mean best result from codings type I and type II. Lower table shows the A.Q. above R.S. for both codings.

4.3.3 About random search

The following are results obtained from a random search[4] of 5,000,000 to 180,000,000 evaluations (of the objective function), we report the mean value of the best cost from seven different runs using seven different seeds in the pseudo-random generator. As we are using the best solution obtained from random search as a component in our A.Q.'s formula, we show some of their values as reference for the reader in Table 4.10, Table 4.11 and Table 4.12.

For our test suite, random search didn't yield good result, this fact can be observed in Fig. 4.7. There we can see that the result from a random search of up to 180,000,000 evaluations is still some order of magnitudes away from the optima cost. Although the probability of reaching the optima is not zero, the improvement rate is just too low. Remember that the experiments have a huge solution space size, and the sampling is done for an insignificantly tiny portion of it.

[4] An uniform sampling with replacement.

Random search - 5,000,000 evaluations

	best result cost			worst result cost	
name	best	mean	σ	mean	σ
AA	7249	8854.57	920.04	43036.57	285.96
FF	16271	19386.57	1580.87	77558.14	1484.51
BB	12718	13691.00	546.66	52791.57	763.13
EE	21750	22651.14	753.29	84853.14	1262.03
GG	20012	20570.86	492.67	77197.71	1113.84
CC	18559	19661.00	613.46	63174.29	992.58
DD	68984	72918.29	1892.79	225678.57	4706.59
AAA	43665	45873.86	2028.20	114268.86	347.24
FFF	107242	108336.14	737.25	224801.43	2909.38
BBB	65953	73300.43	3179.27	154785.14	665.98
EEE	119292	126274.43	3664.07	256931.71	3759.94
CCC	100234	103162.71	1666.28	196899.14	2483.64
GGG	112346	115497.86	1815.31	229969.86	2251.71
DDD	348509	352399.71	3664.20	647006.14	6225.28

Table 4.10: Statistics of the best and worst solution appeared in the seven different runs of random search with 5,000,000 evaluations of the objective function. Results ordered by size of the solution space.

Random search - 30,000,000 evaluations

	best result cost			worst result cost	
name	best	mean	σ	mean	σ
AA	6589	7149.43	412.30	43406.00	291.58
FF	16274	18000.00	1066.51	79714.86	830.64
BB	12718	13045.29	450.69	53756.43	724.13
EE	19217	20901.00	1079.22	86654.29	405.58
GG	19180	19308.00	199.86	80434.00	1281.46
CC	17903	18234.43	247.98	64737.00	147.39
DD	61760	68795.43	3035.44	229097.43	3270.11
AAA	43665	44635.00	772.69	115022.14	364.93
FFF	102841	103559.00	781.51	226248.43	1781.25
BBB	65953	71860.00	2543.76	157756.00	626.59
EEE	119292	123010.71	2687.15	262437.86	2825.59
CCC	100234	100735.43	447.79	199794.14	2027.90
GGG	109165	111523.29	1652.32	234307.14	1622.24
DDD	341367	344171.14	2318.81	648819.71	4973.85

Table 4.11: Statistics of the best and worst solution appeared in a random search of 30,000,000 evaluations of the objective function. Results ordered by size of the solution space.

Random search - 180,000,000 evaluations

	best result cost			worst result cost	
name	best	mean	σ	mean	σ
AA	6589	6963.29	275.41	44001.71	154.40
FF	16208	16463.00	599.29	80554.00	292.31
BB	11122	11798.43	427.81	54489.57	8.58
EE	19209	19212.14	4.97	87259.43	668.35
GG	16690	17044.86	561.08	81160.00	762.11
CC	15998	16084.86	137.33	65738.14	576.89
DD	61760	63802.29	3229.14	235817.71	3.61
AAA	39343	41812.14	1671.86	117089.29	218.02
FFF	99020	98216.29	1990.75	229763.71	1634.04
BBB	65953	69645.00	1970.30	159465.00	859.02
EEE	119292	121483.57	1075.37	265057.71	1934.07
CCC	94942	97356.43	1588.61	202823.00	1738.03
GGG	107906	108993.86	740.05	239361.43	2653.74
DDD	325056	334130.43	6365.35	655514.57	2328.96

Table 4.12: Statistics of the best and worst solution appeared in a random search of 180,000,000 evaluations of the objective function. Results ordered by size of the solution space.

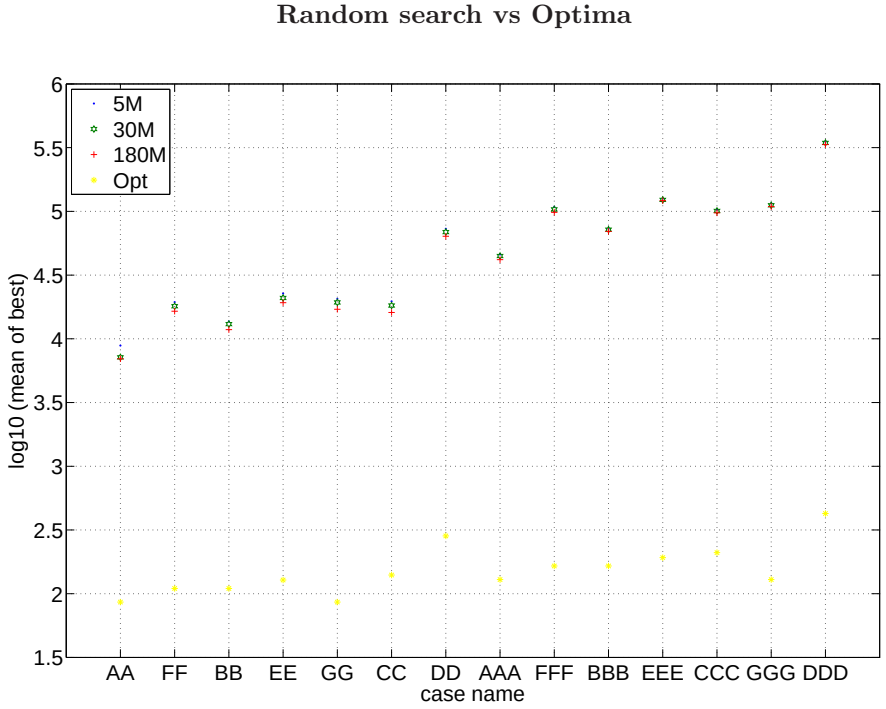


Figure 4.7: Comparison in logarithmic scale showing how R.S. yields results 2 to 3 orders of magnitude away from the known optima in our test cases, and the difference between R.S. of different number of evaluations.

4.3.4 Comparing against random search

Using the mean result of different runs with different random seeds, Fig. 4.8 shows the cost in logarithmic scale of the results from random search and genetic algorithm using coding of type II. We can observe that for the first half of the test suite -smaller instances-, G.A. of 90M evaluations yields results within the same order of magnitude as the known optima, and it performed better in all the cases comparing against random search, even for evaluation of 30M in gaII against random search of 90M. To be better than random search is not an attractive news, to check how far does G.A. reach in the segment between R.S.'s result and the optima cost, we study Fig. 4.9. There we can observe that in $6/14 = 43\%$ of the cases the approximated quality above random search was above 95% for a G.A. with 90M evaluations. And in the worst case, it was only above 65% of A.Q. above R.S. We can also see that the efficiency was lower for bigger problem instances.

These results are not good enough, as we can see that bigger instances in the test suite are two to three orders of magnitudes away from the exact solution, and the number of generation required is very big.

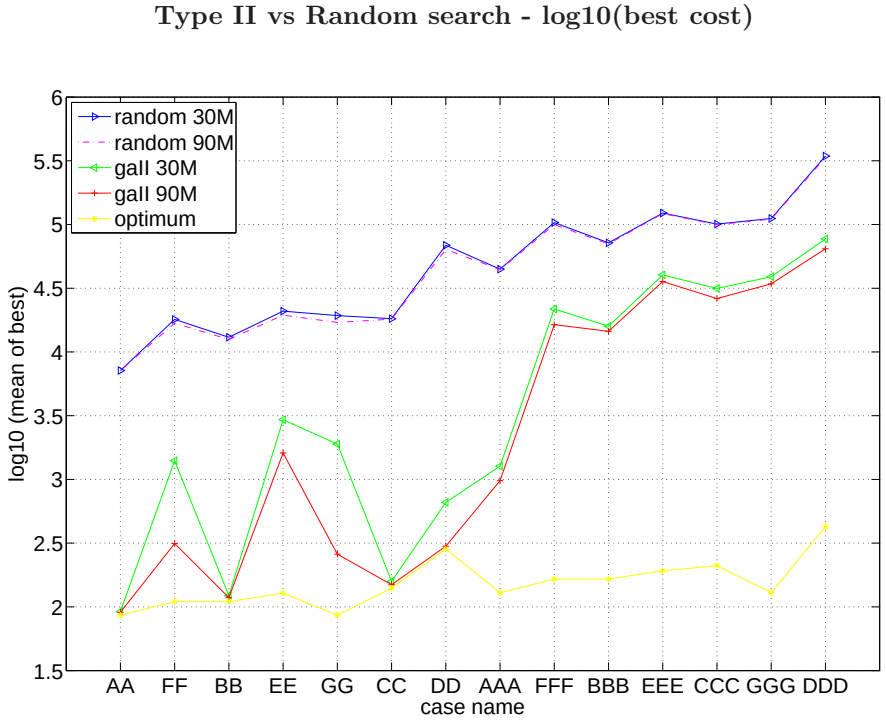


Figure 4.8: Comparison in logarithmic scale showing the improvement rate in random search of 30M and 90M evaluations against genetic algorithms of 30M and 90M evaluations (type II coding).

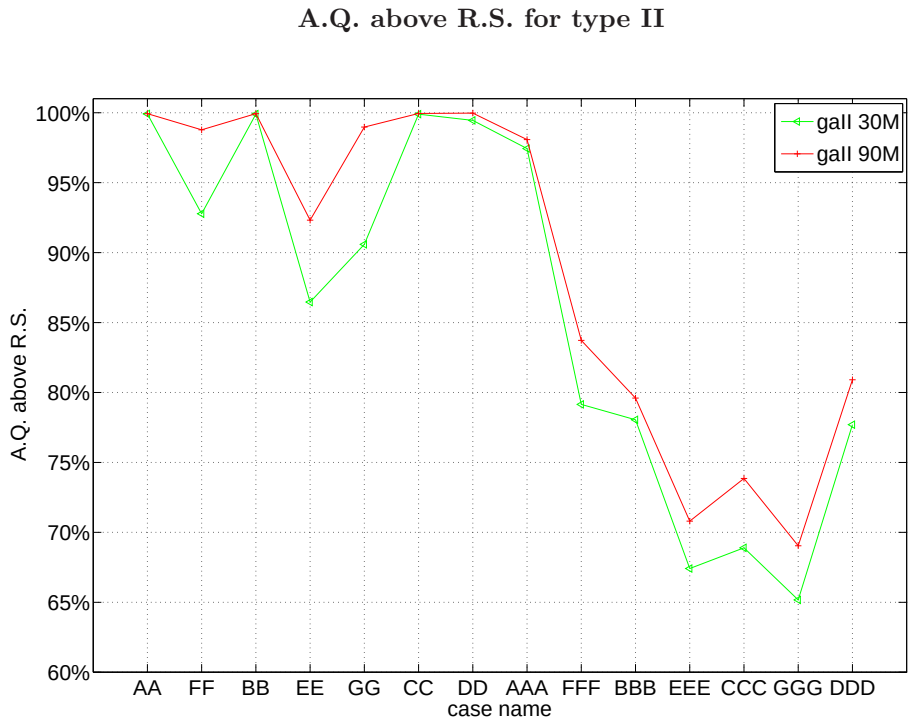


Figure 4.9: A.Q. above R.S. showing the quality of solution compared to random search of same number of evaluation. 0% is assigned to solution from random search, while 100% is assigned to optimal cost.

RTD

Among the literature about evaluation of stochastic search methods, we found the study of runtime distribution (RTD) as in [Hoos 04], it is a three dimensional graph in which the runtime, solution quality and probability are shown in the three axis. To construct a RTD graph, a large number of test cases should be considered, for which the evolution of the solution quality over runtime is recorded. Then for each pair of (s=solution quality,r=runtime), the probability is calculated by checking the number of test cases that got a solution quality higher than s while the runtime is equals to r.

We found RTD quite useful as it gives us some insight into the evolution of the solution quality as increased among of time is invested in the solving. The 3D graph of RTD is a very compact way to express many different aspect of the heuristic. In this part of the experiment, we used the same 14 composite problems and 7 random seeds, and trace the evolution of the best individual in different generations.

Notice that the RTD in use is not exactly the one presented in [Hoos 04], but a modified version of it. At the first place we used number of objective function evaluation as time measure. Secondly, we replaced the traditional quality with A.Q. above R.S., and in particular we take the mean result (from 7 different seed) of a random search with 180M evaluations as the basis of comparison for all the data in RTD. The latter is more flexible than the dynamic A.Q. above R.S., and it allowed us to change the grid size without recalculating the intermediate results from R.S., as the stored data from R.S. were sparsely selected.

One consequence of these changes is that for small number of evaluations, the result from G.A. will yield a negative A.Q. above R.S., and as we can observe in Fig. 4.13. When the number of evaluation is close to zero (first curve next to the right border), the quality is below zero and lower extreme has higher probability. But in the next curve -3M evaluations-, we get results higher than 40% of A.Q. with 100% probability (more clearly shown in Fig. 4.14), it means that the proposed G.A. outperforms a R.S. of 180M evaluations, with less than 3M (to be more detailed, we found out that our G.A. with 210,000 evaluations already outperforms R.S. of 180M, for this test suite).

4.3.5 Enters pre-processing

Trying to improve the results, we continue our experiment by adding pre-processing. In particular, we found one that works well, instead of generating the initial population randomly, we can try to optimise demands individually and then include the combination of those solutions somehow in the first generation. Fig. 4.10 shows the comparison in logarithmic scale of genetic algorithms with and without pre-processing, it is clear that both the number of generations and the quality of the solution improved with pre-processing in all the test cases. In this series, the demands are treated individually in the pre-processing, if the solution space is smaller than one million, then exhaustive search is applied; otherwise, a G.A. of 600,000 evaluations is applied. Then we choose one solution in each demand to compose a special solution, which is included into the initial population (all others are generated randomly). This pre-processing is an effective way to accelerate the search. We can observe that the A.Q. above R.S. greatly improved even when using a smaller number of generations. For 30 million evaluations, all cases yielded above 90% of A.Q., see Fig. 4.11. And for the first half of the test suite -smaller instances-, the A.Q. reaches 99% in a consistent way.
One explanation for the lower A.Q. observed in bigger instances is because of the fixed number of evaluations assigned to the pre-processing initialisation. For bigger instances, even when the problem has a unique demand, the solution space is much larger than smaller instances. Therefore, the limited and small number of evaluations yields a poor initial population in comparison when it is applied to smaller instances. Remember that the problem with one single demand is already $\mathcal{NP}$-hard.
We observe from Fig. 4.11 that the improvement obtained by increasing the number of evaluations is quite small for bigger instances. But on the other hand, the quality obtained by using only 30M evaluations is already much better than G.A. without pre-processing.

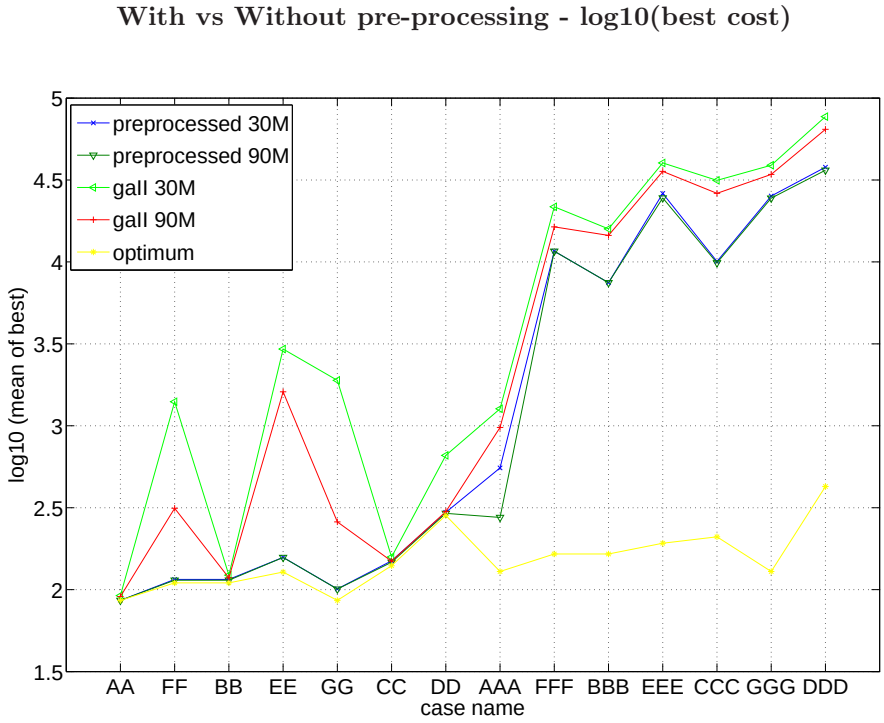


Figure 4.10: Comparison in logarithmic scale showing the improvement rate of genetic algorithms with and without pre-processing of 30M and 90M evaluations (type II coding).

We also compared coding of type I and type II with pre-processing included (see Fig. 4.12), the latter outperformed the former in most of the cases. Type I only performed better in 2/14=14%, and in those cases the difference in solution quality are very small, on the other hand, type II yielded solutions up to one or two orders of magnitude better than type I (take case CC in Fig. 4.12 as an example).

With the usage of runtime distribution graph, we can compare more easily both solution methods, we show them in the same scale Fig. 4.13, Fig. 4.14 and Fig. 4.15.

By studying Fig. 4.14, we can observe that there is a clear dominance of pre-processing with 3M over the simple one with 120M evaluations (yellow line in the upper figure vs blue line in the lower one), and even the 0-th generation from the former one gives with almost 30% probability a solution with A.Q. above R.S. of 99.9% (black line in lower figure). And for the case with pre-processing, the increase in A.Q. above R.S. become poor as the number of evaluations increases. But the fact that it manages to outperform the simple version makes it a better alternative, especially when it requires a much lower number of evaluations to do it.

Then, in Fig. 4.15 we can study the portion of the experiment and the achieved A.Q. above R.S. as the number of evaluations increases. The most important information here is the evolution of the portion of high quality results. For instance, we can compare the curve of 99.9% A.Q. above R.S. (purple line in both graph), for the simple G.A. it evolves from 0% up to below 30% in 120M evaluations. On the other hand, with pre-processing it begins just below 30% and almost reaches 99.9% in 90M evaluations. The lowermost (yellow) curves in both graphs from Fig. 4.15 correspond to the portion of cases reaching exact solution, we can see that with pre-processing we got a higher portion than without it.

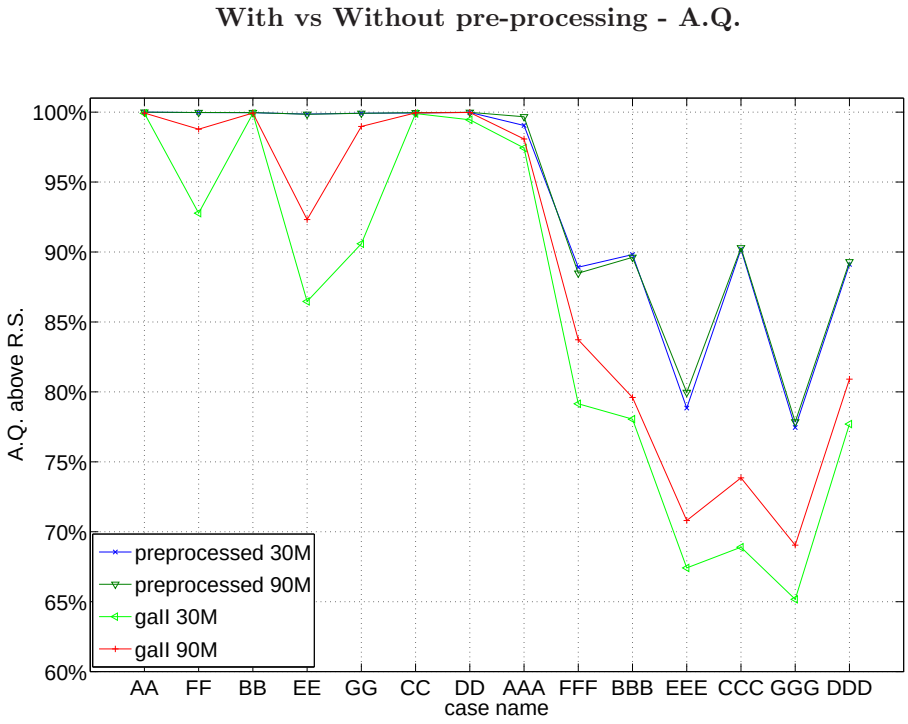


Figure 4.11: A.Q. above R.S. showing the improvement in quality of genetic algorithms with and without pre-processing of 30M and 90M evaluations (type II coding).

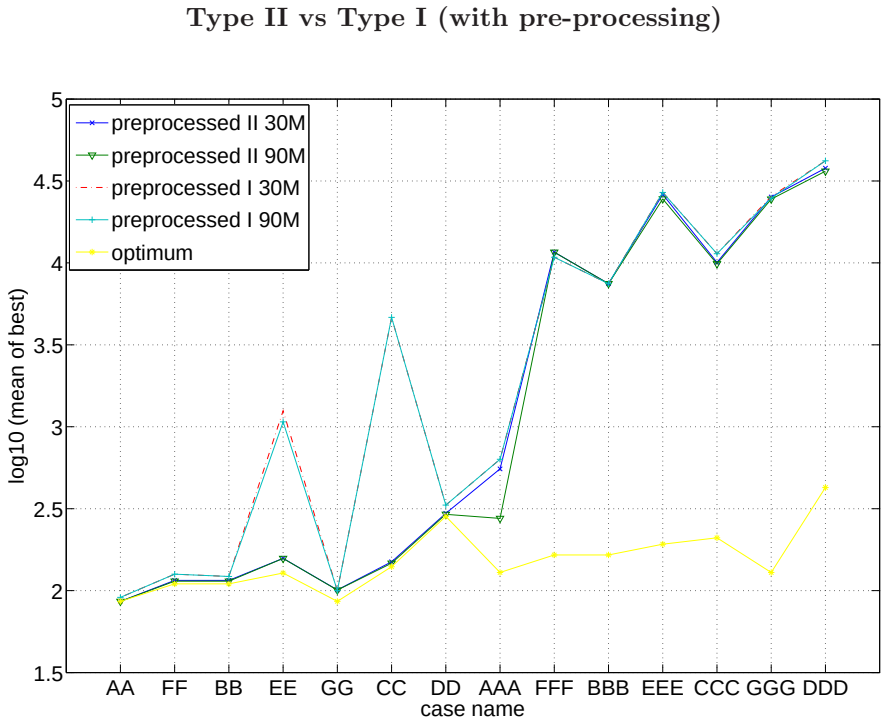


Figure 4.12: Comparison in logarithmic scale for both types of coding using pre-processing. The test cases are arranged in increasing order of solution space size.

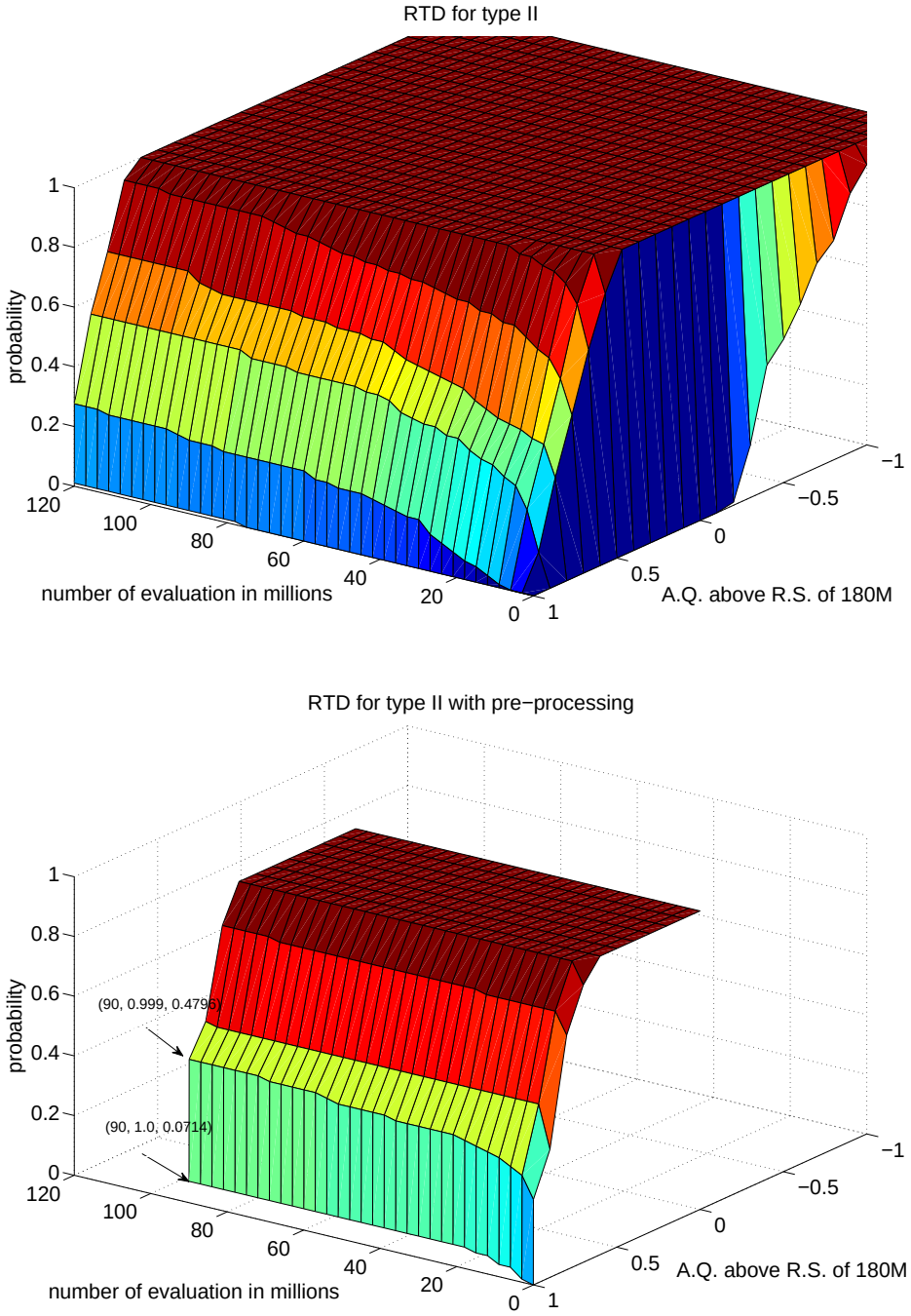


Figure 4.13: Without pre-processing above, and with it below. A.Q. above R.S. is taking R.S. of 180M evaluations as basis of comparison. We included the coordinates of two points in the lower graph in an attempt to aide its interpretation as there is an ambiguous visual effect.

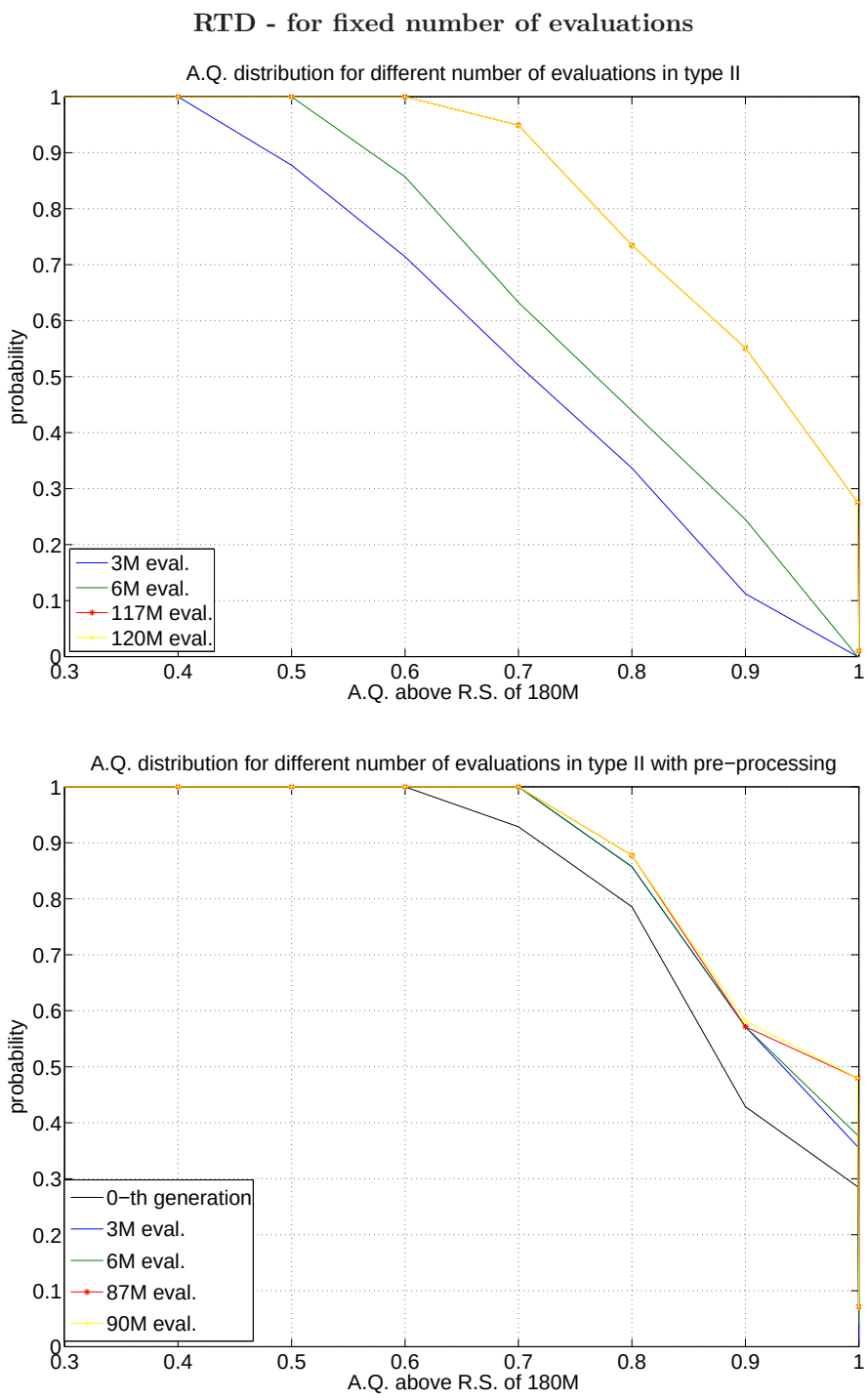


Figure 4.14: Without pre-processing above, and with it below.

RTD - for fixed A.Q. above R.S.

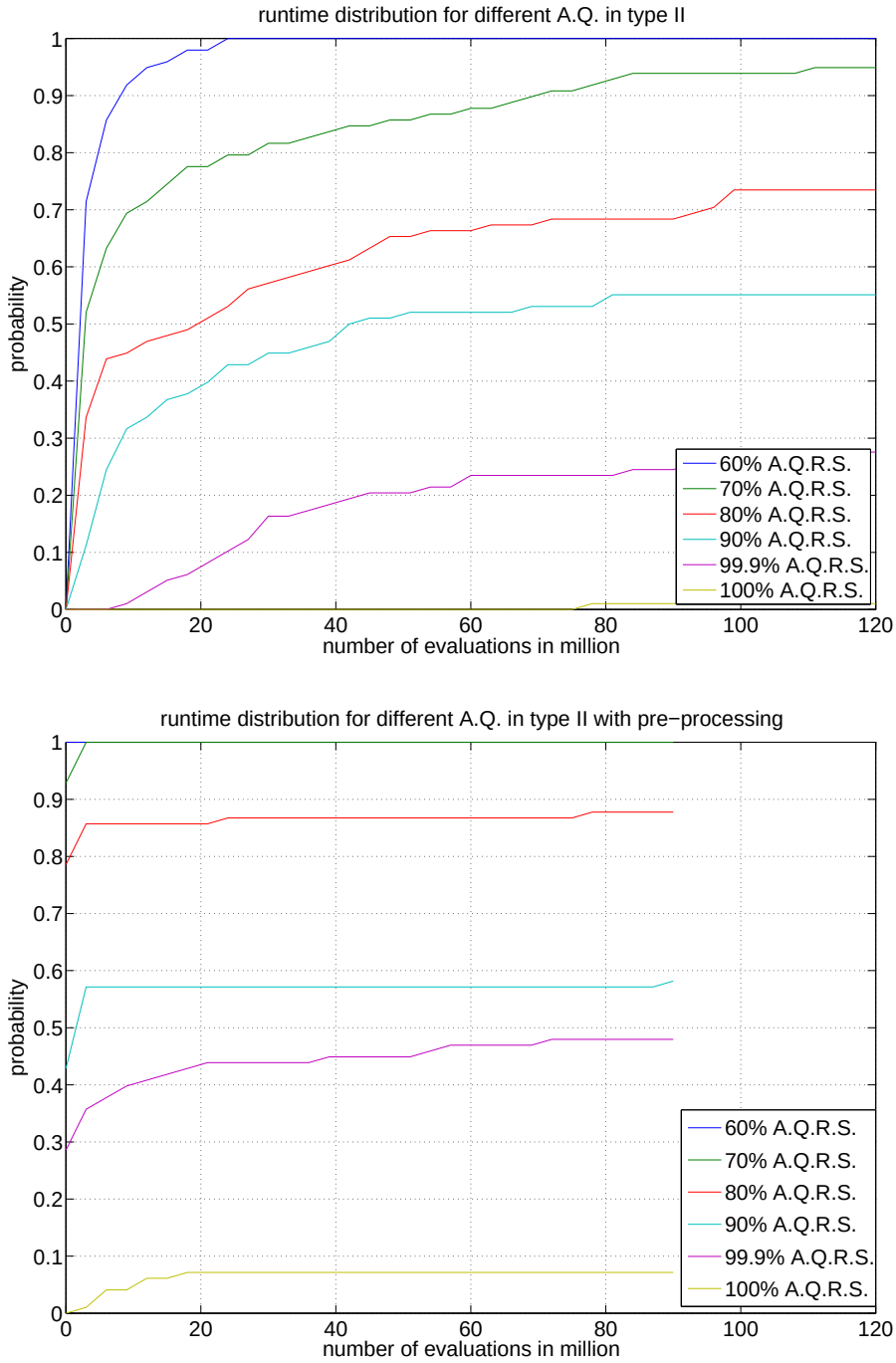


Figure 4.15: Without pre-processing above, and with it below.

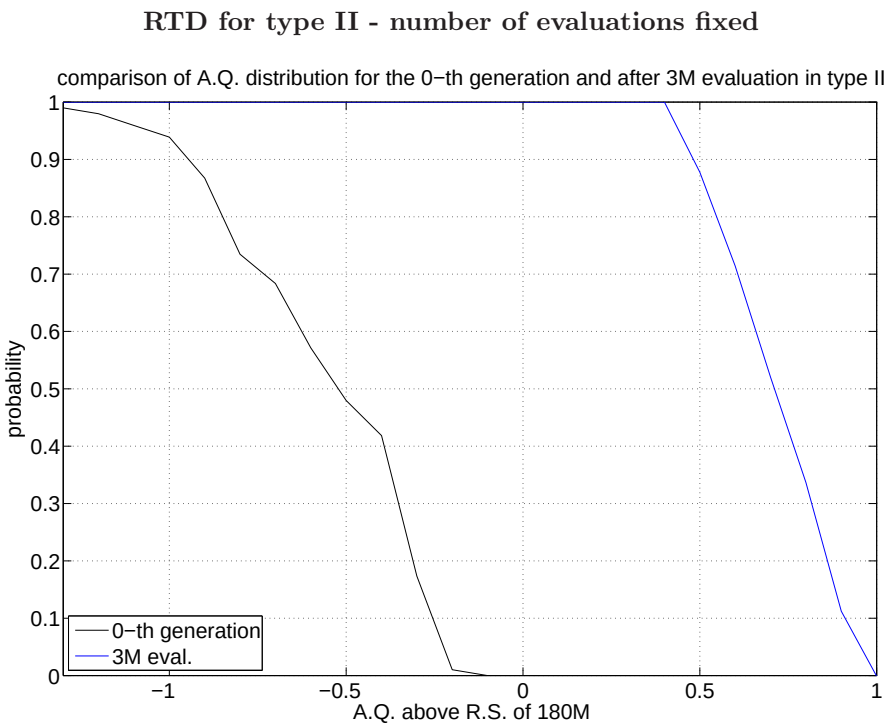


Figure 4.16: Comparison of A.Q. above R.S. for the 0-th generation and after 3M evaluations.

Chapter 5

Conclusion

5.1 When does a problem become hard?

What makes an optimisation problem hard to solve? Let's consider the following factors that are known to be hard for a G.A. and solvers in general:

- huge solution space size together with tiny portion of optimal solutions
- many local optima,
- deep local optima.

Notice that one condition alone is not enough to make a problem hard, for example, a convex (continues) function with a huge solution space which has one single deep local optima is not hard to optimise with G.A. or most other local search heuristics. But all the three conditions above altogether do make the problem complex, and G.A. require a large number of iterations (generations) to overcome it, which directly translates into long run-time.

5.1.1 About the solution space

The solution space in the problem under study in this work is the Cartesian product of sets, where each set is the solution space of one demand. Moreover, for a single demand, the size of its solution space increases with its hop-limit and the number of nodes in the graph $|V|$.

The size of the solution space presents a hill's shape when the diversity factor K (number of node-disjoint paths) varies (check Fig. A.10); there exists a region $K \in [1, \frac{|V|}{R}]$, in which the volume under the hill's surface is many orders of magnitudes away from the one outside. Moreover, R takes values from 2 to 3 for $|V|$ ranging from 10 to 40, and R increases slowly as $|V|$ increases. The latter is caused by the increasing difference in orders of magnitudes between the peak of the hill and values outside the region, as the number of nodes $|V|$ increases. On the other hand, as the hop limit H increases, the size of the solution space also increases, and with monotonicity.

We are working with the assumption that there exists a high cost capacity type for each link as explained in the previous chapter, and under this assumption, the solution space is invariant to the other configuration of capacity types. And for complete graphs, the solution space is a function that depends only on the number of nodes, the number of demands d_i, and their associated diversity factor K_{d_i} and hop limit H_{d_i}. In real life problems, the most likely scenery is to have many demand pairs, but for each pair there will be a low diversity factor (K) ranging from two to four; with a hop limit much lower than $|V|/2$; and cost of the different link types depending on the map distance of the involved nodes (therefore complete graph is meaningful). In other words, those problem instances are among the smallest that one can build with the same number of nodes and demands.

5.1.2 About local optima

Reviewing the results in the last chapter, we can see that G.A. with pre-processing tends to arrive to a good local optima quickly, but after some improvements in the first series of generations, it

stalls and becomes hard to improve (staying in the same result for a large number of generations). This behaviour is caused by local optima, which in turn are related to the coding we designed.

The day when someone finds out how to codify (encode and decode) efficiently solutions of this problem, in such a way that the cost function is monotone, then this optimisation problem will be solved by just decoding the most economic solution. Or even when there is only one single local optima (convex minimisation or concave maximisation), then we can apply local search to solve it. Unfortunately, the coding we propose here hasn't got those nice properties, and it has many local sub-optima.

By studying the local optima in which the G.A. takes a lot of generations to escape from or in those it fails to escape. We observed that deep local optima appear when a bad link (read sub-optima link type with big capacity step) is shared by many demands. Different strategies were tried in the fitness function, among them we find:

- replacing step cost/capacity function with piecewise linear function;
- replacing step cost/capacity function with piecewise concave function;
- replacing step cost/capacity function with piecewise convex function.

For concave functions, the variation of cost is high for low capacity saturation, and low for high saturation. In this way concave functions penalise the usage of low saturation over high one, as opposite to convex functions, see Fig. 5.1. They offered no clear improvement over the usage of step cost function, one of the reasons is that any optimal solution can have high and/or low capacity usage, penalising them may mislead the G.A. For the case of piecewise linear substitute, it promotes the usage of many different arcs which translates into expensive solutions in the original step cost, to be effective, a more intelligent discretisation mechanism is required.

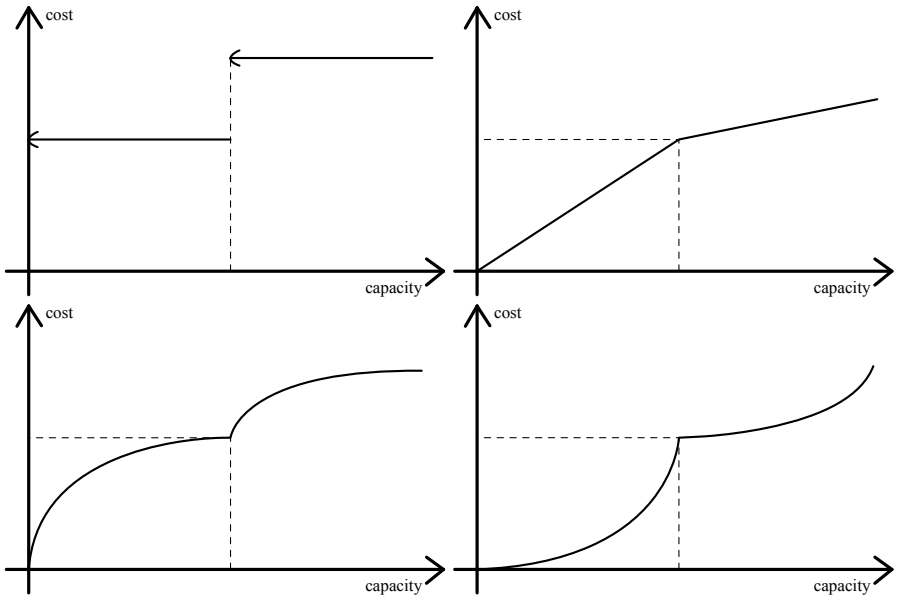


Figure 5.1: Step, piercewise linear, concave and convex function, check the variation of cost for different capacity usage.

5.2 Summary of the thesis

In Chapter 2 we presented a list of interesting problem in the area of network design, and in Section 2.4.3 we defined our network design problem which contains some of the most interesting design objectives, but then its complexity is very high. To the best of our knowledge, this problem as it is presented here, has not been proposed previously, but there are a number of very similar problems treated by other authors (like in [Botton 07]) which confirms the interest of this problem in

the community. Begin a problem too hard to solve with an exact approach (with nowadays knowledge), we adopt a heuristic approach, and the chosen heuristic is genetic algorithms. The choice of genetic algorithms is motivated by literature where we found good results for other hard combinatorial problems. The latter also motivated the development of our G.A. library, namely GGEngine.

During the development of a coding for the G.A. in Chapter 3, we discovered that to encode and therefore enumerate the solution space is intractable for generic graphs. [Bley 97, Itai 82] proved that SFP is $\mathcal{N}P$-complete, and as the existence of an efficient K-path enumeration for generic graphs may be used to solve SFP directly, we arrive to the conclusion that such enumeration can not be done efficiently (in polynomial time) unless $P = \mathcal{N}P$. Also from [Allender 99, Valiant 79] we learned that the number of simple path (s-t path with s,t fixed) in a directed graph is $\#P$-complete, in other words, it is hard even to count the number of paths when we have only one single demand without hop-limit and $K = 1$. All these theoretical limitations force us to limit the coding to complete graph, which is an acceptable simplification for network design.

By studying the combinatorial structure of K-paths and H-limited K-paths, we managed to develop a coding (Appendix A) which maps H-limited K-paths to natural numbers, in a bijection. The aforementioned coding has the interesting properties of being always feasible (with respect to the hop-limit and path diversity constraints), and represents the whole solution space without repetitions. In this study we focus on the TC-DHK problem, but we remark that the developed coding can also be applied in other problems with hop-limited K-paths as decision variables, one example is TC-DHK with a linear cost per edge capacity usage. One of the by-products of the combinatorial analysis is a number of tree structures with interesting properties. Later on, we found out that permutation tree and binomial tree (Appendix A.3.1) are already known in the literature. But we couldn't find any reference to the structure we defined as the bounded binomial tree (Appendix A.4.1), which leads us to believe strongly that it is an original contribution.

Before being able to compare the different solutions, we evaluated the different traditional measures, and ended up defining "A.Q. above R.S.", which is more consistent and robust than the other traditional alternatives. We also found the RTD graph to be a compact and useful tool in the comparison of stochastic search algorithms.

We tested the proposed solution in Chapter 4, by first determining the most suitable values for the different parameters to use. Some other authors consider the large number of parameters in G.A. a disadvantage, and in this work we managed to adjust some of them dynamically, reducing in some degree the number of parameters. We then proceeded to compare the two proposed codings, type I and II, the result is that type II always gave better result in all the experiments. But the results so far were not acceptable as the algorithm required a very large number of evaluations. As an attempt to improve it, we experimented with the addition of a pre-processing step, and we managed to develop one that achieves better results in a much lower number of evaluations. The latter is the final solution method that we propose (G.A. type II with pre-processing), it outperforms the other alternatives in quality and to requires a relatively small number of evaluations.

The usage of pre-processing has the effect of advancing the evolution in the first step, saving a large number of generations; it is also a well known technique to accelerate the evolution. But later on the improvement in each generation remains very low. This shows a limitation in G.A., to improve further the solution quality, some mechanism to increase the diversification or even other optimisation techniques should be considered.

Although this work has not ended with a best-ever solution method for the proposed problem, we started here a first step that provides a basis of comparison so further research can be done for this network design problem. At the same time, good upper bounds are useful in combination with many other optimisation techniques, and the proposed G.A. with pre-processing and fitness caching can give such bound. Being a generic version of other complex problems, any advance here has a direct contribution to the other special cases. Moreover, further improvements can be done in the proposed solution.

Appendix A

Enumerating H-limited K-paths

A.1 Introduction

"Enumerate" is the verb to name things on a list one by one, mapping each element with an identification, and it can be done easily when there is an order defined over those elements. At first sight, there isn't any natural or intuitive ordering for the case of node disjoint paths on a complete graph, and here we present an algorithm to map this kind of elements to identifications (together with its reverse mapping). Such a mapping is useful under many different situations, like: the application of genetic algorithms when the decision variable of a certain problem is a set of node disjoint paths[1] ; or the generation of solutions for a brute force approach[2]; or just when the exact size of the solution space is to be determined. Some algorithms solve this kind of mapping by generating systematically solutions, in our approach, the mapping is done directly without need for solution generation. Additionally, we present an extension of the result for node disjoint paths with hop limit.

A.1.1 Some general definitions

By $G(V, E)$ we denote an undirected graph which is a pair of sets V and E, where V represents a non empty set of vertices and E represents the set of edges that are drawn between pairs of vertices. In this work will use vertices and nodes as synonymous, the same for edges and links. The cardinal number of a set S is referred to by $|S|$.

$G(V, E)$
set V
set E
| |

A path is a sequence of nodes, without repetitions (no cycles or loops). An origin to destination or source-target (s-t) path is one that begins with a given source node s and ends with a given destination or target node t. Paths that don't share any intermediate nodes are called node-disjoint paths. Paths that don't share any edges are called edge-disjoint paths (in an undirected graph). In this work an s-t K-path, or K-path in short corresponds to a set composed of K node-disjoint $s-t$ paths. Moreover, a hop limited K-path is denoted as H-limited K-path, where H is a natural number denoting the maximum number of hops. We represent a path as a list of node identifications, and denote the concatenation operations for list with : (the colon mark[3]).

s-t path
K-path
s-t K-path
H-limited K-path

By encoding we refer to a transformation $\Theta : \mathcal{A} \rightarrow \mathcal{B}$ which transforms each element of the set $\mathcal{A}$ into a different element of the set $\mathcal{B}$. Here, the set $\mathcal{A}$ is the set of K-paths for a given K and given $s - t$ node pair, and $\mathcal{B}$ is a subset of natural numbers $\mathbb{N}$. The reverse operation is called decoding $\Theta^{-1} : \mathcal{B} \rightarrow \mathcal{A}$, for each element in $\mathcal{B}$ it returns the corresponding element in $\mathcal{A}$ such that $\Theta^{-1}(\Theta(a)) = a$, $\forall a \in \mathcal{A}$. In the general case, it is possible to have $|\mathcal{B}| \geq |\mathcal{A}|$, but for bijection it is $|\mathcal{A}| = |\mathcal{B}|$.

encoding
decoding

[1] In this case, an encoding/decoding function is required to map a solution with a gene, and it's easier to map natural numbers to genes than mapping paths to genes.

[2] This is viable only for tiny solution spaces

[3] The colon mark is used as concatenation operator for list operation, and range operator in Matlab codes.

A.1.2 Conventions used in the article

In the present article, we are working always with **complete graphs** $G(V, E)$, and we consider two natural numbers $H, K < |V|$. It has been proved that there is no efficient way to find a H-limited K-path in a generic simple graph, unless the complexity class P equals to $\mathcal{N}P$ [Bley 97, Itai 82]. Therefore, we consider the limitation of working with complete graphs acceptable.
We exclude loops and the direct path s to t (of one single hop) in the enumeration of K-path and H-limited K-path. Later on, in Section A.4.4 we will lift the no direct s to t path restriction, but during the rest of the article, this restriction will help to simplify the explanations.
We consider all the nodes in V labelled and arranged according to a total order, most of the times we are interested in the set $V' = V \backslash \{s, t\}$ and without loss of generality we can label the nodes in V' as $\{n_1, n_2, .., n_{|V|-2}\}$, in this way, the i-th node is referred to the index i of such arrangement.
When naming the levels of a tree, we call level zero to the root level of a tree, so level one is the following one.
Observe that to have a maximum of H hops in a path is to have a maximum of $B = (H - 1)$ B intermediate nodes (without counting origin and destination nodes).

A.2 Enumerating single paths $(K = 1)$

We are interested in mapping paths to natural numbers (and its reverse mapping), let's start with the simple case where $K = 1$, this section deals with single s-t paths. Without s and t, there will be $|V'| = (|V| - 2)$ vertices. In a complete graph, there are many $s - t$ paths, but there are only:

- $|V'|$ paths with exactly one intermediate node, those paths have two hops,
- $|V'|(|V'| - 1)$ paths with exactly two intermediate nodes, those have three hops,
- and exactly $\frac{|V'|!}{(|V'|-i)!}$ paths for the case of i intermediate nodes, those have $(i + 1)$ hops.

We can model this situation with a rooted tree of $|V'|$ levels, its nodes are labelled after those $|V'|$ vertices. Each node in the i-th level represents a path with i intermediate vertices, leaving $(|V'| - i)$ unvisited vertices, which are the possible labels for its children. To define its structure, let's arrange all the children under a common parent in an ascending order form left to right. Fig. A.1 shows an example of a tree with $(|V'| = 3)$ three labels.

The idea is to assign a unique identification for each of those $s - t$ paths, and an easy way to do it, is in a breadth first fashion. The number next to each node in Fig. A.1 is its assigned identifier.

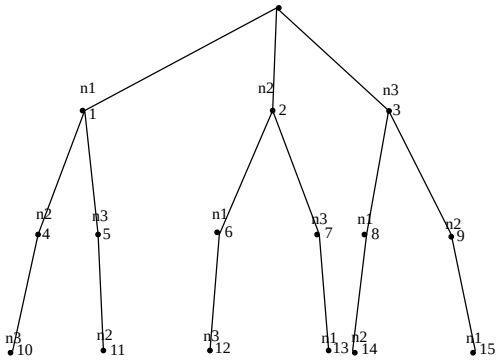


Figure A.1: Breadth first enumeration for a single-path-graph with 3 nodes.

This tree has some interesting properties:

- it has $|V'|$ levels,
- all nodes in the same level branch in the same number of children,

- every node branches into one less child than its parent, and so, level i has $\frac{|V'|!}{(|V'|-i)!}$ nodes,
- every path or sequence of vertices (without cycle or repetition) is mapped to a specific node.

A.2.1 Encoding/decoding paths with exactly H hops - Θ_1/Θ_1^{-1}

Consider $\mathcal{A}_1$ as the set of $s-t$ acyclic paths in G with exactly H hops, observe that each path has $B=(H-1)$ intermediate nodes, and each path can also be considered as a permutation of B nodes. To construct such paths, from a set of $|V'|$ possible intermediate nodes, there are $|\mathcal{A}_1| = \frac{|V'|!}{(|V'|-B)!}$ different ways. When we design the encoding, we are also designing the neighbourhood's structure for the solution space $\mathcal{A}_1$. We choose $\mathcal{B}_1$ as the range $[1, \frac{|V'|}{(|V'|-B)!}]$ of natural numbers, and distribute paths with the same prefix among a same range. Our study reveals that we can map a path with exactly H hops or in other words B intermediate nodes, into the B-th level of a permutation tree with $|V'|$ nodes on the first level. $\mathcal{A}_1$ $\mathcal{B}_1$

encode: $\Theta_1 : \mathcal{A}_1 \to \mathcal{B}_1$ Θ_1

- {input: $P[1:B]$ an array with a permutation of B labels from V'}
- $L = (1, .., |V'|)$ a list; $[l, u] = [1, |\mathcal{A}_1|]$
- for $lev = 1 : B$ do
  - {divide $[l, u]$ into $(|V'| - lev + 1)$ partitions of the same size}[4]
  - j = the position of $P[lev]$ in L
  - $[l, u]$ = the j-th partition of $[l, u]$
  - {remove $L[j]$ from L}
- {output: l (note that l equals to u at this point)}

decode: $\Theta_1^{-1} : \mathcal{B}_1 \to \mathcal{A}_1$ Θ_1^{-1}

- {inputs: $id \in [1, |\mathcal{A}_1|]$ an identification, H the number of hops}
- $P = ()$ an empty list; $L = (1, .., |V'|)$ a list; $B = (H-1)$; $[l, u] = [1, |\mathcal{A}_1|]$}
- for $lev = 1 : B$ do
  - {the range $[l, u]$ is divided into $(|V'| - lev + 1)$ partitions of the same size}
  - {determine j such that the j-th partition is the one that contains id[5]
  - $P = (P : L[j])$ concatenates $L[j]$ at the end of P
  - {remove $L[j]$ from L}
  - $[l, u]$ = the j-th partition of $[l, u]$
- {output: P}

A.2.2 Encoding/decoding paths without hop limit ($H = |V'|+1$) - Θ_2/Θ_2^{-1}

Consider the set of all $s-t$ acyclic paths in G as $\mathcal{A}_2$. Observe that those paths can be classified according to the total number of intermediate nodes they contain, thus $|\mathcal{A}_2| = \sum_{n=1}^{n=|V'|-1} \frac{|V'|!}{(|V'|-n)!}$. We choose $\mathcal{B}_2$ as the range $[1, |\mathcal{A}_2|]$ of natural numbers. We choose to divide the full range of $\mathcal{B}_2$ into sub-ranges and place paths with the same number of hops into a same sub-range, inside each sub-range the coding Θ_1 is used. Once again it can be explained with the help of a permutation tree with $|V'|$ nodes on the first level, the proposed coding enumerates each node of the tree in a breadth first fashion (level by level like in Fig. A.1). $\mathcal{A}_2$ $\mathcal{B}_2$

encode: $\Theta_2 : \mathcal{A}_2 \to \mathcal{B}_2$ Θ_2

[4]Notice that each partition contains exactly $(|V'| - lev)!$ numbers; if $|V'|!$ is available initially, then this step can be done with only 1 division to get the size of partition, 1 multiplication to set u and 1 subtraction to set l.

[5]Done by one single division

- {input: $P[1:B]$ an array with a permutation of labels from V'}
- $u = 0$
- for $i = 1 : (B-1)$ do
  - $\{size = \frac{|V'|!}{(|V'|-i)!}\}$[6]
  - $u = u + size$, accumulates all the previous sub-range
- $id = u + \Theta_1(P)$, reusing the encoding for paths with exactly H hops
- {output: id}

decode: $\Theta_2^{-1} : \mathcal{B}_2 \to \mathcal{A}_2$ Θ_2^{-1}

- {input: $id \in [1, |\mathcal{A}_2|]$}
- $l = 1$; $h = 1$;
- while $(id >= l)$ do
  - $\{size = \frac{|V'|!}{(|V'|-i+1)!}\}$[7]
  - $l = l + size$
  - $h = h + 1$
- $l = l - size$
- $P = \Theta_1^{-1}(id - l + 1, h)$, a path with exactly h hops
- {output: P}

A.3 Enumerating K-paths ($K \in [1, |V'|]$, $H = |V'| + 1$):

The case where $K = 1$ was quite simple, because each arrangement or permutation determines a unique path and the enumeration of all the possible arrangement can be done systematically, but unfortunately this property doesn't hold for $K \geq 2$. Because for a given arrangement of N vertices, if $1 < K < N$, then many different K-paths can be constructed. Remember that we consider a K-path as a set of K paths, or collection of K different paths where the order is ignored. Let's explain it with the following example, consider a complete graph with five vertices (K_5):

The following 2-paths are the same for us (remember than we are omitting the source and the target nodes):

- $< n_3 > < n_1, n_2 >$ ← denotes the 2-path $< s, n_3, t >, < s, n_1, n_2, t >$
- $< n_1, n_2 > < n_3 >$ the same set, because the order is ignored

But the following are 6 different 2-path

- $< n_1 > < n_3, n_2 >$
- $< n_1 > < n_2, n_3 >$ a different set, because the order inside its elements is different
- $< n_2 > < n_3, n_1 >$ in particular, the same 3 vertices can generate $(2 \times C_2^3)$ 2-paths
- $< n_2 > < n_1, n_3 >$ C_2^3 ways to combine them, and each way generates 2 of them
- $< n_3 > < n_2, n_1 >$

[6] Done with one single multiplication.
[7] Done with one single multiplication.

- $< n_3 > < n_1, n_2 >$

The principal difficulty is to enumerate them with unique identifications (a bijection), to do it, we have to find a way to group identical K-paths together and enumerate them systematically. To solve this problem, we follow an idea like in the 1-path case, but using two kinds of trees. In the first place, we have the binomial tree, which is used to identify, arrange and enumerate combinations. Then in the second place, the permutation tree, which can be used to identify, arrange and enumerate permutations. Before exploring the details of the enumeration, we present the two structures and some of their properties.

A.3.1 Structures

The *Binomial* tree:

By "Binomial Tree of N nodes on the first level" we refer to a rooted tree with N nodes in the first level labelled as $\{n_1, n_2, ..., n_N\}$, and each node n_i has $(i-1)$ branches, whose children are assigned smaller labels $\{n_1, n_2, ..., n_{(i-1)}\}$. In Fig. A.2 we show an example of binomial tree of 5 nodes on the first level. By assigning an ordering to the labels, we can also order the binomial tree so each node has its children labelled increasingly.

Properties of a Binomial Tree of N nodes on the first level:

- it has N levels,
- there are C_l^N nodes in the l-th level,
- paths from the root to any leaf contain nodes labelled in decreasing order,
- there is a bijection between any subset of $\{n_1, n_2, ..., n_N\}$ and nodes in the binomial tree (example: the node n_1 in the branch $root - n_4 - n_2 - n_1$ corresponds to the set $\{n_4, n_2, n_1\}$ in Fig. A.2). Moreover, we can map subsets of the same cardinality to the same level of the tree (example: subsets of cardinality 3 are mapped to the 3rd level of the tree in Fig. A.2).
- a binomial tree with N nodes on the first level can be constructed using two copies of a binomial tree with $(N-1)$ nodes, by adding an edge, see fig Fig. A.3.
- the number of nodes in a binomial tree with $(N+1)$ nodes in the first level is exactly twice the number of nodes in a binomial tree of parameter N.

By knowing the number of nodes in the first level, we can map a given set of labels to a node. And by ordering the branches of each node according to their labels, we can assign a well determined order to each node. Therefore, the number of nodes in a given level of the binomial tree that precedes a particular one can be calculated easily.

After a more detailed literature reviewing, we found that the structure binomial tree has already been defined previously, and its properties were also presented in the literature [Cormen 04].

The *Permutation* tree:

By "Permutation Tree of N nodes on the first level" we refer to a rooted tree with N nodes in the first level labelled as $\{n_1, n_2, ..., n_N\}$, and if we follow the nomenclature of a family tree, then we can say that each node has as many children as siblings[8] it has. Moreover, each label in the sibling of a node does appear once and only once in its children. An example of a Permutation Tree of 4 nodes on the first level is shown in Fig. A.4.

Properties of a Permutation Tree of N nodes on the first level:

- it has N levels,

[8]Nodes in the same level of hierarchy, originated from the same parent node. A node is not sibling of itself.

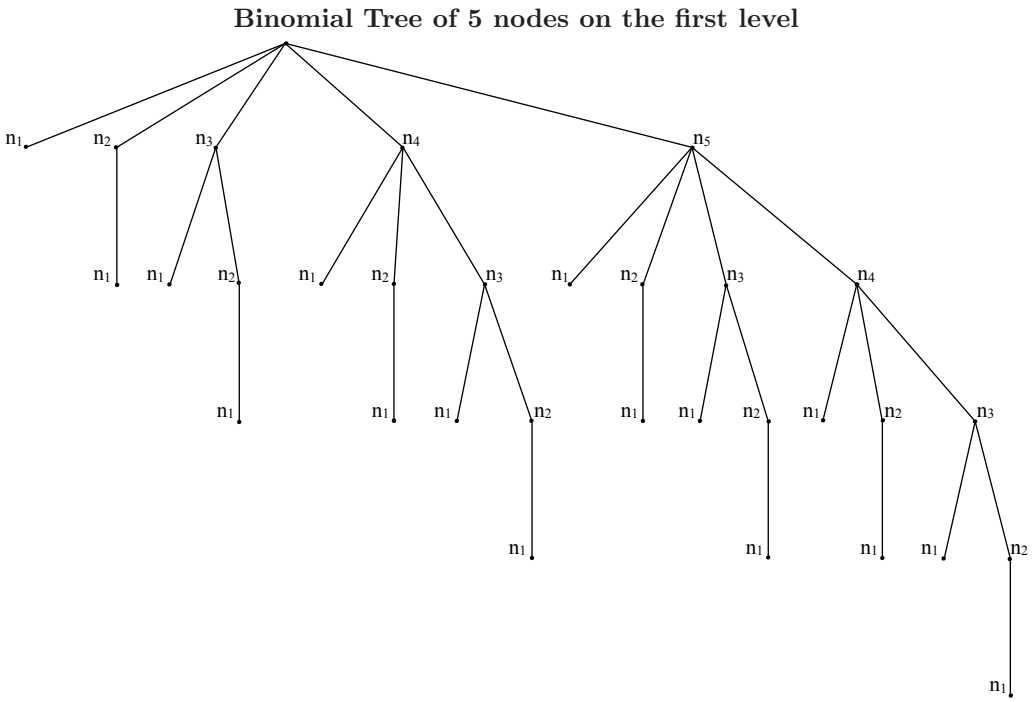


Figure A.2: Each node represents a subset; each level represents all the possible combinations of a given size or cardinality.

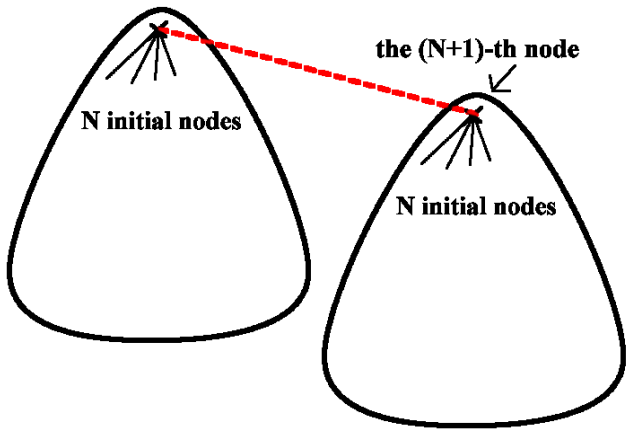


Figure A.3: Two copies of a binomial tree with N nodes on the first level and an edge is all needed for the construction of another one with $(N+1)$ nodes on the first level.

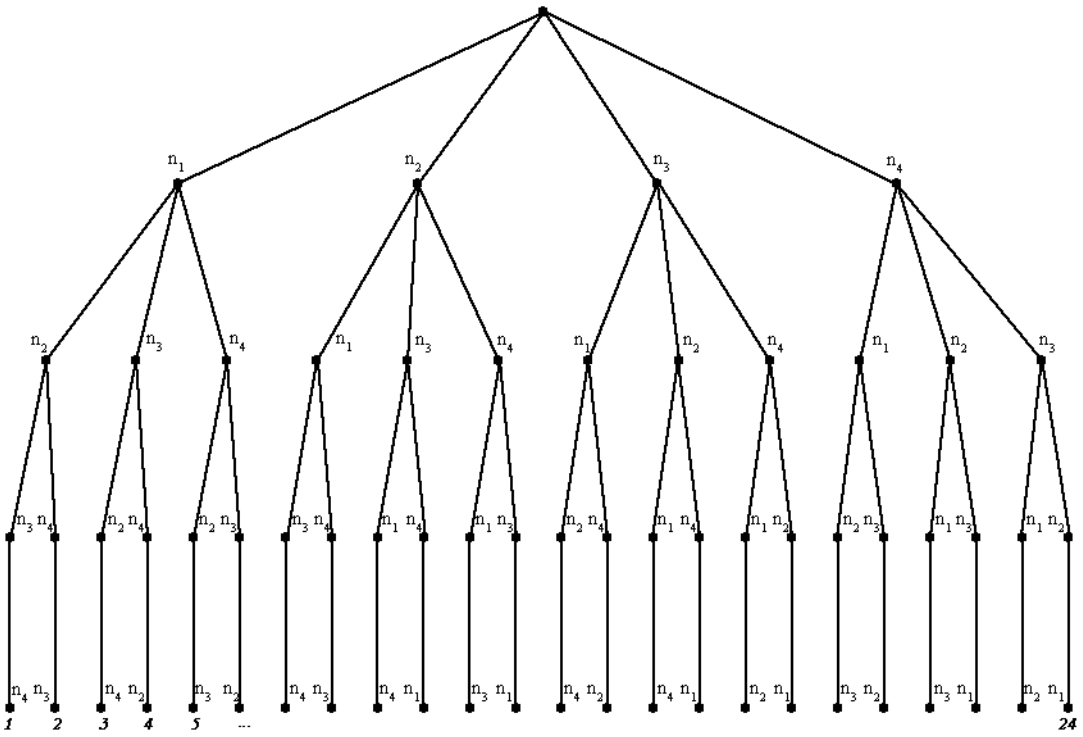


Figure A.4: Each leaf represents a permutation of $N = 4$.

- all the nodes on the same level share the same number of branches,
- all non-root nodes have one less branch than their parent,
- the N-th (last) level or the level of leaves has exactly $N!$ leaves,
- for $0 \leq l \leq N$, the l-th level has exactly $\frac{N!}{(N-l)!}$.
- all the possible permutations can be obtained by taking each one of the leaves and travelling backwards to the root.

By assigning an ordering to the labels, we can also order the permutation tree so each node has its children labelled increasingly. We found in [Kruse 84] another definition of permutation tree, one that has a different node labelling, while having most of the properties listed here.

The *Cycle* tree:

As a by-product in the study of permutation tree, we developed a way to enumerate cycles. Notice that each cycle of $(N+1)$ labels can be written as $(N+1)$ different shifted permutations, furthermore, each shifted permutation can be represented by two permutations considering clockwise or counter clockwise ordering, therefore the number of different cycles containing $(N+1)$ labels is $\frac{N!}{2}$. With "Cycle Tree of N nodes on the first level" we refer to a permutation tree with N nodes on the first level, with an additional ordering constraint over two of the nodes. The ordering constraint removes permutations, in which, a given node appears after another. For example, if we declare that n_1 must precede n_2 when visiting from root to leaves, we have the cycle tree of 4 nodes on the first level as shown in Fig. A.5.
By assigning an ordering to the labels, we can order the cycle tree so each node has its children labelled increasingly.

Properties of Cycle Tree of N nodes on the first level:

Cycle Tree of 4 nodes on the first level, with $n_1 < n_2$

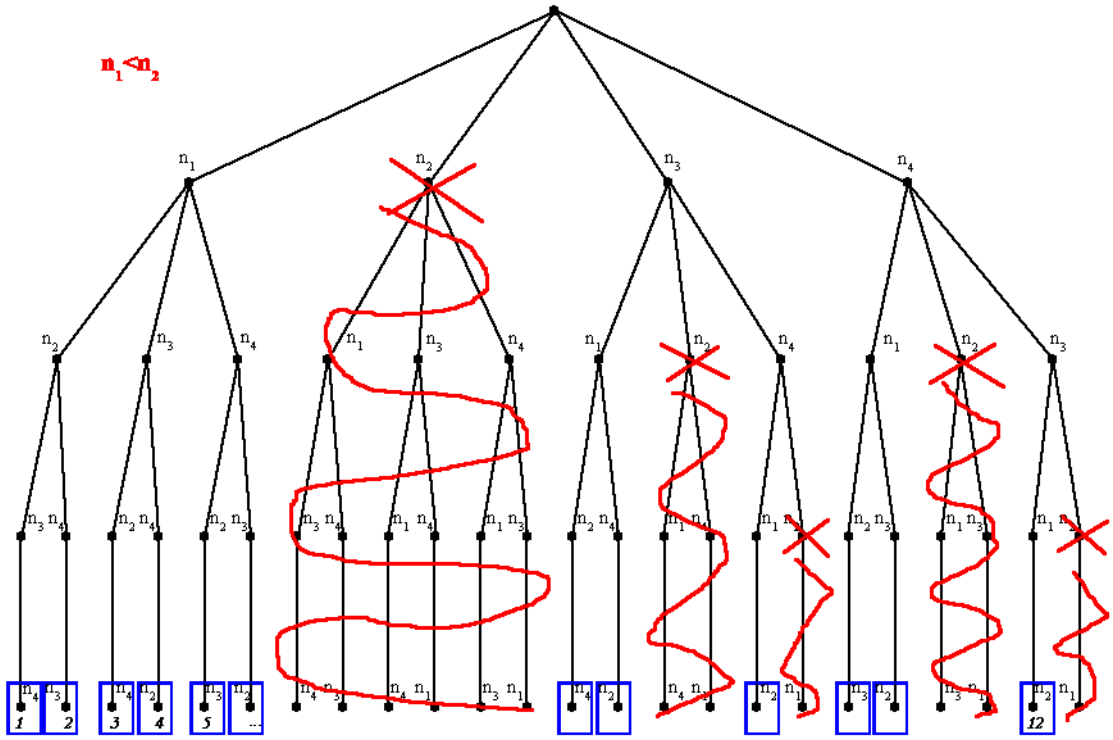


Figure A.5: Each leaf represents a different cycle.

- it has N levels,
- the underlying tree structure is exactly the same as permutation tree,
- when a node is removed, its sub-tree is also removed,
- the ordering constraint $n_1 < n_2$ helps to remove half of the permutations of N labels. For every accepted permutation, its reverse is rejected.
- the last level has exactly $\frac{N!}{2}$ nodes,
- the ordering constraint can be taken over any two arbitrary labels,
- with $(N+1)$ labels, there are a total of $\frac{N!}{2}$ different cycles. We can leave one of the labels apart, namely n_0, and consider a cycle tree with N nodes on the first level. Each leaf matches to a permutation of N labels, by concatenating n_0 to its beginning we have a valid cycle with $(N+1)$ labels.

A.3.2 The big picture:

The next set to consider is $\mathcal{A}_3$, its elements are sets of node-disjoint s-t paths of cardinality K (what we defined as K-paths), but we exclude the direct (s-t) path for this section. $\mathcal{A}_3$

Two K-paths composed of different vertices are always different, and the set of vertices can have cardinality from K to $|V'|$. It means that the sets that map to nodes from the level K to $|V'|$ of a binomial tree with $|V'|$ nodes on the first level are valid for the construction of K-path, moreover, all K-path belong to one and only one node in the binomial tree.

Given a set with N vertices, we are interested in enumerating all the K-paths that can be constructed with them. It's the classical unordered partition problem, where a set of distinguishable objects is to be divided into a fixed number (K) of non-empty partitions, and the way how they are ordered doesn't matter, but the order inside each one of them makes difference. If the set

has N objects, there are $f(N,K) = \frac{N!}{K!} \times C_{K-1}^{N-1}$ different ways to divide them into K partitions [Ríbnikov 85]. Together with the fact that there are $C_N^{|V'|}$ different ways to select N nodes from V', where N is valid in the range $[K, |V'|]$ of integers, we conclude that $|\mathcal{A}_3| = \sum_{lev=K}^{lev=|V'|} C_{lev}^{|V'|} \times f(lev, K)$, and we choose $\mathcal{B}_3 = [1, |\mathcal{A}_3|]$ a range of natural numbers, in Fig. A.6 we show the $|\mathcal{A}_3|$ when K changes.

$f(N,K)$

$\mathcal{B}_3$

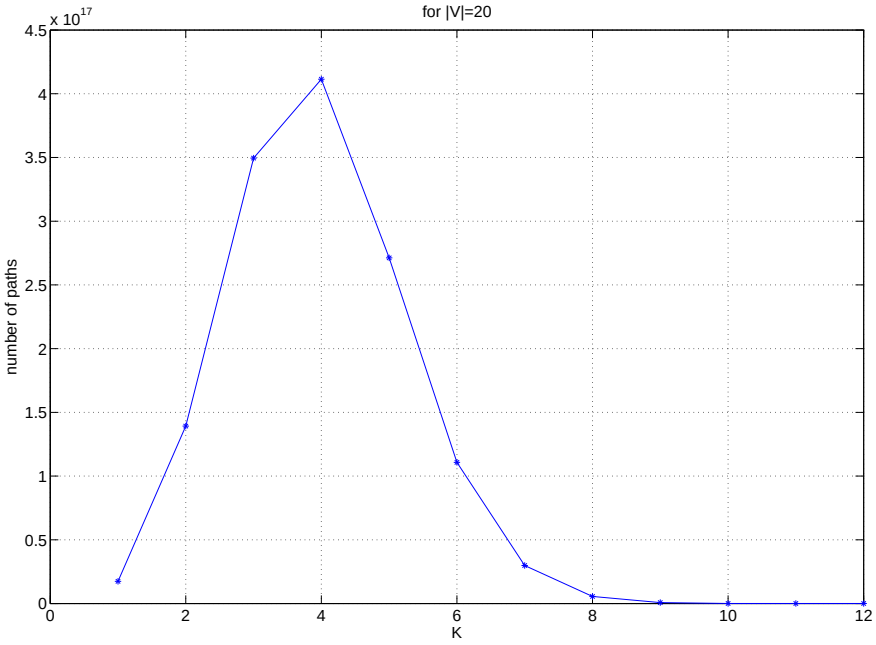


Figure A.6: With $|V|$ fixed to 20, this graph shows the number of K-paths for the different diversity factor K (without considering the direct s-t path as valid).

We can interpret the expression of $|A_3|$ using the structures presented previously.

- In $|\mathcal{A}_3| = \sum_{lev=K}^{lev=|V'|} C_{lev}^{|V'|} \times f(lev, K)$: the factor $C_{lev}^{|V'|}$ can be expressed using a binomial tree with $|V'|$ nodes on the first level, and its nodes enumerated in a breadth first fashion from the K-th to $|V'|$-th level. We call this tree the binomial tree of composition, each node in the lev-th level is an abstraction of $f(lev, K)$ different K-paths, representing all the possible K-paths that can be constructed using that combination of nodes. Every K-path belongs to exactly one node, and it can be identified by sorting the intermediate nodes that compose the K-path.
- In $f(N,K) = \frac{N!}{K!} \times C_{K-1}^{N-1}$: the factor $\frac{N!}{K!}$ can be expressed with nodes of the $(N-K)$-th level of a permutation tree with N nodes on the first level. Each node is an abstraction of a C_{K-1}^{N-1} different K-path, and represents a permutation of $(N-K)$ labels. Note that instead of a permutation of N nodes, the formula suggests that we leave aside exactly K nodes; this is one of the key observations for achieving a bijective mapping. Given a K-path, first we sort it according to the leading node of each path, then concatenate all of them but excluding the K leading nodes, the resulting sequence is the corresponding permutation. Moreover, each node carries an implicit information, a sequence of K leading nodes, those that are left aside from the original N labels.
- C_{K-1}^{N-1}: finally, this number represents the number of ways, in which a sequence of N labels (with $(N-1)$ possible separating points) can be chopped into K parts (by assigning $(K-1)$ separating marks). Each combination identifies a particular way of dividing the sequence into a K-path. Once again we can use a binomial tree to express this factor, resulting in the nodes

of the $(K-1)$-th level of a binomial tree with $(N-1)$ nodes on the first level. We call this tree the binomial tree of marks.

As we are not interested in the permutation of paths inside a K-path, we define the representative format of a K-path as: representative

$< p_1 : tail_1 >< p_2 : tail_2 > ... < p_K : tail_K >$
such that $p_1 < p_2 < ... < p_K$

In other words, we arrange the K-path by the label of the first intermediate node of each path. Later on, when the direct s-t path is admissible, the direct path is placed always in the last position of the arrangement.

$< p_1 : tail_1 >< p_2 : tail_2 > ... < p_{K-1} : tail_{K-1} ><>$
such that $p_1 < p_2 < ... < p_{K-1}$

A.3.3 Encoding/decoding K-paths without hop limit - Θ_3/Θ_3^{-1}

In this case, elements of $\mathcal{A}_3$ are K-paths that don't contain the direct s-t path, and $\mathcal{B}_3$ is the range $[1, |\mathcal{A}_3|]$ of natural numbers.

encode: $\Theta_3 : \mathcal{A}_3 \rightarrow \mathcal{B}_3$ Θ_3

- {input: a K-path}

1. first, using the binomial tree of composition
   - S = a list with the labels of all intermediate vertices in the K-path sorted in decreasing order
   - N = length of S
   - $[l_1, u_1] = [1, C_N^{|V'|}]$
     - the node that we are searching for is in the N-th level of the tree, in that level there are $C_N^{|V'|}$ nodes, each one corresponding to a different combination of vertex.
   - $level = N$
   - $L = (N, .., |V'|)$ a list
   - for $i = 1 : N$ do
     - divide $[l_1, u_1]$ into as many partitions as the length of L (there is a bijection between nodes in L and these partitions), such that the p-th partition has $C_{level-1}^{p+level-2}$ nodes (see Fig. A.7)
     - j = the position of the i-th element of S in L
     - $[l_1, u_1]$ = the j-th partition of $[l_1, u_1]$
     - remove all the elements greater or equals to $L(j)$ from L
     - add $L(1) - 1$ to the beginning of L if possible
       - it adds the next smaller label to the beginning of the list, if there is any
     - $level = level - 1$
   - $[l_1, u_1]$ now contains a unique identification, and it identifies a node in the N-th level of the binomial tree of composition.
2. to get a representative format
   - reorder the given K-path $< p'_1 : tail'_1 >< p'_2 : tail'_2 > ... < p'_K : tail'_K >$ to get $< p_1 : tail_1 >< p_2 : tail_2 > ... < p_K : tail_K >$, where $p_1 < p_2 < ... < p_K$
   - $P = (tail_1 : tail_2 : ... : tail_K)$, a list formed by concatenating the tails of the paths
     - *":" is the concatenation operator*
     - *P is a permutation of $(N-K)$ nodes without considering the first node of each path, it maps to a specific node in the $(N-K)$-th level of a permutation tree with N nodes on the first level (which contains $\frac{N!}{K!}$ nodes)*

- $Marks = (length(p_1 : tail_1), length(p_1 : tail_1 : p_2 : tail_2), ..., length(p_1 : tail_1 : ... : p_{K-1} : tail_{K-1}))$
  - $Marks$ is a list of $(K-1)$ numbers, those are the positions of the separation marks for this K-path
  - $length(p_i : tail_i) \geq 1$ for $i \in [1, K-1]$, because there is no direct $s-t$ link

3. then we use the permutation tree
   - M = reverse of S, an increasing list
   - $[l_2, u_2] = [1, \frac{N!}{K!}]$, and $i = 1$
   - While $i \leq (N-K)$,
     - divide $[l_2, u_2]$ into $(N-i+1)$ partitions of same size
     - j = the position of the i-th element of P in M
     - $[l_2, u_2]$ = the j-th partition of $[l_2, u_2]$
     - remove the j-th element from M
     - $i = i+1$
   - $[l_2, u_2]$ now contains a unique identification, it identifies a node in the $(N-K)$-th level of the permutation tree.

4. we now use a binomial tree of marks to further refine the identification
   - $L_2 = ((K-1), K, ..., (N-1))$, a list
   - $[l_3, u_3] = [1, C_{K-1}^{N-1}]$
   - $level = (K-1)$; $i = 1$
   - While $i \leq (K-1)$,
     - divide $[l_3, u_3]$ into $length(L_2)$ partitions, where the p-th partition has $C_{level-1}^{p+level-2}$ numbers
     - j = the position of the i-th element of $Marks$ in L_2
     - $[l_3, u_3]$ = the j-th partition of $[l_3, u_3]$
     - remove from L_2 all element greater or equals to $L_2(j)$
     - add $L_2(1) - 1$ to the beginning of L_2 if possible
       - it adds the next smaller label to the beginning of the list, if there is any
     - $level = level - 1$; $i = i+1$
   - $[l_3, u_3]$ now contains a unique identification, it identifies a node in the $(K-1)$-th level of the binomial tree of marks with $(N-1)$ nodes on the first level

5. The final result of this algorithm is
   $$id = shift + (l_1 - 1) \times \frac{N!}{K!} \times C_{K-1}^{N-1} + (l_2 - 1) \times C_{K-1}^{N-1} + l_3$$
   where
   $$shift = \sum_{level=K}^{level=N-1} C_{level}^{|V'|} \times \frac{level!}{K!} \times C_{K-1}^{level-1}$$
   because in the binomial tree of composition
   - the first level to be considered is $level = K$
   - each level has $C_{level}^{|V'|}$ nodes, and each node contains exactly $(\frac{level!}{K!} \times C_{K-1}^{level-1})$ identifications

- {output: id}

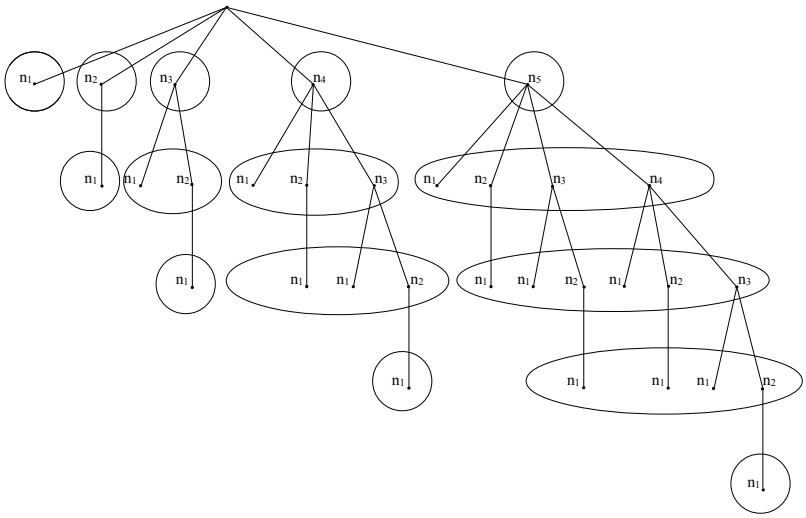


Figure A.7: The number of nodes in each level, that share the same node in the first level.

decode: $\Theta_3^{-1} : \mathcal{B}_3 \rightarrow \mathcal{A}_3$ Θ_3^{-1}

- {input: $id \in [1, |\mathcal{A}_3|]$}

1. first determine the number of vertices in the encoded K-path
   - partition $[1, Max]$ into $(|V'|-K+1)$ partitions, each one corresponds to one level, therefore the p-th partition has $C_{K-1+p}^{|V'|} \times \frac{K-1+p!}{K!} \times C_{K-1}^{K-2+p}$
   - $j =$ the index of the partition to which id belongs
   - $N = K + j - 1$, this is the number of vertices in the K-path
   - $[l, u] =$ the j-th partition's range
   - $marksize = C_{K-1}^{N-1}$, number of nodes in the $(K-1)$-th level of a binomial tree of marks with $(N-1)$ nodes on the first level
2. then determine the nodes that compose the K-path
   - $L = (N, N+1, ..., |V'|)$ a list of the largest $(|V'| - N + 1)$ labels
   - $S = ()$ an empty list
   - $level = N$; $i = 1$
   - $d = (id - l)/(\frac{N!}{K!} \times marksize)$; $q = floor(d) + 1$
     - because each node in the N-th level of a binomial tree of composition contains $\frac{N!}{K!} \times marksize$ identifications
     - $floor(d)$ yields results in [0, $C_N^{|V'|}$-1], and a shift is need to restore the range [1, $C_N^{|V'|}$] for q
   - $size = \frac{N!}{K!} \times marksize$; $[l, u] = [l + (q-1) \times size, l + q \times size - 1]$
     - updating $[l, u]$ with the range of the binomial tree's node
   - $[l_1, u_1] = [1, C_N^{|V'|}]$
   - While $i \leq N$,
     - divide $[l_1, u_1]$ into as many partition as the length of L, such that the p-th partition has $C_{level-1}^{p+level-2}$ nodes
     - determine j such that the j-th partition is the one that contains q
     - assign the j-th element in L to the i-th position in S
     - $[l_1, u_1] =$ the j-th partition of $[l_1, u_1]$

- remove all elements greater or equals to $L(j)$ from L
- add $L(1) - 1$ to the beginning of L if possible
  - ∗ it adds the next smaller label to the beginning of the list, if there is any
- $level = level - 1$; $i = i + 1$

- S now contains N labels, those are vertices that compose the K-path

3. we have to determine how the N vertices are arranged to form the K-path

- $P = ()$ an empty list
- $L = sort\_inc(S)$, sorted increasingly
- $level = 1$
- $d = (id - l)/marksize$; $q = floor(d) + 1$
  - because each node in the $(N - K)$-th level of the permutation tree has $marksize$ identifications
  - the q-th node in the $(N - K)$-th level is the one that contains id
- $size = marksize$; $[l, u] = [l + (q - 1) \times size, l + q \times size - 1]$
  - updating $[l, u]$ with the range of the permutation tree
- $[l_2, u_2] = [1, \frac{N!}{K!}]$
- While $level \leq (N - K)$,
  - divide $[l_2, u_2]$ into as many partition as length of L, all of the same size
  - determine j such that the j-th partition is the one that contains q
  - $P = (P : L(j))$, concatenate the j-th element in L to the tail of P
  - $[l_2, u_2]$ = the j-th partition of $[l_2, u_2]$
  - remove the j-th element from L
  - $level = level + 1$
- P now contains a permutation of $(N - K)$ labels

4. and finally, we determine the way how the N vertices are arranged and separated into K paths

- $H = S \backslash P$, a list with the nodes in S that don't appear in P, there are K of them
- $H = sort\_inc(H)$, order them increasingly
  - we want to reconstruct the representative in the form of $< p_1 : tail_1 >< p_2 : tail_2 > ... < p_K : tail_K >$
  - we have already determined the heads $(p_1, p_2, ..., p_K) = H$
  - and the concatenation of the tails $(tail_1 : tail_2 : ... : tail_K) = P$
  - we still need to know the length of each path to fully re-construct the K-path, among a total of $marksize$ possibilities
- $Marks = ()$, an empty list
- $L_2 = ((K - 1), K, ..., (N - 1))$ a list
- $level = (K - 1)$; $i = 1$
- While $i \leq (K - 1)$,
  - divide $[l, u]$ into as many partition as the length of L_2, where the p-th partition has $C_{level-1}^{p+level-2}$ nodes
  - determine j such that the j-th partition contains id
  - $Marks = (L_2(j) : Marks)$, concatenate the j-th element of L_2 to the head of $Marks$
  - $[l, u]$ = the j-th partition of $[l, u]$
  - remove from L_2 all the element greater or equals to $L_2(j)$
  - add $L_2(1) - 1$ to the beginning of L_2 if possible
  - $level = level - 1$; $i = i + 1$

- $Marks$ now contains $(K-1)$ numbers, those are the separating marks of the K-path, in other words, its the following list:
  $(length(p_1 : tail_1), length(p_1 : tail_1 : p_2 : tail_2), ..., length(p_1 : tail_1 : ... : p_{K-1} : tail_{K-1}))$

5. the K-path is completely determined
   - $Path_1 =<>$; ...; $Path_K =<>$, K empty paths
   - $Marks = (Marks : N)$, add N to the end to simplify the loop
   - $i = 1$; $j = 0$; $l = 1$
   - While $i \leq K$,
     - $Path_i(1) = H(i)$; $j = j + 1$
     - $aux = Marks(i) - j$, the number of nodes to extract from P to form the tail of the i-th path
     - if $aux > 0$ then
       * $tail = sublist(P, l, aux)$; $l = l + aux$
       * $Path_i =< Path_i : tail >$; $j = j + aux$

- {output: a K-path}

A.4 Enumerating Hop-limited K-paths

To add a hop limit constraint to the enumeration of K-paths require some small modifications. If H is the maximum number of hops, then paths with more than $B = (H-1)$ intermediate nodes are no longer valid, furthermore, the total number of intermediate nodes in a H-limited K-path must be in the range $[K, min\{|V'|, B \times K\}]$. This implies that we have to consider a reduced number of levels in the binomial tree of composition. The way of placing separation marks is also affected, but before showing the details, we have to present a new structure and some results from combinatorial analysis.

A.4.1 The *Bounded Binomial* tree

We call a "Bounded Binomial Tree of N nodes on the first level and bound B" to a binomial tree of N nodes on the first level in which the distance of labels between the children and the parent are bounded by B. In other words, for node n_i, we are removing all the children (and their subtrees) whose label is less than $n_{(i-B)}$. Check Fig. A.8 for an example with $N = 5$ and $B = 2$.

Properties of the Bounded Binomial tree of N nodes on the first level and bound B:

- it has N levels,
- each node has at most B branches, by construction,
- paths from root to any node contain nodes labelled in decreasing order,
- a bounded binomial tree with $(N+1)$ nodes on the first level and bound B can be constructed using two copies of a one with N nodes, by adding an edge like in Fig. A.3, and then removing the first branch from the root, if there are more than B branches.
- the number of nodes with label less than or equal to B in each level can be determined, it is a key result and the subject of discussion in Section A.4.2. Furthermore, we can determine the number of those nodes even when there is an additional constraint stating that they must share a set of common ancestors, as a result of the recursive structure of the tree.
- comparing with a binomial tree of same number of nodes on the first level, the number of node in any level of a bounded one is always less than or at most equals to an unbounded one.
- although we call it the bounded binomial tree with N nodes on the first level and bound B, the first level has only B nodes, being N the largest label.

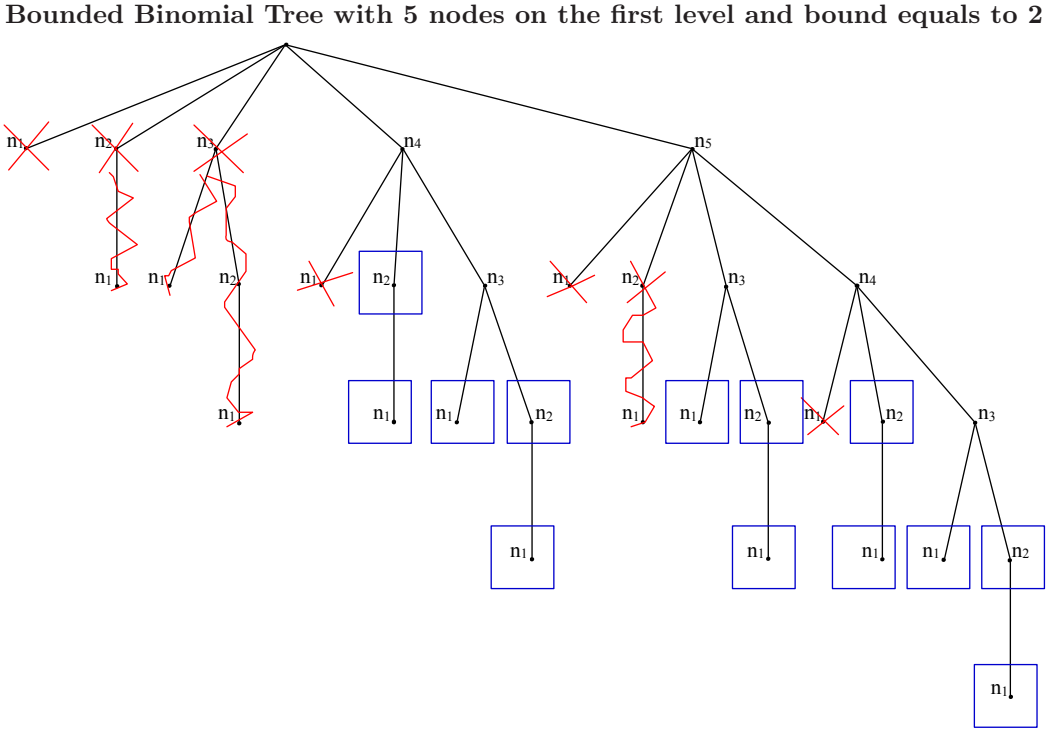


Figure A.8: When a node is removed, all its sub-tree is also removed; each node has at most B=2 children, the two biggest. All nodes (and sub-trees) erased with red crosses do not belong to the tree.

A.4.2 Combinatoric Analysis and Hop limited $s-t$ K-Path Enumeration

Overview of K-path enumeration

A s-t K-path in a complete graph without direct (single hop) s to t path can be identified uniquely by:

- N the total number of intermediate nodes, which must belong to $[K,|V'|]$,
- nodes' composition identified as a set of N nodes, there are a total of $C_N^{|V'|}$ different compositions,
- the shape of the K-path without considering the labels of the nodes, there are a total of C_{K-1}^{N-1} different shapes[9] of K-paths with N intermediate nodes.
- and finally, the sequence in which the nodes appear in the K-path, a total of $\frac{N!}{K!}$.

Notice that the last two steps altogether can have a total of $\frac{N!}{K!} * C_{K-1}^{N-1}$ different cases. Which matches the result of the classical unordered partition problem [Ríbnikov 85], where a set of distinguishable objects (set of N nodes) is to be divided into a fixed number (K) of non-empty partitions (paths). The way how the partitions are ordered doesn't matter, but the order inside each one of them does. To add hop limit, the necessary changes are:

- N the total number of intermediate nodes, which can only belong to $[K,(H-1)K]$
- the possible shapes of the hop limited K-paths are different from the relaxed case

The first item is trivial, the only practical effect is that it truncates a summation. For the case of the second item, a binomial coefficient is replaced by another formula somehow similar, such formula can

[9] Consider N objects placed on a line and $(N-1)$ possible positions where $(K-1)$ separation marks can be placed to get K partitions. There are C_{K-1}^{N-1} different ways to separate them.

be obtained by combinatorial analysis or by studying the bounded binomial tree. In the following, we show the analysis of some combinatorial structures, the goal is to arrive to a formula that can count all the possible shapes of a hop limited K-path, in the same way as binomial coefficient of the binomial tree of marks do for general K-paths.
The objective of the following section is to present some known results of combinatorial analysis, together with their relation to our problem. Then we present the in depth details for the particular case of hop limited K-path. It is not possible to present all the notation and demonstration for all the intermediate results, therefore we refer the readers to [Flajolet 06], which is the literature we followed for this part of the study.

Integer composition

In literature of combinatorial analysis, like in [Flajolet 06], there is a known combinatorial structure called integer composition . A composition of an integer number n is a sequence of integer numbers, such that their sum is exactly n.

integer composition

$$n = x_1 + x_2 + ... + x_p \text{ and } 1 \leq x_i \text{ for } i \in [1, p].$$

Its specification is: $SEQ\{SEQ_{\geq 1}\{Z\}\}$.
Its OGF is: $\frac{1}{1-(\frac{1}{1-z}-1)} = \frac{1-z}{1-2z}$.
Its $[z^n]$ coefficient is: 2^{n-1}.

Integer composition with K summands

We can limit the number of summands to a fixed number ($K > 0$), so the given integer is partitioned into exactly K summands:

$$n = x_1 + x_2 + ... + x_K \text{ and } 1 \leq x_i \text{ for } i \in [1, K],\ K > 0.$$

Its specification is: $SEQ_K\{SEQ_{\geq 1}\{Z\}\}$.
Its OGF is: $(\frac{z}{1-z})^K$.
Its $[z^n]$ coefficient is: C_{K-1}^{n-1}.
Notice that $[z^N](\frac{z}{1-z})^K = C_{K-1}^{N-1}$ is exactly the total number of shapes (or ways to place separation marks) for K-paths of N intermediate nodes.

Integer composition with K summands, each with size between [1..B]

We can impose that each summand can not exceed a given bound B, later we will take $B = (H-1)$ where H is the number of hops. For example, summand of size one maps to path with one single intermediate node, such path has two hops.

$$n = x_1 + x_2 + ... + x_K \text{ and } 1 \leq x_i \leq B \text{ for } i \in [1, K],\ K > 0.$$

Its specification is: $SEQ_K\{SEQ_{1..B}\{Z\}\}$.
Its OGF is: $(z + z^2 + ... + z^B)^K = (\frac{1}{1-z} - \frac{z^{B+1}}{1-z} - 1)^K = (\frac{z-z^{B+1}}{1-z})^K$
Its $[z^n]$ coefficient is given below:

knowing that $\frac{1}{(1-z)^{M+1}} = \sum_{n\geq 0} \binom{n+M}{n} z^n$
$\frac{1}{(1-z)^K} = \sum_{n\geq 0} \binom{n+K-1}{n} z^n$, let's call it b_n b_n

knowing that $(1+z)^M = \sum_{M\geq n\geq 0} \binom{M}{n} z^n$
$(1-z^B)^K = \sum_{K\geq n\geq 0} \binom{K}{n} (-z^B)^n = \sum_{K\geq n\geq 0} \binom{K}{n} (-1)^n z^{(B*n)}$
$(z-z^{B+1})^K = z^K(1-z^B)^K = z^K \sum_{K\geq n\geq 0} \binom{K}{n} (-1)^n z^{(B*n)} = \sum_{K\geq n\geq 0} \binom{K}{n} (-1)^n z^{(B*n+K)}$

```
function r=nchoosekb(n,k,b)
r=coefnkb(n+1,k+1,b);
return;

function r=coefnkb(n,k,b)
% meaningful only when k*b≥n and k≤n
if (k*b<n) | (k>n) | (k<1) | (n<1) | (b<1),
    r=0; return;
end

r=0;
i=0;
for j=0:n,
    if(j==b*i+k),
        aux=(-1)^i;
        aux=aux*nchoosek(k,i);
        aux=aux*nchoosek(n-j+k-1,n-j);
        r=r+aux;
        i=i+1;
    end
end
```

Figure A.9: Implementation in Matlab of *nchoosekb()*

change of variable $j = (B * n + K) \Rightarrow n = \frac{j-K}{B}$, there are $K+1$ sums in the last summation $(z - z^{B+1})^K = \sum_{j=Bi+K, K \geq i \geq 0} \binom{K}{\frac{j-K}{B}} (-1)^{\frac{j-K}{B}} z^j$, let's call it a_j, it evaluates to zero outside the summation. a_j

The original generating function's coefficients can be obtained by applying convolution:
$(z - z^{B+1})^K \frac{1}{(1-z)^K} = \sum_{n \geq 0} (\sum_{0 \geq j \geq n} a_j b_{n-j}) z^n$
$= \sum_{n \geq 0} (\sum_{0 \geq j \geq n, j=(B*i+K), K \geq i \geq 0} \binom{K}{\frac{j-K}{B}} (-1)^{\frac{j-K}{B}} \binom{n-j+K-1}{n-j}) z^n$
As a result: $[z^n](z - z^{B+1})^K \frac{1}{(1-z)^K} = \sum_{0 \geq j \geq n, j=(B*i+K), K \geq i \geq 0} \binom{K}{\frac{j-K}{B}} (-1)^{\frac{j-K}{B}} \binom{n-j+K-1}{n-j}$

Summary: $[z^N](z - z^{B+1})^K \frac{1}{(1-z)^K}$ is all the following:

- integer composition of K summands in the range of [1..B], such that they sum exactly N,
- ways to place $(K-1)$ separating marks on a line with $(N-1)$ possible positions, such that every segment contains at most B elements,
- number of nodes with labels less than or equals to B in the $(K-1)$-th level of a bounded binomial tree with $(N-1)$ nodes on the first level, and bound B.

We include a Matlab code in Fig. A.9, namely *coefnkb(N,K,B)* which implements the formula $[z^N](z - z^{B+1})^K \frac{1}{(1-z)^K}$, together with the function *nchoosekb(N,K,B)*. The latter gives the number of nodes with labels less than or equals to B in a bounded binomial tree with N nodes on the first level and bound B, the relation between them is simply $nchoosekb(N,K,B) = coefnkb(N+1, K+1, B)$. The introduction of $nchoosekb(N,K,B)$ is because of its role with bounded binomial tree, in analogy with the role of $C_K^N = nchoosek(N,K)$ in binomial tree. Moreover, we also introduce a notation ${}^B C_K^N$ to symbolise $nchoosekb(N,K,B)$. ${}^B C_K^N$
Note that the efficiency can be improved by calculating binomial coefficients using already calculated ones, or even better to pre-calculate and store some of them, that is the approach followed in the code of our genetic algorithms. In Fig. A.10 we show how the number of H-limited K-paths changes with the parameters H and K when the number of nodes $|V|$ is fixed.

A.4.3 Encoding/decoding H-limited K-Paths - Θ_4/Θ_4^{-1}

Being a K-path with hop limit H, it has no more than $(H-1) \times K$ intermediate nodes. To have a limit H on the number of hops, is to have a limit $B = (H-1)$ on the number of intermediate B

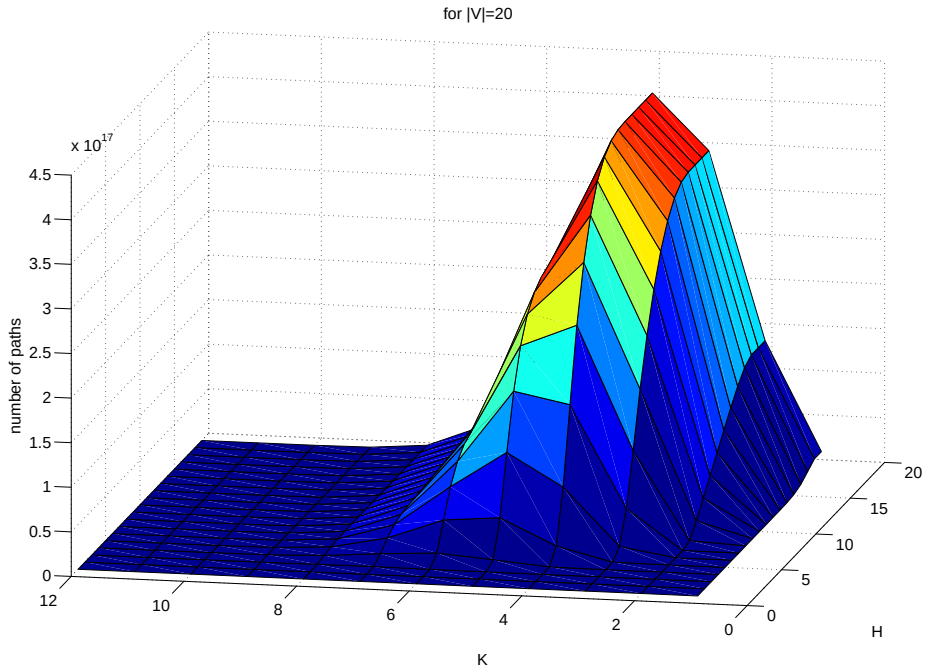


Figure A.10: With $|V|$ fixed to 20, this graph shows the number of H-limited K-paths for the different combination of diversity factor K and hop limit H(without considering the direct s-t path as valid).

nodes in each path. We find that the latter is more intuitive in part of the explanation, so we will use both of them from now on. The idea is very similar to the algorithm presented in Section A.3.3, with only two differences:

- the composition or set of intermediate nodes has cardinality between $[K, min\{|V'|, B \times K\}]$,
- the binomial tree of marks is replaced by a bounded binomial tree, and therefore its related binomial coefficients C_Y^X are replaced by ${}^BC_Y^X$ which is the function *nchoosekb(X,Y,B)*.

In practice, the function $f(N,K) = \frac{N!}{K!} \times C_{K-1}^{N-1}$ is replaced by $f(N,K,B) = \frac{N!}{K!} \times {}^BC_{K-1}^{N-1}$. Therefore, the size of the current set $|\mathcal{A}_4| = \sum_{lev=K}^{lev=min\{|V'|,B\times K\}} C_{lev}^{|V'|} \times f(lev,K,B)$, and once again we can choose $\mathcal{B}_4 = [1, |\mathcal{A}_4|]$ a range of natural numbers. $f(N,K,B)$ $\mathcal{A}_4$ $\mathcal{B}_4$

encode: $\Theta_4 : \mathcal{A}_4 \rightarrow \mathcal{B}_4$ Θ_4

1. identical to Θ_3, calculate l_1
2. identical to Θ_3, reformat the H-limited K-path in a representative form
3. identical to Θ_3, calculate l_2
4. now we use a bounded binomial tree to further refine the identification
   - $L_2 = (N-B, .., N-1)$**, a list**
   - $[l_3, u_3] = [1, {}^B C_{K-1}^{N-1}]$
   - $level = level = K-1$; $i = 1$
   - While $i \leq (K-1)$,
     - divide $[l_3, u_3]$ into $length(L_2)$ partitions, **where the p-th partition has ${}^BC_{level-1}^{L_2(p)}$ numbers**

- j = the position of the i-th element of $Marks$ in L_2
- $[l_3, u_3]$ = the j-th partition of $[l_3, u_3]$
- $L_2 = (max\{1, Marks(i) - B\}, ..., Marks(i) - 1)$ **replacing** L_2
- $level = level - 1$; $i = i + 1$

- $[l_3, u_3]$ now contains a unique identification, it identifies a node in the $(K-1)$-th level of the bounded binomial tree with $(N-1)$ nodes on the first level

5. **The final result of this algorithm is**
   $id = shift + (l_1 - 1) \times \frac{N!}{K!} \times^B C_{K-1}^{N-1} + (l_2 - 1) \times^B C_{K-1}^{N-1} + l_3$
   where
   $shift = \sum_{level=K}^{level=N-1} C_{level}^{|V'|} \times \frac{level!}{K!} \times^B C_{K-1}^{level-1}$

decode: $\Theta_4^{-1} : \mathcal{B}_4 \to \mathcal{A}_4$ Θ_4^{-1}

1. first determine the number of vertices in the encoded K-path
   - partition $[1, Max]$ into $(|V'| - K + 1)$ partitions, each one corresponds to one level, **therefore the p-th partition has** $C_{K-1+p}^{|V'|} \times \frac{K-1+p!}{K!} \times^B C_{K-1}^{K-2+p}$
   - j = the index of the partition to which id belongs
   - $N = K + j - 1$, this is the number of vertices in the K-path
   - $[l, u]$ = the j-th partition's range
   - $marksize =^B C_{K-1}^{N-1}$**, number of nodes in the** $(K-1)$**-th level of a bounded binomial tree of marks with** $(N-1)$ **nodes on the first level and bound** B
2. identical to Θ_3^{-1}, reconstruct S the list of composition
3. identical to Θ_3^{-1}, reconstruct P the list of permutation (nodes that form the concatenation of $tail_i$)
4. and finally, we determine the way how the N vertices are arranged and separated into K paths
   - $Heads = S \backslash P$, a list with the nodes in S that don't appear in P, there are K of them
   - $Heads = sort\_inc(Heads)$, order them increasingly
   - $Marks = ()$, an empty list
   - $L_2 = ((N - B), ..., (N - 1))$ **a list**
   - $level = (K - 1)$; $i = 1$
   - While $i \leq (K - 1)$,
     - divide $[l, u]$ into as many partition as the length of L_2, **where the p-th partition has** ${}^B C_{level-1}^{L_2(p)}$ **nodes**
     - determine j such that the j-th partition contains id
     - $Marks = (L_2(j) : Marks)$, concatenate the j-th element of L_2 to the head of $Marks$
     - $[l, u]$ = the j-th partition of $[l, u]$
     - $L_2 = (max\{1, Marks(i) - B\}, ..., Marks(i) - 1)$ **replacing** L_2
     - $level = level - 1$; $i = i + 1$
   - $Marks$ now contains $(K-1)$ numbers, those are the separating marks of the H-limited K-path, in other words, its the following list:
     $(length(p_1 : tail_1), length(p_1 : tail_1 : p_2 : tail_2), ..., length(p_1 : tail_1 : ... : p_{K-1} : tail_{K-1}))$
5. identical to Θ_3^{-1}, reconstruct the H-limited K-path from the lists $Heads$, P and $Marks$

A.4.4 Encoding/decoding H-limited K-Path admitting direct s-t path - Θ_5/Θ_5^{-1}

The only difference with the preceding case is that now the direct path is counted as a valid path. Note that a K-path may contain at most one direct path, and we can divide the set $\mathcal{A}_5$ into two sets. Where the first of them $\mathcal{A}_{5a}$ contains all the K-paths with direct path, and a second one $\mathcal{A}_{5b}$ containing all the others. Moreover, the former is equivalent to a set of $(K-1)$-paths without direct path, so we have $|\mathcal{A}_5| = |\mathcal{A}_{5a}| + |\mathcal{A}_{5b}| = (\sum_{lev=K-1}^{lev=|V'|} C_{lev}^{|V'|} f(lev, K-1)) + (\sum_{lev=K}^{lev=|V'|} C_{lev}^{|V'|} f(lev, K))$ and we choose $\mathcal{B}_5 = [1, |\mathcal{A}_5|]$ a range of natural numbers.

encode: $\Theta_5 : \mathcal{A}_5 \rightarrow \mathcal{B}_5$

- {input: KP a K-path in the representative form}
- {determine if there is a direct path in KP and set the flag $hasDirect$}
- if $hasDirect == true$ then
- $id = \Theta_4(KP[1 : (K-1)]) \leftarrow$ treated as a $(K-1)$-path[10]
- else
- $shift = |\mathcal{A}_{5a}|$
- $id = shift + \Theta_4(KP)$
- end if
- {output: id}

decode: $\Theta_5^{-1} : \mathcal{B}_5 \rightarrow \mathcal{A}_5$

- {input: $id \in [1, |\mathcal{A}_5|]$}
- set initially: $shift = |\mathcal{A}_{5a}|$
- if $id \leq shift$ then
- $KP = \Theta_4^{-1}(id, K-1) \leftarrow$ treated as a $(K-1) - path$
- $KP[K] = [] \leftarrow$ set an empty path at the end
- else
- $KP = \Theta_4^{-1}(id - shift, K)$
- end if
- {output: KP the K-path}

A.5 Summary and Remarks

In this article we presented algorithms for mapping and reverse mapping between:

- the set of K-paths and a range of natural numbers
- the set of Hop-limited K-paths and a range of natural numbers

[10] Although Θ_4 was defined for $0 < K \leq (|V'| - 1)$, we can extend it so it returns $id = 1$ when $K == 0$. In this way, it is more consistent with the following coding. Analogously, Θ_3^{-1} should return an empty KP when $K = 0$.

During the development of those mappings, we worked with a number of useful tree structures, namely permutation, binomial, bounded binomial and cycle trees; we found that some of them were already known in the literature, while the last two are new contributions from this thesis. We also applied combinatorial analysis to relate the different kinds of K-paths to other combinatorial structures.

To apply the results of this article, it is important to keep in mind that the number of elements in a set of K-path can be extremely large when V the number of vertices in the complete graph grows. As a result, the implementation of our algorithms requires the usage of mathematic library with large enough precision.

If the algorithm is to be used frequently, the intermediate results of the different binomial coefficients and the hop-limited variation (${}^{B}C_{K}^{N}$) should be saved to increase the efficiency.

Appendix B

GGEngine 2.0

B.1 Abstract:

The present report replaces [Ho 02], it serves as a short user's guide and a summary of the development of Generic Genetic Engine (GGEngine) 2.0, a second version of our genetic algorithms library in C. In short, this library implements the skeleton of genetic algorithm (G.A.) which is an invariant among different G.A., together with some of the most popular codings and genetic operators. Users can use the offered components directly or extend them further with customised codings and operators.

There exist many libraries of genetic algorithms, as an example there is [Grefenstette 90] which is considered as a pioneer work in the community. The development of the GGEngine library begun in 2002 with the goal of simplifying the application of genetic algorithms, and of reducing the required work in implementation done by researchers.

GGEngine is one of the first steps in our goal to achieve a parallel generic genetic engine, with support for the different parallel and distributed architectures available in our institute. The version 2.0 is an improved serial version, and it has already been used in a number of projects.

B.2 Introduction

B.2.1 Motivation

Genetic Algorithms is a meta-heuristic for treating optimisation problems, which is dated back to the 1960s. Based on the simulation of the process of evolution, in which only the most suitable variations of each species survive nature's selection, and generation by generation the entire population becomes more suitable to the environment.

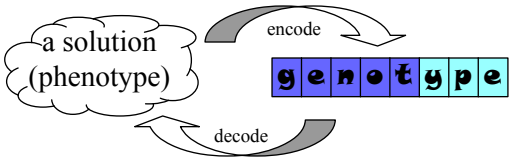
The main goal of this library is the creation of a tool (an engine) that simplifies the application of this technique to solve problems in general. The first version of GGEngine was developed in 2002, in InCo (Instituto de Computación) from the Facultad de Ingeniería, Universidad de la República, in Uruguay. It has been used in a number of research projects and many additional features has been implemented since the release of the first version, up to a point that justifies another distribution together with an update in the documentation.

B.2.2 Genetic Algorithm

The basic structure common to all genetic algorithms (G.A.) is called the skeleton, which is the invariant part of G.A. On the other hand, there are parts of the implementation that change from a G.A. to another, we implement them as components. To apply G.A. to solve a certain problem, it is necessary to model the problem correctly by choosing a suitable implementation for each of the following components.

Coding for a certain problem:

- An injective function to encode solutions (**phenotypes**) of a problem into individuals (**genotypes**) of the population, each individual is composed by atomic components called **genes**. This is complemented with an inverse function, so a genotype can be transformed back to the solution to the problem.

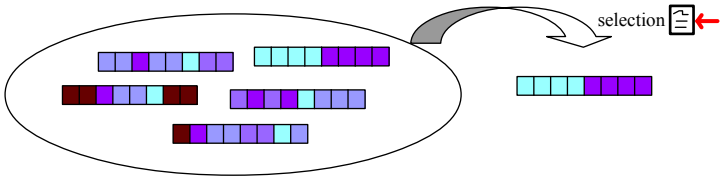


- The *fitness* function returns for each solution (phenotype) a real number representing how suitable or how fit it is for the problem. It is strongly related to the objective function.

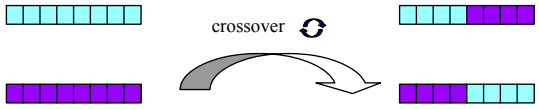
$$fitness : \{a\ solution\} \rightarrow R^{+}$$

Genetic operators:

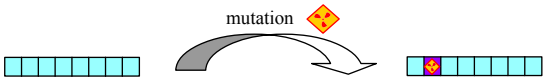
- **Selection**: an operator that chooses in each generation, the most suitable individuals, so they can have a chance to propagate to the next generation.



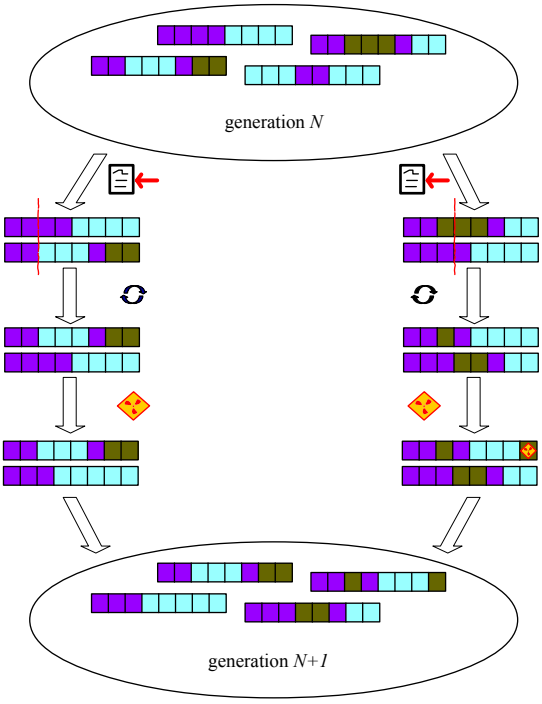
- **Crossover**: an operator that takes two or multiples individuals and generates offspring by combining their genes. Traditional crossover operators involve two genotype parents, but in general any number of genotypes can be used as parent.



- **Mutation**: an operator that introduces variations by changing some of the genes of a given individual. It plays an important role in the escape of local optima which are very common in optimisation problems.



To simulate evolution, the algorithm begins with the first generation, which can be obtained randomly or as a result of some other heuristic. Then it generates the next generation by first applying the selection operator, and later the crossover and finally mutation. According to the theorem of schemes [Goldberg 89], the population will converge (evolve) to good solutions of the problem, given a large enough number of generations.



B.2.3 List of requirements

Be easy to use:
To reduce the effort required to apply G.A., the engine to implement must offer the most common codings and operators ready to use, and beside that, it should allow an easy integration with user defined codings or operators. Changes to configuration parameters, and types of genes or operators to use should not require recompilation or re-linkage. The list of basic components ready to be chosen is as follow. Codings: list of bits, integer, real; Crossover operator: one-point, two-points, n-points, uniform, bias; Mutation: uniform; Selection: roulette, tournament, ranking.

Parallelism as the next step:
Many interesting optimisation problems involve a huge among of data, or require intensive processing. Although this is a serial G.A. engine, our plan is to extend it in the future with distributed and parallel processing techniques. At the beginning of the project, the state of art of most popular distributed programming libraries like PVM, MPI, etc had a more advanced development for C than for C++, that's why we chose C as programming language.

Things to improve from last version:
We had an issue in the first version of the library, it wasted too much time with random number generation in the mutation and crossover operator, which is corrected in the current version by replacing calls to Bernoulli distribution with a more suitable geometric one.
In the current version, we also added the capability of user correction mechanism call backs, in this way if the result after applying crossover and mutation is not feasible for the problem, users can implement their problem specific corrections. Among the additions in this version, we have elitism and fitness scaling, which are reported in the literature to have a positive effect on the solution quality.
Define interface for a new component to manage the storage and access of problem specific data inputs, and another for user defined data structures. The latter encapsulates data structures to use in the different user defined operators.

B.2.4 Layout of the report

In the next two section, we describe the engine's design, beginning in Section B.3 with the architecture, and each of the modules, following by the implementation details in Section B.4.

Later on, in Section B.5 we present a small series of basic tests, and finally in Section B.6 we list the future steps and unfinished tasks.

Find in Section B.7 the API of the engine, Section B.8 format of the configuration file, and a brief user's manual in Section B.9.

B.3 Architecture of GGEngine

B.3.1 Introduction

The main objective of this library is to offer an engine that can solve a large variety of problems, and different problems can be modelled in many different ways. Therefore the engine must use abstraction to stay independent of the representation. In practice, we apply a modular architecture, and an interchangeable components design that allows the user to choose components like type of genes, specific genetic operators, etc., from a pool of components.
The library is divided into three modules: "Population", "Engine", and "Utilities".

B.3.2 Module: "Population"

Population is one of the basic structures in the engine, and in this module we encapsulate the representation and manipulation of population related data structures. A population is a set of individuals (genotypes), and each individual is represented as an arrangement of genes. The different components in this module are: *Population*, *Genotype*, and *Gene*.

- *Population*: implements a dictionary of genotypes and offers the related operations like insertion, data access, etc.
- *Genotype*: represents each genotype or chromosome[1], it can be implemented as a list of bits, list of integers, list of decimals, among others. We design genotype as an abstract collection of genes.
- *Gene*: represents a gene (a genetic characteristic), it can be implemented as a bit, an integer, a decimal, among others.

B.3.3 Module: "Engine"

This module encapsulates the core of the engine, and is composed by the following components: *Operators*, *Coding*, *Engine*.

- *Operators*: there are three kinds of genetic operators, being *Selection*, *Crossover* and *Mutation*. They can be implemented in many different ways, and being our interest to offer a number of varieties, it is necessary to define an interface in such a way that abstract components can be replaced transparent to the system. Therefore, we will use abstract genotype and gene in their definition.
  - *Selection*: the most popular selection operators are roulette (standard proportional selection), tournament, ranking, among others. By taking as input a given population and the fitness evaluation of each of its members, this operator chooses a selected few as output. In general, it is not a deterministic procedure, but a stochastic one. Beside the already mentioned inputs, configuration parameters may also be required.

    $$Population + Evaluation + Random\ Generator + Configuration \rightarrow selected\ Population$$

  - *Crossover*: the most popular crossover operators are one point, two points, n points crossover[Goldberg 89]; uniform crossover[Syswerda 89]; biased crossover, among others. In the traditional definition of this operator, a pair of genotypes is used as input, but to simplify the engine's design, we use an equivalent definition where the input and output are populations. In this way the input is a parent population, and the operator generates the offspring generation according to the configuration parameters. For non-deterministic crossover operators, the random generator is an important input, therefore we add it as another parameter in the interface.

    $$Population + Random\ Generator + Configuration \rightarrow children\ Population$$

[1] In biology, genotype is a group of chromosomes. Human beings have 46 chromosomes and each one is a sequence of DNA. In G.A. we use both genotype and chromosome as synonymous.

– *Mutation*: uniform mutation is the most popular mutation operator. Once again we use the definition over a population instead of over individuals. Given a population, the operator will apply some sort of mutation according to the configuration parameter and with the help of the random number generator, it generates a mutated population.

$$Population + Random\ Generator + Configuration \rightarrow mutated\ Population$$

- *Coding*: this component encapsulates all the problem specific changes, it is divided into two primary parts, these are *Coding* and *Fitness*. The first one represents the encode/decode functions that map genotypes into and out-of phenotypes, in other words, they translate problem solutions into genotypes. The second one of them, *Fitness* as its name says, contains the fitness function, which gives a fitness value to each solution. These two components are problem specific, and therefore are not available to choose from the pool of components.

- *Engine*: encapsulates the skeleton of G.A., it manages populations and applies genetic operators as shown in the following scheme.

```
{generate randomly/read a first generation}
                                ← generation G_0
while {stopping condition}
                                ← generation G_N
    {select the most suitable individuals with selection op.}
    {apply crossover op. over the selected population}
    {apply mutation op. over the children population}
                                ← generation G_{N+1}
    end
```

B.3.4 Module: "Utilities"

All the other components are grouped into the Utility module, here we have: *Configuration*, *Random*, *DB-fitness*.

- *Configuration*: all the configuration parameters of the engine are stored into a dictionary structure called Configuration. It manages the data access and configuration file reading.

- *Random*: G.A. relies strongly on the generation of pseudo-random numbers, this component is the core of pseudo-random number generator. In the first version, we used a simple linear congruent algorithm, and in this version, we offer two generators. The first one is the Mersenne Twister[Matsumoto 98], and on the second place, a linear congruent generator from the library GMP.

- *DB-fitness*: it is of our interest to study and develop mechanisms that improves the performance of G.A., one possibility is the usage of cache in the evaluation of fitness. Therefore we created this component with a defined interface, so the user can integrate a customised cache easily.

B.4 Implementation

B.4.1 Introduction

In this section we discuss the implementation of the modules presented in the previous section. Using the technique of function pointer together with a component based design, we show how a generic genetic algorithms engine can be constructed.

B.4.2 Module: "Population"

As basic type of *Gene*, we admit Booleans, integer, decimals and user defined. Whenever it is necessary to instantiate the gene type, we use function pointers instead of explicit function calls, in this way the user can change gene type without recompilation.

As for *Genotype*, list of *Genes* is the unique implementation that we offer, as it is a very general representation with which many problems can be modelled. Moreover, user defined structures can be integrated thanks to the abstract interface.

We implemented the classical abstract data type list in the case of *Population*.

B.4.3 Module: "Engine"

Operators

The level of abstraction in the module "Population" allows the operators to manage the different entities in an abstract way, in practice, one single procedure can be used together with all kind of *Gene* or *Genotype*.

A large number of genetic operators are implemented, and more can be added using the interface reserved for user defined operators. To active a particular operator, it is enough to active the correspondent parameter in the configuration file. Once again, we benefit from the usage of function pointer, to achieve this.

In the case of selection operator, three implementations are available: roulette (standard proportional selection), tournament, and ranking. This is the only operator that uses the fitness function and we can insert calls to the fitness cache right here, more details about fitness cache can be found in Section B.4.4.

As for the crossover operator, the following bi-genotype crossovers are implemented: one point, two points, uniform at one point, uniform at two points, and biased.

And for mutation, only two mutation operators are implemented. The first one is inversion, which can only be applied in combination with Boolean *Gene* type. The other one is uniform mutation, which is a general version applicable to all type of *Gene*. One of the changes from the previous version is the replacement of multiple Bernoulli sampling, with a single geometric one, in this way we reduced the CPU consumption. A pseudo-random number generator using uniform distribution is used, then given U an uniform variable in $[0, 1]$, we can obtain a geometric variable $geometric(p)$ as $\lceil \frac{log(U)}{log(1-p)} \rceil$, more details can be found in [Devroye 86].

After the application of crossover or mutation operators, it is possible to obtain unfeasible solutions for some combination of coding and problems, in those cases, correction mechanisms may be necessary. To include correction mechanisms, user defined procedures can be inserted after the mutation operation.

Coding

To model an optimisation problem, the user has to implement this component respecting the defined interface (in Section B.7). This component contains the following.

- phenotype definition
- encode function: phenotype → genotype
- decode function: genotype → phenotype
- fitness function: fitness function: phenotype → $\mathbb{R}$

If the genotype and genetic operators to use are already implemented, users only have to implement this component and compile/link it to the library to get a functional G.A.

Engine

Although this is the core component of the engine, its implementation is a simple loop as shown in the scheme of the skeleton. It manages the population and calls the other abstract components using the technique of function pointer.

One of the configuration parameters is the number of generations to run, which is the only implemented stopping condition. When the engine reaches the desired number of generation, it saves all the data and further runs can be performed from the saved point. Other stopping condition like detection of external signal, or reaching a certain fitness value can be included easily, but were not included in this version.

B.4.4 Module: "Utilities"

Configuration

This is a traditional data dictionary, with the traditional operations and configuration file reading. The configuration file contains entries in the form of "*Name = Value*", a two-level name mapping is implemented, to allow an easy change of interface language. We also decided to allow comments to be included into the configuration file, and to be as flexible as possible, we truncate lines whenever the comment character # is detected.

Random

Two pseudo-random generators are available: the first one is based on the Mersenne Twister, in the web site of its author we located an implementation *mt19937a* which we managed to integrate into your library after some small changes. The Mersenne Twister is a 32 bit integer generator with a period of $2^{19937} - 1$, it is reported to be competent in speed and memory usage [Matsumoto 98], [Matsumoto 06].

The second generator is the one from the library of GMP, a library that implements an arbitrary precision arithmetic. In this case, the generator uses a linear congruent algorithm in the form of $X_{n+1} = (AX_n + B)\ mod\ C$, where A, B, C are constants with X_0 the seed. We remind the user that as for all pseudo-random generators the seeding is an important element, especially during a statistical study or comparison of different run of a stochastic algorithm. Seeds should be chosen wide apart in the case of linear congruent generator, one easy way to do it is by running the sequence, and take different X_n far away from each other as seeds of different stochastic algorithms. The drawback of linear congruent generators is that the seed only shifts the sequence, and if it is not chosen with care, one can end up with small shifts.
For the case of Mersenne Twister, seeds with small Hamming distance or which share similar bit pattern will produce shifted sequences.

DB-fitness

Different alternatives were implemented for this component: no-cache, and double-cache. The first one is an empty interface that calls directly the underlying fitness function. The case of double-cache uses two caches, the objective here is to have old cached elements to be removed. Each certain number of generations, the active cache is switched into obsolete mode while the other is emptied and set as active. Then each cache query will go firstly to the active and then the obsolete cache, if it is a new entry, the fitness function is called after all. Whenever a result that doesn't belong to the active cache is obtained, it is stored into the active cache.

B.5 Basic testing and analysis of results

The following experiment is present as an upgraded repetition of the same experiment done in [Ho 02]. A series of test was performed to test the functionality of the engine, the series consists of trivial problems that test specific features of the solver. We first list the set of parameters in use during the experiment, and then the list of chosen problems together with the result and its analysis. This is not an exhaustive validation experiment, which would be out of the scope of this report.

Configuration A:

- number of generations: 150
- population size: 200
- mating pool size: 100
- genotype length: 10
- crossover rate: 0.7
- mutation rate: 0.01
- selection operator: roulette
- crossover operator: 1 point
- mutation operator: uniform

Configuration B:

- number of generations: 50
- population size: 100
- mating pool size: 50
- genotype length: 10
- crossover rate: 0.7
- mutation rate: 0.01
- selection operator: roulette
- crossover operator: 1 point
- mutation operator: uniform

Problem 1:

- Binary coding of natural numbers,
- in a search for the maximum,
- of the linear function $f(x) = x$
- In GGEngine v1.0, configuration A reached the optima value of 1023, while B only reached 959. With the current version, both configurations reached the optima value in experiments run with three different random seeds.

Problem 2:

- Binary coding of natural numbers,
- in a search for the maximum,
- of the step function: $f(x) = \begin{cases} 40 \text{ if } x \leq 8 \\ 3 \text{ otherwise} \end{cases}$
- In this problem the rate for the number of optimal solutions vs. size of the solution space is $\frac{3}{1024}$; the function doesn't help/lead the search to the optima solution (this is called in the literature a "deceptive" problem). But once again, both configurations reached the optima solution, with the three different seeds. The same happened in the previous version. This test is an easy one as it allows us to do 30,000 and 5,000 evaluations to find a solution in a solution space of size 1024. If the random number generation is not biased, then there should be no problem.

Problem 3:

- Binary coding of natural numbers,
- in a search for the maximum,
- of the step function: $f(x) = \begin{cases} 3.1 \text{ if } x \leq 8 \\ 3 \text{ otherwise} \end{cases}$
- Similar to problem 2, this is another step function, but in this instance the function value's difference is very small. In the previous version of GGEngine, any of the configurations reached the optima without increasing the mutation rate. In this version, we managed to reach the optima for both configurations without increasing the mutation rate. Once again this is an easy problem, the primary difficulty is that the difference between the best and worst solution is very small, the engine should be able to detect the improvement before getting to the optima.

Problem 4:

- Binary coding of natural numbers,
- in a search for the maximum,
- of the function: $f(x) = \begin{cases} 100 - (x - 128)^2 \text{ if } 118 \leq x \leq 138 \\ 50 \text{ otherwise} \end{cases}$
- the optima value is 100, for $x = 128$
- $\frac{2}{3}$ of the cases with configuration A reached the optima value of 100; $\frac{1}{3}$ of configuration A and all cases of the B one reached the value 99, which is a good approximation. In the previous version, the optima value was not reached, only the value 99 was done for configuration A.

Although these are only basic functional tests, we can observe an overall improvement comparing to the previous version.

B.6 Plan for next version

The current version implements most of the features in the wish list of the test users. Some missing features are:

- more stopping condition to choose from;
- a more complete documented test case to validate the functionality of the different types of gene-operator combinations;
- a more user friendly implementation of statistic report.

On the list of objective for the next version, one of the ideas is to apply parallel and distributed computing technique to extend the engine, and offer support for the different platforms available in our laboratory, like clusters, shared memory nodes, etc.

B.7 API

Files in each module:

- "Population"
  - "*user_gene_def.h*"
  - "*user_gene.h .c*"
  - "*gene.h .c*"
  - "*genotype.h .c*"
  - "*population.h .c*"
- "Engine"
  - "*selection.h .c*"
  - "*crossover.h .c*"
  - "*mutation.h .c*"
  - "*user_genotype_def.h .c*"
  - "*user_genotype.h .c*"
  - "*user_phenotype.h .c*"
  - "*user_encoding.h .c*"
  - "*user_operator.h .c*"
  - "*user_globalstructure.h .c*"
  - "*user_problemdata.h .c*"
  - "*engine.c*"
- "Utilities"
  - "*cte.h*"
  - "*environment.h .c*"
  - "*errors.h .c*"
  - "*configuration.h .c*"
  - "*nrand.h .c*"
  - "*fitnessdb.h .c*"

Files in each module:

user_gene_def.h

typedef:

TYPE_GENE_USER_DEF
A user defined *Gene* data type definition.

TYPE_LIMIT_USER_DEF
A user defined structure with information about the domain or range of values valid in TYPE_GENE_USER_DEF.

---

user_gene.h

procedure interfaces:

code gene_create_udef(TYPE_GENE*, const Configuration*)
Creates an empty gene, memory should be assigned to dynamic structures. The usage of this function is paired with gene_free_udef(), which frees the assigned memory. A control code as defined in "*errors.h*" is returned.

code gene_free_udef(TYPE_GENE*)
Frees the memory assigned to a gene.

code gene_initiate_udef(TYPE_GENE*, const Limit*, Nrand*, const Configuration*)
Randomly initiates a gene, the values should be limited to the domain of the gene as indicated by the structure Limit.

code gene_initiateFromFile_udef(TYPE_GENE*, FILE*, const Configuration*)
Initiates a gene from a file.

code gene_writeToFile_udef(TYPE_GENE*, FILE*, const Configuration*)
Writes a gene to a file, using the same format as in gene_initiateFromFile_udef.

int gene_display_udef(TYPE_GENE*, char*)
Generates a ready-to-print string representation of the gene. Returns the length of the corresponding string.

code gene_clone_udef(TYPE_GENE*, const TYPE_GENE*)
Copies the data from one gene to another, both source and target must already been created with gene_create_udef previously.

code range_read_udef(Limit*, const Configuration*)
Creates and loads the data structure Limit, which defines the domain and valid range of values for TYPE_GENE. This procedure can obtain a filename from the configuration and read the required data from a file.

code range_free_udef(Limit*, const int)
Frees the memory assigned to the data structure Limit.

---

gene.h

typedef:

TYPE_GENE
Generic definition of *Gene*, implemented as an union of different genes.

Limit
Generic definition of domain or range for values in TYPE_GENE.

code FUN_range_read(Limit*, const Configuration*)
Abstract definition of a function to fill the domain data structure with values taken from a file. The file's name is taken from the configuration.

code FUN_range_free(Limit*, const int)
Abstract definition of a function to free the assigned memory to the data structure Limit.

code FUN_gene_create(TYPE_GENE*, const Configuration*)
Abstract definition of a function to create and assign memory to the data structure TYPE_GENE. The usage of this type of function should be paired with call to FUN_gene_free.

code FUN_gene_free(TYPE_GENE*)
Abstract definition of function to free the assigned memory to the data structure TYPE_GENE.
code FUN_gene_clone(TYPE_GENE*, const TYPE_GENE*)
Abstract definition of function that replicates a TYPE_GENE to another. Both pointers must be created previously.

code FUN_gene_initiate(TYPE_GENE*, const Limit*, Nrand*, const Configuration*)
Abstract definition of function to initiate the data structure TYPE_GENE with values inside the domain Limit.

code FUN_gene_initiateFromFile(TYPE_GENE*, FILE*, const Configuration*)
Abstract definition of function to initiate the data structure TYPE_GENE with data taken from a file.

code FUN_gene_writeToFile(TYPE_GENE*, FILE*, const Configuration*)
Abstract definition of function to store the data structure TYPE_GENE in a file, using the same format as in FUN_gene_initiateFromFile.

int FUN_gene_display(TYPE_GENE*, char*)
Abstract definition of function that formats and creates a string representation for TYPE_GENE. Returns the length of the string.

procedure interfaces:

code limit_clone(Limit*, const Limit*)
Copies a data structure Limit to another one.

---

genotype.h

typedef:

Genotype
Generic definition of genotype, implemented as an union between list of TYPE_GENE and Genotype_USER_DEF.

GeneDomain
As genotype is a collection of TYPE_GENE, GeneDomain is a collection of Limit. It encapsulates the domain of each gene in the different position of the genotype.

code FUN_range_initiateDomain(GeneDomain*, Configuration*)
Abstract definition of function that initiates the data structure GeneDomain, for example by reading a file.

code FUN_range_freeDomain(GeneDomain*)
Abstract definition of function that frees the assigned memory to the data structure GeneDomain.

code FUN_range_getAt(const GeneDomain*, Limit*, const int)
Abstract definition of function that returns a clone of the inner structure Limit at a given position of GeneDomain.

code FUN_range_getReferenceAt(const GeneDomain*, Limit**, const int)
Abstract defintion of function that returns a reference to the inner data structure Limit at a given position of GeneDomain.

code FUN_range_setAt(GeneDomain*, const Limit*, const int)
Abstract definition of function that changes the range at a certain position of GeneDomain, with a new data structure of type Limit.

code FUN_genotype_create(Genotype*, const Configuration*)
Abstract definition of function that creates and allocates memory for data structure of type Genotype. Calls to this type of function should be paired with calls to the type FUN_genotype_free.

code FUN_genotype_free(Genotype*, const Configuration*)
Abstract definition of function that frees the assigned memory to data structure Genotype.

code FUN_genotype_clone(Genotype*, const Genotype*,const Configuration*)
Abstract definition of function that replicates the data structure Genotype, both pointers should be created previously.

code FUN_genotype_initiate(Genotype*, const GeneDomain*,Nrand*, const Configuration*)
Abstract definition of function that initiates the data structure Genotype randomly.

code FUN_genotype_initiateFromFile(Genotype*, FILE*, const GeneDomain* gdom, Nrand* NR, const Configuration*)
Abstract definition of function that initiates the data structure Genotype from a file.

code FUN_genotype_writeToFile(Genotype*, FILE*, const Configuration*)
Abstract definition of function that stores an instance of the data structure Genotype into a file, using the same format as in FUN_genotype_initiateFromFile.

int FUN_genotype_sizeOf(const Genotype*)
Abstract definition of function that returns the length of a Genotype.

code FUN_genotype_getAt(const Genotype*, TYPE_GENE*, const int, const Configuration*)
Abstract definition of function that obtains a clone of the inner structure TYPE_GENE from a given position of a Genotype.

code FUN_genotype_getReferenceAt(const Genotype*, TYPE_GENE**, const int)
Abstract definition of function that obtains a reference of the inner structure TYPE_GENE from a given position of a Genotype.

code FUN_genotype_setAt(Genotype*, const TYPE_GENE*, const int, const Configuration*)
Abstract definition of function that replaces the TYPE_GENE at a given position of Genotype.

int FUN_genotype_display(const Genotype*, char*, const Configuration*)
Abstract definition of function that creates a string representation of a Genotype, the length of the string is returned.

---

population.h

typedef:

Population
Data structure that represents *Population*, implemented as a list of Genotype.

procedure interfaces:

code pop_create(Population*, const int, const Configuration*)
Creates and assigns memory for a Population, takes the population's capacity as an input. This procedure should be used in pair with pop_free.

code pop_free(Population*, const Configuration*)
Frees the assigned memory from a population.

code pop_initiate(Population*, const int, const GeneDomain*, Nrand*, const Configuration*)
A population of a given size is initiated randomly, with values inside the specific domain.

code pop_initiateFromFile(Population*, const char*, const int, const GeneDomain*, Nrand*, const Configuration*)
A population of a given size is initiated from a file, if the file doesn't exist or doesn't contain enough individuals a random initialisation is used to complete the missing members.

code pop_writeToFile(Population*, const char*, const Configuration*)
Saves the population into a file, using the same format as in pop_initiateFromFile.

int pop_sizeOf(const Population*)
Returns the current size of the population.

int pop_capacityOf(const Population*)
Returns the total capacity of the population.

code pop_getAt(const Population*, Genotype*, const int, const Configuration*)
Gets a clone of the Genotype at a certain position of the population.

code pop_getReferenceAt(const Population*, Genotype**, const int)
Gets a reference to the Genotype at a certain position of the population.

code pop_setAt(Population *, const Genotype *, const int, const Configuration*)

Replaces the Genotype at a certain position of the population.

code pop_pushGenotype(Population*, const Genotype*, const Configuration*)
Adds a Genotype at the last position of the population.

code pop_popGenotype(Population*, Genotype*, const Configuration*)
Removes a Genotype at the last position of the population.

int pop_display(const Population*, char*, const Configuration *)
Creates a string representation for the population, and returns the length of the string.

---

selection.h

typedef:

code FUN_selection(Population*, const Population*, FitnessDB*, Nrand*, Configuration*)
Abstract definition of the selection operator. Takes a population and generates a chosen population according to parameters in the Configuration and the fitness evaluation of the individuals. Like all other functions, it returns a code as defined in "*errors.h*" .

---

crossover.h

typedef:

code FUN_crossover(Population*, const Population*, const GeneDomain* range, Nrand*, const Configuration*)
Abstract definition of crossover operator. Takes a parent population and generates an offspring population according to the parameters in Configuration. Like all other functions, it returns a code as defined in "*errors.h*" .

---

crossover.h

typedef:

code FUN_mutation(Population*, const Population*, const GeneDomain*, Nrand*, const Configuration*)
Abstract definition of mutation operator. Takes a population and generates a mutated population according to the parameters in Configuration. Once again, it returns a control code as defined in "*errors.h*" .

---

user_phenotype.h

typedef:

Phenotype
User defined data structure to represent a *Phenotype* or in other words, a solution of the problem to optimise.

procedure interfaces:

code phenotype_create(Phenotype*, const Configuration*)
Create and assigns memory for the data structure Phenotype. The usage of this function should be paired with phenotype_free. A control code is returned as defined in "*errors.h*" .

code phenotype_free(Phenotype*, const Configuration*)
Frees the memory assigned for the data structure Phenotype.

int phenotype_display(const Phenotype*, char*, const Configuration*)
Creates a string representation of the Phenotype and returns the length of such string.

---

user_genotype_def.h

typedef:

Genotype_USER_DEF
User defined Genotype, only necessary if the list of implemented Genotypes is not suitable for the problem.

GeneDomain_USER_DEF
When a new Genotype_USER_DEF is defined, the associated domain is defined here.

---

user_genotype.h

procedure interfaces:

FUN_range_initiateDomain range_initiateDomain_udef
FUN_range_freeDomain range_freeDomain_udef
FUN_range_getAt range_getAt_udef
FUN_range_getReferenceAt range_getReferenceAt_udef
FUN_range_setAt range_setAt_udef
FUN_genotype_create genotype_create_udef
FUN_genotype_initiate genotype_initiate_udef
FUN_genotype_initiateFromFile genotype_initiateFromFile_udef
FUN_genotype_writeToFile genotype_writeToFile_udef
FUN_genotype_free genotype_free_udef
FUN_genotype_sizeOf genotype_sizeOf_udef
FUN_genotype_getAt genotype_getAt_udef
FUN_genotype_getReferenceAt genotype_getReferenceAt_udef
FUN_genotype_setAt genotype_setAt_udef
FUN_genotype_clone genotype_clone_udef
FUN_genotype_display genotype_display_udef

Refer to "*genotype.h*" for the definition of the abstract function types. This series of functions should be implemented by the user only if the predefined Genotype is not suitable for the problem.

---

user_encoding.h

procedure interfaces:

code encode(Genotype*, const Phenotype*, const Configuration*)
Mapping function that transforms a Phenotype into a Genotype, its implementation is problem specific and should be provided by the user.

code decode(Phenotype*, const Genotype*, const Configuration*)
Mapping function that transforms Genotype back into a Phenotype, its implementation is problem specific and should be provided by the user.

float user_fitness(const Phenotype*, const Configuration*,FitnessDB*)
Function that determines for each Phenotype a fitness value, this function is strongly related with the objective function of the problem. And is provided by the user.

---

user_operator.h

procedure interfaces:

FUN_selection selection_udef
FUN_crossover crossover_udef
FUN_mutation mutation_udef

The definition of these function types can be found in "*selection.h*", "*crossover.h*", and "*mutation.h*" . These functions are here to extend the list of operators with user defined new operators, they are not required when using already implemented operators.

---

user_globalstructure.h

typedef

USER_STRUCT
User defined structure, for general data storage. This data structure will be available in any part of the library, including user defined operators, encoding/decoding functions, fitness evaluation, among others.

procedure interfaces:

void ugs_initiate(USER_STRUCT *ugs, ProbData *data)
Creates and initiates the data structure USER_STRUCT, allocating memory to inner structures. The usage of this function is paired with ugs_free.

void ugs_free(USER_STRUCT *ugs, Configuration* conf)
Frees the assigned memory from the data structure USER_STRUCT.

---

user_problemdata.h

typedef

ProbData
User defined data structure to store problem related data. This component is designed to read problem instance's data from file and store them into ProbData.

procedure interfaces:

code prdata_create(ProbData*)
Creates and assigns memory for the data structure ProbData, this function should be used in pair with prdata_free.

code prdata_free(ProbData*)
Frees the allocated memory.

code prdata_read(ProbData*, char*)
Read from a file and fills the data strucutre ProbData, the structure must be created with prdata_create previously.

void prdata_display(const ProbData*)
Displays on screen the data in ProbData.

---

environment.h

typedef

Environment
Data structure that contains the mapping of function pointer to particular implementation to use in the engine.

procedure interfaces:

void environment_initiate(Environment*, Configuration*)
Initiates the data structure Environment with the correct instances for each function pointers to use, by reading the configuration. The function environment_free should be called to free the assigned memory before exit.

void environment_free(Environment*, Configuration*)
Frees the memory assigned to the data structure Environment.

---

errors.h

typedef

enum {OK=0, ERROR=1} code
General purpose control code used by all function of the library.

procedure interfaces:

code initCode(code*)
Initiates the code with a default value (TRUE).

code setError(code*)
Set the code to ERROR status.

code setOk(code*) Set the code to OK status.

code joinCode(const code, const code)
Applies the Boolean OR operation over the two codes and returns the result.

boolean isOk(const code)
Check if the code is in OK status.

---

configuration.h

typedef

Configuration
Data structure (a dictionary) that encapsulates the configuration parameter to value mapping.

procedure interfaces:

code conf_create(Configuration*)
Creates and assigns memory for the data structure Configuration. Should be paired with conf_free.

code conf_free(Configuration*)
Frees the assigned memory from the data structure.

code conf_read(Configuration*, char*, char*)
Reads from a file and fills the data structure, which must be already created with conf_create.

int conf_sizeOf(const Configuration*)
Returns the size of the dictionary Configuration.

code conf_getInt(const Configuration*, const char*, int*)
Consults an integer valued parameter.

code conf_setInt(Configuration*, const char*, int)
Changes the value of an integer valued parameter.

code conf_getFloat(const Configuration*, const char*, float*)
Consults a float valued parameter.

code conf_setFloat(Configuration*, const char*, float)
Changes the value of a float valued parameter.

code conf_getStr(const Configuration*, const char*, char*)
Consults a string valued parameter.

code conf_getProbData(const Configuration*, ProbData**)
Changes the value of a string valued parameter.

void conf_display(const Configuration*)
Prints on screen the data of the data structure Configuration.

---

nrand.h

typedef

Nrand
Data structure to store the status and parameters of the pseudo-random generator.

procedure interfaces:

code nrand_initiate(Nrand *, unsigned long mt_seed, mpz_t a, const unsigned long c, const unsigned long m2exp, mpz_t gmp_seed)
Creates and initiates the generator with the given random seed.

code nrand_initiateFromFile(Nrand *, char *)
Creates and initiates the generator, then loads a random seed from a specific file.

void nrand_free(Nrand *)
Frees the assigned memory from the data structure.

double nrand_uni(Nrand *)
Returns a decimal according to a uniform distribution with parameter $[0, 1]$.

float nrand_floatRange(const double min, const double max, Nrand*)
Returns a decimal according to a uniform distribution with parameter $[a, b]$.

int nrand_intRange(const int min, const int max, Nrand*)
Returns an integer according to a uniform distribution with parameter $[a, b]$.

boolean nrand_flip(const double p, Nrand *)
Returns a Boolean according to a Bernoulli distribution with parameter p.

int nrand_intGeo(const double p, Nrand *)
Returns an integer according to a geometric distribution with parameter p.

int nrand_intGeoAlpha(const double alpha, Nrand *)
Returns an integer according to a geometric distribution with parameter p. This version is designed for repeated calls with the same parameter, then the input *alpha* is $\frac{1}{log(1-p)}$

void nrand_MPZUni(Nrand *, const mpz_t,mpz_t)
Returns a big integer of type mpz_t (a GMP data type) according to a uniform distribution with parameter $[0, 1]$. The underlying generator in use is a linear congruent generator coming from the library GMP.

---

fitnessdb.h

typedef

FitnessDB
Data structure to be used as the fitness database.

procedure interfaces:

code fitdb_create(FitnessDB*, const Configuration*)
Creates an empty data structure FitnessDB and assigns memory if needed. This function should be used in pair with fitdb_free.

code fitdb_free(FitnessDB*)
Frees the memory assigned to the data structure.

float fitdb_get(Genotype*, Phenotype*, FitnessDB*, const Configuration*)
Given a Genotype, tries to determine the corresponding fitness value by first searching in the cache, and then if fails, it will decode and evaluate the phenotype using the fitness function. The resulting phenotype is the decode function is called will then be returned too.

---

cte.h

List of default parameter values, which are used when the parameter is missing.

DEFAULT_TYPE_GENE	INDEX_BOOL
DEFAULT_GENOTYPE_LEN	10
DEFAULT_POPULATION_SIZE	100
DEFAULT_CROSSOVER_RATE	0.7
DEFAULT_MUTATION_RATE	0.01
DEFAULT_GENERATIONS	150
DEFAULT_GENEDOMAIN_BOOL_FILE	"*range_bool.dat*"
DEFAULT_GENEDOMAIN_INT_FILE	"*range_int.dat*"
DEFAULT_GENEDOMAIN_FLOAT_FILE	"*range_float.dat*"
DEFAULT_GENEDOMAIN_USER_FILE	"*range_user.dat*"
RANDOM_FILE	"*nr.dat*"
CONFIGURATION_FILE	"*config.dat*"
DEFAULT_GENE_TRUE_RATE	0.5
DEFAULT_GENE_INT_MIN	0
DEFAULT_GENE_INT_MAX	10
DEFAULT_GENE_FLOAT_MIN	0.0
DEFAULT_GENE_FLOAT_MAX	1.0
DEFAULT_CROSSOVER_WEIGHT	0.6
DEFAULT_SELECTION_SIZE	SELECTION_SIZE/2

Other parameters are defined in "*cte.h*", advanced users who want to implement user defined operators should consult that file for the names of the different parameters stored in the *Configuration*.

B.8 Format of the configuration file

The name of the configuration file is "*config.dat*", parameters are stored in each line in the form of "Parameter = Value". Any number of space, tab or enter characters can be inserted in the line, comments can also be inserted anywhere with the comment mark #. Whenever the comment mark is detected, the remaining line is truncated.

Parameter	Description	Value range
TYPE_GENE	type of *Gene*	0 Boolean (default) 1 integer 2 float 3 user defined
TYPE_GENOTYPE	type of *Genotype*	0 predefined, list of *Gene* 1 user defined
SELECTION_OPERATOR	selection op. to use	0 roulette 1 tournament 2 ranking 3 user defined
CROSSOVER_OPERATOR	crossover op. to use	0 one point 1 two points 2 uniform one point 3 uniform two points 4 weighted one point 5 user defined
MUTATION_OPERATOR	mutation op. to use	0 uniform 1 inversion (Boolean genes only) 2 use defined
GENERATIONS	number of generations to run	natural number
POPULATION_SIZE	size of the population	natural number
SELECTION_SIZE	size of mating pool	natural number not larger than population size
CROSSOVER_RATE	crossover rate	decimal in $[0, 1]$
MUTATION_RATE	mutation rate	decimal in $[0, 1]$
GENOTYPE_LEN	number of genes in each genotype	natural number (for predefined genotype only)
TOURNAMENT_SIZE	size of the list of tournament	natural number (for tournament selection only)
CROSSOVER_WEIGHT	the weight of weighted crossover	decimal in $[0, 1]$ (for weighted crossover only)
BOOLEAN_GENE_RANGE_FILE	name of domain file	path and name (for Boolean gene only)
INTEGER_GENE_RANGE_FILE	name of domain file	path and name (for integer gene only)
FLOAT_GENE_RANGE_FILE	name of domain file	path and name (for decimal gene only)
USER_GENE_RANGE_FILE	name of domain file	path and name (for user defined gene only)

Redundant parameters like those designed for operators that are not chosen, can be left in the configuration file commented or uncommented. Any parameter-value pair will be read and stored into the *Configuration* data structure, and it doesn't matters if there are redundant pairs, as long as the required are present. If a parameter is missing, default value for that parameter will be used instead, the list of default values can be found in "*cte.h*" .

B.9 User's Manual

To solve a problem using GGEngine, first of all it is necessary to model the problem. In other words, it requires a suitable coding, a fitness function and the different genetic operators. In this section, we present a brief guideline together with some simple examples.

The first step: a type of *Gene* must be chosen and setup in the configuration file "*config.dat*", the associated parameter is TYPE_GENE:

- 0 Boolean gene
- 1 integer gene
- 2 decimal gene
- 3 user defined gene

If 3 is chosen, then the user has to define some data structures and implement a list of procedures as specified in the interface. Beginning with "*user_gene_def.h*" where TYPE_GENE_USER_DEF and TYPE_LIMIT_USER_DEF are defined. The former is the data structure for the *Gene* while the latter is a data structure associated with the domain of the gene.

Example I.1.a:

```
typedef char TYPE_GENE_USER_DEF;          // defined as a char here

typedef enum{e1='A', e2='B', e3='C'} AUX;

typedef struct{
        AUX domain;          // defines a domain here
        } TYPE_LIMIT_USER_DEF;
```

Example II.1.a:

```
typedef struct LIST_NODE{
        char a;
        chat b;
        };

typedef struct
        LIST_NODE *dat;
        } TYPE_GENE_USER_DEF;          // or some complicated structure

typedef struct{
        int min_number_nodes;
        int max_number_nodes;
        AUX2 domain_field;          // can be as strange as the user needs
        } TYPE_LIMIT_USER_DEF;
```

Then in the file "*user_gene.c*" each of the following procedure should be implemented according to the interface. Sub-procedures can be added there too.

- gene_create_udef
- gene_free_udef
- gene_initiate_udef
- gene_initiateFromFile_udef

- gene_clone_udef
- gene_writeToFile_udef
- gene_display_udef
- range_read_udef
- range_free_udef

Among the input parameters of those procedures, we have TYPE_GENE and Limit:

```
typedef union{
    ...
    TYPE_GENE_USER_DEF TYPE_GENE_UDEF;
    } TYPE_GENE;

typedef union{
    ...
    TYPE_LIMIT_USER_DEF TYPE_LIMIT_UDEF;
    } Limit;
```

Some of the procedures also have "Configuration * conf" and/or "Nrand * NR" as input, the former is for configuration parameter access, some important parameters are:

- PARAM_GENEDOMAIN_USER_FILE
- DEFAULT_GENEDOMAiN_USER_FILE
- and all other parameter in "*cte.h*" that has the prefix PARAM_ o DEFAULT_

The user can access those data with conf_getStr (and other get series procedures, check "*configuration.h*" in API for a complete list).

```
    code c;
    c=conf_getStr(conf, PARAM_GENEDOMAIN_USER_FILE, &filename);
    c=conf_getStr(conf, DEFAULT_GENEDOMAIN_USER_FILE, &filename);
```

The returned control code is defined in "*errors.h*", to indicate if the operation was successful, in brief the function isOk(code) will extract the status of the code.
About the input parameter *Nrand*, it is used to access the random number generator (more details can be found in "*nrand.h*" from the API), a brief list of useful procedures:

- nrand_uni
- nrand_intRange
- nrand_floatRange
- nrand_flip

These functions are useful for the implementation of gene_initiate_udef.

Example I.1.b (following the example I.1.a)

```
code gene_create_udef(TYPE_GENE* g, const Configuration* conf){
    // no dynamic memory needed in this case
    code c; initCode(&c);
    return c; //a long way to return OK
    }

code gene_free_udef(TYPE_GENE* g){
    return OK; //nothing to free in this case
    }
code gene_initiate_udef(TYPE_GENE* g, const Limit* lim, Nrand* NR, const Configuration* conf){
```

```
        int n;
        n=nrand_intRange(0,2,NR); //o other sampling using information in lim
        switch(n){
                case 0:
                        g→TYPE_GENE_UDEF = e1;
                        break;
                case 1:
                        g→TYPE_GENE_UDEF = e2;
                        break;
                case 2:
                        g→TYPE_GENE_UDEF = e3;
                        break;
                }
        return OK; // or ERROR if something when wrong
        }

code gene_initiateFromFile_udef(TYPE_GENE* g, FILE* fp, const Configuration* conf){
        code c; initCode(&c);
        fscanf(fp, "%c\n", &(g→TYPE_GENE_UDEF) ); // may verify integrity and change c here
        return c;
        }

code gene_clone_udef TYPE_GENE* dest, const TYPE_GENE* orig){
        dest→TYPE_GENE_UDEF = orig→TYPE_GENE_UDEF;
        return OK;
        }

code gene_writeToFile_udef(TYPE_GENE* g, FILE* fp, const Configuration* conf){
        fprintf(fp, "%c\n", g→ TYPE_GENE_UDEF);
        return OK;
        }

int gene_display_udef(TYPE_GENE* g, char* result){
        result[0]=g→ TYPE_GENE_UDEF; // knowing that this is a char
        result[1]='\0';
        return 1; // length of the string
        }

code range_read_udef(Limit* lim, const Configuration* conf){
        // in this case TYPE_LIMIT_USER_DEF contains an enumeration
        return OK; // no need to read file
        }

code range_free_udef(Limit* lim, const int size){
        return OK; // nothing to free
        }
```

The second step: definition of *Phenotype*, *Genotype* together with the associated encode/decode and fitness function.

- In "*user_phenotype.h*" define the data structure *Phenotype*, which represents the solutions to the problem to solve. All the procedures in the same file should also be implemented.
- If the predefined genotype is not suitable, define in "*user_genotype_def.h*" the data structures Genotype_USER_DEF and GeneDomain_USER_DEF. And set TYPE_GENOTYPE = 1 in "*config.dat*"

- Then implement in "*user_encoding.c*" the following functions
  - code decode(Phenotype*;const Genotype*,const Configuration*)
  - code encode(Genotype*,const Phenotype*,const Configuration*)
  - float user_fitness(const Phenotype*,const Configuration*,FitnessDB*)

Example I.2.a
For the problem of maximising the identity function $f(x) = x$, with x a natural number. We will use a ternary coding, that is to encode natural numbers in base three. Then with a genotype of length l, we can represent the range $[0, 3^l - 1]$. To implement this model, we can use the examples I.1.a and I.1.b as the first steps, then as second step we do as follows.

In "*user_phenotype.h*":
```
typedef int Phenotype;
```

In "*user_phenotype.c*":
```
code phenotype_create(Phenotype* p, const Configuration* conf){
        return OK; // no dynamic memory in this case
        }

code phenotype_free(Phenotype* p, const Configuration* conf){
        return OK; // nothing to free neither
        }

int phenotype_display(const Phenotype* p, char* str, const Configuration* conf){
        return sprintf(str,"%i\n",p);
        }
```

Then in "*user_encoding.c*":
```
code decode(Phenotype* phen,const Genotype* geno, const Configuration* conf){
        // note that at this point, the user should know the type of Gene in use
        int i,l;
        TYPE_GENE *gen;
        conf_getInt(conf,PRARM_GENOTYPE_LEN,&l);
        (*phen)=0;
        for(i=0;i<l;i++){
                GLOBAL_env.genotype_getReferenceAt(geno,&gen,i,conf); // reference to the original
                switch(gen→TYPE_GENE_UDEF){
                        case e1: break;
                        case e2: (*phen)+=pow(3,i);break;
                        case e3: (*phen)+=2*pow(3,i);break;
                        }
                }
        return OK;
        }

code encode(Genotype* geno,const Phenotype* phen, const Configuration* conf){
        int i, l, value, remain;
        TYPE_GENE *gen;
        conf_getInt(conf,PRARM_GENOTYPE_LEN,&l);

        value=(*phen);
        for(i=l-1;i>=0;i–){
                GLOBAL_env.genotype_getReferenceAt(geno,&gen,i,conf);
                remain=pow(3,i);
                switch(value/remain){
```

```
                case 0:
                        gen→TYPE_GENE_UDEF = e1;
                        break;
                case 1:
                        gen→TYPE_GENE_UDEF = e2;
                        break;
                case 2:
                        gen→TYPE_GENE_UDEF = e3;
                        break;
                }
        value-=remain;
        }
return OK;
}

float user_fitness(const Phenotype* phen, const Configuration* conf, FitnessDB *db){
        return (float)(*phen);//f(x)=x
        }
```

Fitness scaling is implemented as an option, details on how to active it can be found in "*user_operator.h .c*" .

The third step: And as the last step, genetic operators to apply and the stopping condition have to be setup in the configuration file "*config.dat*" . See Section B.8 for the details. When all the steps are completed, the engine is ready for execution, the result and some basic statistic is output on screen. If further run with different random seeds is required, the user should change the seed in the seed file "*nr.dat*", only the first line is read in the latter.

Appendix C

Problem's input file format

The solver reads "*graph.dat*" as input, in this file all the necessary data of our network design problem is obtained. The file is written with a tagged structure design, and is divided into four sections:

- $< G >\ n =?\ d =?\ fill\_cost =?\ < /G >$
- $< T >\ i =?\ c =?\ s =?\ < /T >$
- $< A >\ f =?\ t =?\ < T >\ i =?\ < /T >< /A >$
- $< D >\ f =?\ t =?\ k =?\ s =?\ h =?\ < /D >$

The first section is one single line with the $< G >$ tag containing general information about the graph, being the number of $n$odes, $d$emands and the filling cost for the fictional link (with unlimited capacity) for each edge.
The second section is a sequence of $< T >$ tags, each one of them is a declaration of a type of link option. Each entry contains the $i$dentification of the link, $c$ost of instalation, and $s$ize of the maximum capacity. Later on, link options will be referred to, by using the assigned identification.
The third section of the file contains a sequence of $< A >$ tags, each one of them represents an edge. It contains information about $f$orm which node to which other is the edge placed, and a list of possible link options on a list of $< T >$ tags, the latter only contains the $i$dentification of the link type.
And finally in the last section, a sequence of $< D >$ tags are placed with information about the different demand pairs. In this case, each tag tells $f$rom which origin to which destination is the demand required, together with the required throughput's $s$ize for the h-limited k-path.

Appendix D

Test suite: basic cases


```
# case A
#
# a complete graph with 6 nodes and 6 s-t pairs
# solution space size 788050640
# optimal cost=43

<G>n=6 d=6 fill_cost=161</G>

<T> i=1 c=1 s=1</T>
<T> i=2 c=2 s=2</T>
<T> i=3 c=3 s=3</T>
<T> i=4 c=4 s=4</T>
<T> i=5 c=5 s=5</T>
<T> i=6 c=6 s=6</T>
<T> i=7 c=7 s=7</T>
<T> i=8 c=8 s=8</T>
<T> i=9 c=9 s=9</T>
<T> i=10 c=10 s=10</T>

<A> f=1 t=2           <T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=1 t=3                     <T>i=3</T>          <T>i=5</T>          <T>i=7</T>                    <T>i=10</T></A>
```

```
<A> f=1 t=4 <T>i=1</T><T>i=2</T>                    <T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=5 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T>          <T>i=7</T>                    <T>i=10</T></A>
<A> f=1 t=6           <T>i=2</T><T>i=3</T>          <T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=3 <T>i=1</T><T>i=2</T><T>i=3</T>          <T>i=5</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=4           <T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T>          <T>i=8</T>          <T>i=10</T></A>
<A> f=2 t=5                               <T>i=4</T><T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=2 t=6 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=4 <T>i=1</T><T>i=2</T>          <T>i=4</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=5 <T>i=1</T>                    <T>i=4</T><T>i=5</T>                    <T>i=8</T>          <T>i=10</T></A>
<A> f=3 t=6 <T>i=1</T>          <T>i=3</T>          <T>i=5</T>                                        <T>i=10</T></A>
<A> f=4 t=5                               <T>i=5</T>                                        <T>i=10</T></A>
<A> f=4 t=6 <T>i=1</T><T>i=2</T>          <T>i=4</T>          <T>i=6</T>                    <T>i=10</T></A>
<A> f=5 t=6 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T>          <T>i=10</T></A>

<D> f=1 t=2 k=2 s=5 h=4</D>
<D> f=1 t=5 k=2 s=1 h=5</D>
<D> f=3 t=5 k=1 s=2 h=1</D># H=1 => K=1
<D> f=3 t=6 k=2 s=3 h=3</D>
<D> f=4 t=6 k=3 s=2 h=3</D>
<D> f=5 t=6 k=1 s=3 h=3</D>

######################################################################
# case B
#
# a complete graph with 6 nodes and 6 s-t pairs
# solution space size 327688400
# optimal cost=55

<G>n=6 d=6 fill_cost=161</G>

<T> i=1 c=1 s=1</T>
<T> i=2 c=2 s=2</T>
<T> i=3 c=3 s=3</T>
<T> i=4 c=4 s=4</T>
<T> i=5 c=5 s=5</T>
<T> i=6 c=6 s=6</T>
```

```
<T> i=7 c=7 s=7</T>
<T> i=8 c=8 s=8</T>
<T> i=9 c=9 s=9</T>
<T> i=10 c=10 s=10</T>

<A> f=1 t=2 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T>                                                    <T>i=9</T><T>i=10</T></A>
<A> f=1 t=3 <T>i=1</T>          <T>i=3</T>          <T>i=5</T><T>i=6</T>                               <T>i=9</T><T>i=10</T></A>
<A> f=1 t=4 <T>i=1</T>          <T>i=3</T>                              <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=5                     <T>i=3</T>          <T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T>          <T>i=10</T></A>
<A> f=1 t=6 <T>i=1</T>          <T>i=3</T>                                                         <T>i=9</T><T>i=10</T></A>
<A> f=2 t=3           <T>i=2</T><T>i=3</T>          <T>i=5</T><T>i=6</T><T>i=7</T>                     <T>i=10</T></A>
<A> f=2 t=4 <T>i=1</T>                    <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=2 t=5 <T>i=1</T><T>i=2</T><T>i=3</T>          <T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=6 <T>i=1</T><T>i=2</T><T>i=3</T>                              <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=4 <T>i=1</T><T>i=2</T>          <T>i=4</T>                    <T>i=7</T>                     <T>i=10</T></A>
<A> f=3 t=5 <T>i=1</T>                              <T>i=5</T>                    <T>i=8</T>          <T>i=10</T></A>
<A> f=3 t=6                     <T>i=3</T><T>i=4</T><T>i=5</T>                                         <T>i=10</T></A>
<A> f=4 t=5                     <T>i=3</T><T>i=4</T>                              <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=4 t=6 <T>i=1</T>          <T>i=3</T><T>i=4</T><T>i=5</T><T>i=6</T>                               <T>i=10</T></A>
<A> f=5 t=6 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>

<D> f=1 t=3 k=2 s=3 h=4</D>
<D> f=1 t=5 k=2 s=1 h=5</D>
<D> f=2 t=3 k=1 s=4 h=1</D># H=1 => K=1
<D> f=2 t=4 k=3 s=2 h=3</D>
<D> f=3 t=5 k=3 s=5 h=5</D>
<D> f=4 t=5 k=1 s=1 h=2</D>

######################################################################
# case C
#
# a complete graph with 6 nodes and 6 s-t pairs
# solution space size 430941160
# optimal cost=70

<G>n=6 d=5 fill_cost=161</G>
```

```
<T> i=1 c=1 s=1</T>
<T> i=2 c=2 s=2</T>
<T> i=3 c=3 s=3</T>
<T> i=4 c=4 s=4</T>
<T> i=5 c=5 s=5</T>
<T> i=6 c=6 s=6</T>
<T> i=7 c=7 s=7</T>
<T> i=8 c=8 s=8</T>
<T> i=9 c=9 s=9</T>
<T> i=10 c=10 s=10</T>

<A> f=1 t=2 <T>i=1</T><T>i=2</T><T>i=3</T>          <T>i=5</T><T>i=6</T>                    <T>i=9</T><T>i=10</T></A>
<A> f=1 t=3 <T>i=1</T><T>i=2</T><T>i=3</T>          <T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T>          <T>i=10</T></A>
<A> f=1 t=4 <T>i=1</T>                    <T>i=4</T>                                    <T>i=9</T><T>i=10</T></A>
<A> f=1 t=5       </T><T>i=2</T>          <T>i=4</T><T>i=5</T>                    <T>i=8</T>          <T>i=10</T></A>
<A> f=1 t=6 <T>i=1</T>                              <T>i=5</T>                              <T>i=9</T><T>i=10</T></A>
<A> f=2 t=3       </T>                    <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=2 t=4       </T>          <T>i=3</T><T>i=4</T>          <T>i=6</T>          <T>i=8</T>          <T>i=10</T></A>
<A> f=2 t=5 <T>i=1</T>          <T>i=3</T>          <T>i=5</T><T>i=6</T>          <T>i=8</T>          <T>i=10</T></A>
<A> f=2 t=6 <T>i=1</T>                    <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T><T>i=8</T>          <T>i=10</T></A>
<A> f=3 t=4 <T>i=1</T>          <T>i=3</T><T>i=4</T>                                    <T>i=9</T><T>i=10</T></A>
<A> f=3 t=5 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=6 <T>i=1</T>                    <T>i=4</T>          <T>i=6</T><T>i=7</T><T>i=8</T>          <T>i=10</T></A>
<A> f=4 t=5 <T>i=1</T>                                        <T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=4 t=6 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=5 t=6 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>

<D> f=1 t=2 k=3 s=1 h=3</D>
<D> f=1 t=3 k=2 s=3 h=5</D>
<D> f=1 t=4 k=3 s=3 h=5</D>
<D> f=2 t=5 k=4 s=5 h=4</D>
<D> f=2 t=6 k=1 s=2 h=4</D>

#####################################################################
# case D
```

```
#
# a complete graph with 6 nodes and 6 s-t pairs
# solution space size 304538000
# optimal cost=142

<G>n=6 d=6 fill_cost=601</G>

<T> i=11 c=2 s=1</T>
<T> i=12 c=4 s=2</T>
<T> i=13 c=6 s=3</T>
<T> i=14 c=8 s=4</T>
<T> i=15 c=10 s=5</T>
<T> i=16 c=12 s=6</T>
<T> i=17 c=14 s=7</T>
<T> i=18 c=16 s=8</T>
<T> i=19 c=18 s=9</T>
<T> i=20 c=20 s=10</T>

<A> f=1 t=2 <T>i=11</T>           <T>i=13</T><T>i=14</T><T>i=15</T><T>i=16</T><T>i=17</T>           <T>i=19</T><T>i=20</T></A>
<A> f=1 t=3 <T>i=11</T>                      <T>i=14</T>                      <T>i=17</T>           <T>i=19</T><T>i=20</T></A>
<A> f=1 t=4            <T>i=12</T>           <T>i=14</T>           <T>i=16</T><T>i=17</T>           <T>i=19</T><T>i=20</T></A>
<A> f=1 t=5 <T>i=11</T>                      <T>i=14</T>                                 <T>i=18</T>           <T>i=20</T></A>
<A> f=1 t=6                                           <T>i=15</T><T>i=16</T>           <T>i=18</T>           <T>i=20</T></A>
<A> f=2 t=3 <T>i=11</T>                               <T>i=15</T><T>i=16</T>                                 <T>i=20</T></A>
<A> f=2 t=4            <T>i=12</T><T>i=13</T>                                          <T>i=18</T><T>i=19</T><T>i=20</T></A>
<A> f=2 t=5 <T>i=11</T>                      <T>i=14</T><T>i=15</T>                                           <T>i=20</T></A>
<A> f=2 t=6 <T>i=11</T><T>i=12</T>                                                                  <T>i=19</T><T>i=20</T></A>
<A> f=3 t=4                     <T>i=13</T><T>i=14</T>           <T>i=16</T>                                 <T>i=20</T></A>
<A> f=3 t=5 <T>i=11</T>           <T>i=13</T>                      <T>i=16</T><T>i=17</T>           <T>i=19</T><T>i=20</T></A>
<A> f=3 t=6 <T>i=11</T>                                                     <T>i=17</T>           <T>i=19</T><T>i=20</T></A>
<A> f=4 t=5            <T>i=12</T>           <T>i=14</T><T>i=15</T>           <T>i=17</T><T>i=18</T><T>i=19</T><T>i=20</T></A>
<A> f=4 t=6            <T>i=12</T>           <T>i=14</T><T>i=15</T>                                 <T>i=19</T><T>i=20</T></A>
<A> f=5 t=6            <T>i=12</T><T>i=13</T>           <T>i=15</T><T>i=16</T><T>i=17</T>           <T>i=19</T><T>i=20</T></A>

<D> f=1 t=2 k=2 s=2 h=5</D>
<D> f=1 t=4 k=5 s=2 h=3</D>
```

```
<D> f=1 t=6 k=2 s=1 h=2</D>
<D> f=2 t=6 k=2 s=1 h=5</D>
<D> f=3 t=5 k=4 s=3 h=4</D>
<D> f=4 t=5 k=2 s=5 h=4</D>

#####################################################################
# case E
#
# a complete graph with 7 nodes and 4 s-t pairs
# solution space size 714918000
# optimal cost=64

<G>n=7 d=4 fill_cost=211</G>

<T> i=1 c=1 s=1</T>
<T> i=2 c=2 s=2</T>
<T> i=3 c=3 s=3</T>
<T> i=4 c=4 s=4</T>
<T> i=5 c=5 s=5</T>
<T> i=6 c=6 s=6</T>
<T> i=7 c=7 s=7</T>
<T> i=8 c=8 s=8</T>
<T> i=9 c=9 s=9</T>
<T> i=10 c=10 s=10</T>

<A> f=1 t=2           <T>i=2</T>          <T>i=4</T><T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=1 t=3 <T>i=1</T>          <T>i=3</T><T>i=4</T>                                <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=4           <T>i=2</T>                                <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=5 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=6                                     <T>i=5</T>                      <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=7 <T>i=1</T><T>i=2</T><T>i=3</T>                                                <T>i=9</T><T>i=10</T></A>
<A> f=2 t=3 <T>i=1</T><T>i=2</T><T>i=3</T>          <T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=2 t=4 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=5           <T>i=2</T><T>i=3</T>                                      <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=6 <T>i=1</T><T>i=2</T>          <T>i=4</T>                                <T>i=9</T><T>i=10</T></A>
<A> f=2 t=7 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
```

<A> f=3 t=4 <T>i=1</T> <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T> <T>i=9</T><T>i=10</T></A>
<A> f=3 t=5 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T> <T>i=6</T> <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=6 <T>i=1</T> <T>i=3</T> <T>i=5</T> <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=7 <T>i=3</T> <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=4 t=5 <T>i=2</T><T>i=3</T> <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=4 t=6 <T>i=2</T> <T>i=4</T> <T>i=7</T> <T>i=9</T><T>i=10</T></A>
<A> f=4 t=7 <T>i=1</T> <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=5 t=6 <T>i=1</T> <T>i=3</T><T>i=4</T> <T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=5 t=7 <T>i=2</T><T>i=3</T><T>i=4</T> <T>i=6</T><T>i=7</T> <T>i=9</T><T>i=10</T></A>
<A> f=6 t=7 <T>i=1</T> <T>i=5</T><T>i=6</T> <T>i=8</T><T>i=9</T><T>i=10</T></A>

#if h=1 => k=1
<D> f=1 t=7 k=4 s=3 h=6</D>
<D> f=2 t=4 k=1 s=3 h=6</D>
<D> f=3 t=5 k=2 s=3 h=2</D>
<D> f=4 t=6 k=3 s=5 h=5</D>

##
case F
#
a complete graph with 7 nodes and 4 s-t pairs
solution space size 737256000
optimal cost=55

<G>n=7 d=4 fill_cost=211</G>

<T> i=1 c=1 s=1</T>
<T> i=2 c=2 s=2</T>
<T> i=3 c=3 s=3</T>
<T> i=4 c=4 s=4</T>
<T> i=5 c=5 s=5</T>
<T> i=6 c=6 s=6</T>
<T> i=7 c=7 s=7</T>
<T> i=8 c=8 s=8</T>
<T> i=9 c=9 s=9</T>
<T> i=10 c=10 s=10</T>

```
<A> f=1 t=2           <T>i=2</T>          <T>i=4</T><T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=1 t=3 <T>i=1</T>          <T>i=3</T><T>i=4</T>                                <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=4           <T>i=2</T>                                <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=5 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=6                                     <T>i=5</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=1 t=7 <T>i=1</T><T>i=2</T><T>i=3</T>                                              <T>i=9</T><T>i=10</T></A>
<A> f=2 t=3           <T>i=2</T><T>i=3</T>          <T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=2 t=4 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=5           <T>i=2</T><T>i=3</T>                                      <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=2 t=6 <T>i=1</T><T>i=2</T>          <T>i=4</T>                                    <T>i=9</T><T>i=10</T></A>
<A> f=2 t=7 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=3 t=4                               <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=3 t=5 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=6 <T>i=1</T>          <T>i=3</T>          <T>i=5</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=3 t=7                     <T>i=3</T>                          <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=4 t=5           <T>i=2</T><T>i=3</T>                                      <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=4 t=6           <T>i=2</T>          <T>i=4</T>                <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=4 t=7 <T>i=1</T>                                                <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=5 t=6 <T>i=1</T>          <T>i=3</T><T>i=4</T>          <T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A> f=5 t=7           <T>i=2</T>          <T>i=4</T>          <T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A> f=6 t=7 <T>i=1</T>          <T>i=3</T>          <T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>

#if h=1 => k=1
<D> f=1 t=7 k=3 s=3 h=6</D>
<D> f=2 t=4 k=3 s=4 h=3</D>
<D> f=2 t=5 k=1 s=1 h=2</D>
<D> f=4 t=6 k=2 s=5 h=5</D>

######################################################################
# case G
#
# a complete graph with 7 nodes and 3 s-t pairs
# solution space size 376856000
# optimal cost=43
```

```
<G>n=7 d=3 fill_cost=211</G>

<T> i=1 c=1 s=1</T>
<T> i=2 c=2 s=2</T>
<T> i=3 c=3 s=3</T>
<T> i=4 c=4 s=4</T>
<T> i=5 c=5 s=5</T>
<T> i=6 c=6 s=6</T>
<T> i=7 c=7 s=7</T>
<T> i=8 c=8 s=8</T>
<T> i=9 c=9 s=9</T>
<T> i=10 c=10 s=10</T>

<A>  f=1 t=2           <T>i=2</T>          <T>i=4</T><T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A>  f=1 t=3 <T>i=1</T>          <T>i=3</T><T>i=4</T>                              <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=1 t=4           <T>i=2</T>                              <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=1 t=5 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=1 t=6                                        <T>i=5</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=1 t=7 <T>i=1</T><T>i=2</T><T>i=3</T>                                                <T>i=9</T><T>i=10</T></A>
<A>  f=2 t=3 <T>i=1</T><T>i=2</T><T>i=3</T>          <T>i=5</T>          <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A>  f=2 t=4 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=2 t=5           <T>i=2</T><T>i=3</T>                                        <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=2 t=6 <T>i=1</T><T>i=2</T>          <T>i=4</T>                                        <T>i=9</T><T>i=10</T></A>
<A>  f=2 t=7 <T>i=1</T><T>i=2</T>          <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A>  f=3 t=4 <T>i=1</T>                    <T>i=4</T><T>i=5</T><T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A>  f=3 t=5 <T>i=1</T><T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=3 t=6 <T>i=1</T>          <T>i=3</T>          <T>i=5</T>                    <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=3 t=7                    <T>i=3</T>                              <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=4 t=5           <T>i=2</T><T>i=3</T>                                        <T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=4 t=6           <T>i=2</T>          <T>i=4</T>                    <T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A>  f=4 t=7 <T>i=1</T>                                                   <T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=5 t=6 <T>i=1</T>          <T>i=3</T><T>i=4</T>          <T>i=6</T><T>i=7</T><T>i=8</T><T>i=9</T><T>i=10</T></A>
<A>  f=5 t=7           <T>i=2</T><T>i=3</T><T>i=4</T>          <T>i=6</T><T>i=7</T>          <T>i=9</T><T>i=10</T></A>
<A>  f=6 t=7 <T>i=1</T>                              <T>i=5</T><T>i=6</T>          <T>i=8</T><T>i=9</T><T>i=10</T></A>

#if h=1 => k=1
```

```
<D> f=1 t=7 k=3 s=3 h=6</D>
<D> f=2 t=4 k=3 s=2 h=6</D>
<D> f=4 t=6 k=2 s=5 h=6</D>
```

Appendix E

List of acronyms

API	Application Programming Interface
A.Q.	Approximated Quality
A.Q. above R.S.	Approximated Quality above Random Search
AMD	Advanced Micro Devices
ARPANET	World's first operational packet switching computer network, developed in the Advanced Research Projects Agency
AS	Autonomous System
BGP	Border Gateway Protocol
bps	bits per second
Bps	Bytes per second
CIDR	Classless Inter-Domain Routing
CMST	Capacitated Minimum Spanning Tree problem
CPMST	Capacity Planning Minimum Spanning Tree problem
CPU	Central Processing Unit
DB	DataBase
F.A.	First Appearance
FDDI	Fiber Distributed Data Interface
G.A.	Genetic Algorithms
GB	Giga-Bytes
GGEngine	Generic Genetic Engine
GHz	Giga-Hertz
GSP	Generalised Steiner Problem
GSP-NC	Generalised Steiner Problem with Node Connectivity constraints
GSP-EC	Generalised Steiner Problem with Edge Connectivity constraints
HT	Hyper-Threading
IANA	Internet Assigned Numbers Authority
IGP	Interior Gateway Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IS-IS	Intermediate System to Intermediate System
ISP	Internet Service Provider
LAN	Local Area Network
MAN	Metropolitan Area Network
MB	Megabytes
Mb	Megabits
MPI	Message Passing Interface
MPLS	MultiProtocol Label Switching
MPLS-TE	MultiProtocol Label Switching Traffic Engineering
MST	Minimum Spanning Tree problem
$\mathcal{N}P$	Non-deterministic Polynomial time
$\mathcal{N}PO$	Non-deterministic Polynomial optimisation

NSP	Network Service Provider
OGF	Ordinary Generating Function
OSI	Open Systems Interconnection
OSPF	Open Shortest Path First protocol
P	Polynomial time
PO	Polynomial time optimisation
PVM	Parallel Virtual Machine
RIP	Routing Information Protocol
R.S.	Random Search
SFP	Single-demand Feasibility Problem
SPG	Steiner tree Problem in Graph
TC-DHK	Topology and Capacity design with throughput Demand, Hop-limit and path diversity constraints
TCP	Transmission Control Protocol
TSP	Travelling Salesman Problem
WAN	Wide Area Network
WMP	Weighted Multi-demand Problem
WSP	Weighted Single-demand Problem

Bibliography

[Agrawal 95] Ajit Agrawal, Philip Klein & R. Ravi. *When trees collide: An approximation algorithm for the generalized Steiner problem on networks*. SIAM Journal on Computing, vol. 24, pages 440–456, 1995.

[Ahn 01] Sang Ahn, Meejeong Lee & Tatsuya Suda. *Topology reconfiguration of an IP Network embeded over an ATM Network*. In Special Section in the APCC Proceedings, volume E84-B of *11*, pages 542–545. IEICE TRANS. COMMUN., November 2001.

[Allender 99] Eric Allender, Micheal C. Loui & Kenneth W. Regan. ALGORITHMS and THEORY of COMPUTATION HANDBOOK, chapter 29: Other Complexity Classes and Measures. CRC Press, 1999.

[Awduche 99] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell & J. McManus. *Requirements for Traffic Engineering Over MPLS*. IETF RFC2720, September 1999.

[Balakrishnan 91] A. Balakrishnan, T. L. Magnanti, A. Shulman & R. T. Wong. *Models for planning capacity expansion in local access telecommunication networks*. Annals of Operations Research, vol. 33, no. 4, pages 237–284, April 1991.

[Bhandari 99] Ramesh Bhandari. Survivable networks - algorithms for diverse routing. Kluwer, MA, 1999.

[Bley 97] Andreas Bley. Node-Disjoint Length-Restricted paths. Master's thesis, Technische Universität Berlin - Fachbereich Mathematik, January 1997.

[Botton 07] Quentin Botton & Bernard Fortz. *Extended Node-Arc Formulation for the K-Edge-Disjoint Hop-Constrained Network*. In International Network Optimization Conference, 2007. Available on: `http://www.poms.ucl.ac.be/inoc2007/Papers/author.118/paper/paper.118.pdf`, (link verified on 17th Oct 2007).

[Cahn 98] Robert S. Cahn. Wide Area Network Design: Concepts and Tools for Optimization. Morgan Kaufmann Publishers, San Francisco, 1998.

[Calegari 02] Daniel Calegari. *Algoritmos Genéticos Aplicados al Diseño de una red de comunicaciones confiable*. Rapport technique, Instituto de Computación, Facultad de Ingeniería, Universidad de la República, Uruguay, 2002. Available on: `http://www.fing.edu.uy/inco/pedeciba/bibliote/tgradoinf/tg-calegari.pdf`, (link verified on 17th Oct 2007).

[Cantus-Paz 94] Eric Cantus-Paz. *DGenesis 1.0*, April 1994. Available on: `http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/ga/systems/dgenesis/0.html`, (link verified on 17th Oct 2007).

[Chazelle 00] Bernard Chazelle. *A Minimum Spanning Tree Algorithm with Inverse Ackermann Type Complexity*. Journal of the ACM, vol. 47, no. 6, pages 1028–1047, November 2000.

[Clay Math. Inst. 05] Clay Math. Inst. *Clay Mathematics Institute*, October 2005. Available on: `http://www.claymath.org/millennium/P_vs_NP/`, (link verified on 17th Oct 2007).

[Coffman 01] K.G. Coffman & A. M. Odlyzko. *Growth of the Internet.* Rapport technique, AT&T Labs - Rearch, July 2001. Available on: `http://www.dtc.umn.edu/~odlyzko/doc/oft.internet.growth.pdf`, (link verified on 17th Oct 2007).

[Cormen 04] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein. Introduction à l'algorithmique. Dunod, Paris, 2 edition, 2004.

[Crescenzi 99] Pierluigi Crescenzi, Viggo Kann, Riccardo Silvestri & Luca Trevisan. *Structure in approximation classes.* SIAM Journal on Computing, vol. 28, no. 5, pages 1759–1782, 1999.

[Crescenzi 00] Pierluigi Crescenzi & Viggo Kann. *A compendium of NP optimization problems*, March 2000. Available on: `http://www.nada.kth.se/~viggo/problemlist/`, (link verified on 17th Oct 2007).

[Darwin 59] Charles Darwin. The origin of species by means of natural selection, or the preservation of favoured races in the struggle for life. Murray, London, 6th edition, 1859. Available on: `http://www.gutenberg.org/dirs/etext99/otoos610.txt`, (link verified on 17th Oct 2007).

[Devroye 86] Luc Devroye. Non-uniform random variate generation. Springer-Verlag, New York, 1986. Available on: `http://cg.scs.carleton.ca/~luc/rnbookindex.html`, (link verified on 17th Oct 2007).

[Diestel 00] Reinhard Diestel. Graph theory. Springer-Verlag, New York, 2000.

[Esbensen 95] Henrik Esbensen. *Computing near-optimal solutions to the Steiner problem in a graph using a genetic algorithm.* Networks, vol. 26, no. 4, pages 173–185, April 1995.

[Flajolet 06] Philippe Flajolet & Robert Sedgewick. *Analytic Combinatorics.* 8th web edition, 2006. Available on: `http://algo.inria.fr/flajolet/`, (retrieved on 23th Oct 2006).

[Garey 79] Michael R. Garey & David S. Johnson. Computer and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and company, New York, 1979.

[Gerla 77] Mario Gerla & Leonard Kleinrock. *On the Topological Design of Distributed Computer Networks.* IEEE Transactions on Telecommunications, vol. 25, no. 1, pages 48–60, January 1977.

[Goldberg 89] David E. Goldberg. Genetic Algorithms in Search, Optimization, and Machine Learning. Addison Wesley Longman, Inc., San Francisco, 1989.

[Grefenstette 90] John Grefenstette. *Genesis 5.0*, Octubre 1990. Available on: `http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/genetic/ga/systems/genesis/0.html`, (link verified on 17th Oct 2007).

[Grötschel 95] M. Grötschel, C.L. Monma & M. Stoer. Handbook in Operations Research and Management Science, Network Models, chapter 10: Design of Survivable Networks. North Holland, 1995.

[Hain 05] Tony Hain. *A Pragmatic Report on IPv4 Address Space Consumption.* The Internet Protocol Journal, vol. 8, no. 3, September 2005. Available on: `http://www.cisco.com/web/about/ac123/ac147/archived_issues/ipj_8-3/ipv4.html`, (link verified on 17th Oct 2007).

[Ho 02] Vincent Ho, Sergio Nesmachnow, Gerardo Ares, Pablo Ezzatti & Nelson Calero. *Motor Genético Genérico, Generic Genetic Engine - GGEngine 1.0 (versión Serial 1.0)*. Rapport technique, Instituto de Computatción, 2002. `http://www.fing.edu.uy/inco/pedeciba/bibliote/reptec/TR0305.pdf`.

[Hoos 04] Holger Hoos & Thomas Stützle. Stochastic local search : Foundations & applications. Morgan Kaufmann Publishers, 2004.

[Huang 97] Runhe Huang, Jianhua Ma & D. Frank Hsu. *A Genetic Algorithm for Optimal 3-connected Telecommunication Network Designs*. In Proceedings of the 1997 International Symposium on Parallel Architectures (ISPAN '97), page 344, 1997.

[Hwang 86] F. K. Hwang. *A linear time algorithm for full Steiner trees*. Operations Research Letters, vol. 4, no. 5, pages 235–237, February 1986.

[Itai 82] A. Itai, Y. Perl & Y. Shiloach. *The complexity of finding maximum disjoint path with length constraints*. Networks, vol. 12, pages 277–286, 1982.

[Jothi 04] R. Jothi & B. Raghavachari. *Revisiting Esau-Williams' algorithm: On the design of local access networks*. In In Proc. 7th INFORMS Telecommunications Conf., pages 104–107, 2004. Available on: `http://citeseer.ist.psu.edu/jothi04revisiting.html`, (link verified on 17th Oct 2007).

[Karp 72] R.M. Karp. *Reducibility among combinatorial problems*. Complexity of Computer Computations, pages 85–103, 1972.

[Kleinrock 02] Leonard Kleinrock. *Creating a Mathematical Theory of Computer Networks*. The Internet Protocol Journal, vol. 50, no. 1, pages 125–131, January-February 2002.

[Kruse 84] Robert L. Kruse. Data structure and program design. Prentice-Hall, New Jersey, 3 edition, 1984.

[Kumar 93] Anup Kumar, Rakesh M. Pathak & M. C. Gupta. *Genetic Algorithm based approach for designing computer network topology*. In Proceedings of the 1993 ACM conference on Computer science, pages 358–365, Indianapolis, 1993. ACM Press.

[Lee 04] G. Lee & J. Choi. *A survey of multipath routing for traffic engineering*, August 2004. Available on: `http://vega.icu.ac.kr/~gmlee/research/papers/a%20survey%20of%20multipath%20routing.pdf`, (link verified on 17th Oct 2007).

[Marczyk 04] Adam Marczyk. *Genetic Algorithms and Evolutionary Computation*, April 2004. Available on: `http://www.talkorigins.org/faqs/genalg/genalg.html#examples:routing`, (link verified on 17th Oct 2007).

[Martins 98] Simone L. Martins, Celso C. Ribeiro & Mauricio Souza. *A Parallel GRASP for the Steiner Problem in Graphs*. In Workshop on Parallel Algorithms for Irregularly Structured Problems, pages 285–297, 1998. Available on: `http://citeseer.ist.psu.edu/martins98parallel.html`, (link verified on 17th Oct 2007).

[Matsumoto 98] Makoto Matsumoto & Takuji Nishimura. *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*. ACM Transactions on Modeling and Compuyter Simulations: Special Issue on Uniform Random Number Generation, vol. 8, no. 1, pages 3–30, January 1998. Source downloadable from `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/eindex.html`, (link verified on 17th Oct 2007).

[Matsumoto 06] Makoto Matsumoto. *Mersenne Twister (mt19937ar) - Download Page*, January 2006. Available on: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`, (link verified on 17th Oct 2007).

[Meyer 06] D. Meyer & K. Patel. *BGP-4 Protocol Analysis*. IETF RFC4274, January 2006.

[Nesmachnow 04] Sergio Nesmachnow. Algorítmos Genéticos Paralelos y su aplicación al diseño de redes de comunicaciónes confiables. Master's thesis, Universidad de la República Uruguay - Facultad de Ingeniería, July 2004. Available on: `http://www.fing.edu.uy/inco/pedeciba/bibliote/tesis/tesis-nesmachnow.pdf`, (link verified on 17th Oct 2007).

[Papadimitriou 78] C. Papadimitriou. *The complexity of the capacitated tree problem*. Networks, vol. 8, pages 217–230, 1978.

[Pióro 04] Michal Pióro & Deepankar Medhi. Routing, Flow, and Capacity Design in Communication and Computer Networks. Morgan Kaufmann Publishers (an imprint of Elsevier), 2004.

[Ríbnikov 85] K. Ríbnikov. Análisis combinatorio. Mir, Moscú, 1985.

[Rekhter 06] Y. Rekhter, T. Li & S.Hares. *A Border Gateway Protocol 4 (BGP-4)*. IETF RFC4271, January 2006.

[Ribeiro 00] Celso C. Ribeiro & Maurício C. De Souza. *Tabu Search for the Steiner Problem in Graphs*. Networks, vol. 36, no. 2, pages 138–146, April 2000.

[Robertazzi 00] Thomas G. Robertazzi. Computer networks and system: Queueing theory and performance evaluation. Springer Verlag, New York, 2000.

[Robins 00] Gabriel Robins & Alexander Zelikovsky. *Improved Steiner Tree Approximation in Graphs*. In Proceedings of the 11th annual ACM-SIAM symposium on Discrete algorithms, pages 770–779, 2000.

[Robledo 00] Franco Robledo. Diseño Topologico de R-des, Caso de estudio: "The Generalized Steiner Problem" and "The Steiner 2-E-ge-Connected Subgraph Problem". Master's thesis, Universidad de la República Uruguay - Facultad de Ingeniería, February 2000.

[Syswerda 89] G. Syswerda. *Uniform Crossover in Genetic Algorithms*. In J. Schaffer, editor, In Proc. of the Third International Conference on Genetic Algorithms, pages 2–9, Los Altos, CA, 1989. Morgan Kaufmann.

[Tanenbaum 03] Andrew S. Tanenbaum. Computer networks. Prentice Hall, 4th edition, 2003.

[Univ. de Málaga 07] Univ. de Málaga, Universidad de La Laguna & Universitat Politècnica de Catalunya. *The MALLBA Project's web page*, October 2007. Available on: `http://www.lsi.upc.edu/~mallba/`, (link verified on 17th Oct 2007).

[Valiant 79] L. Valiant. *The complexity of computing the permanent*. Theoretical Computer Science, vol. 8, pages 189–201, 1979.

[Weisstein 05] Eric W. Weisstein. *MathWorld–A Wolfram Web Resource*, October 2005. Available on: `http://mathworld.wolfram.com/`, (link verified on 17 Oct 2007).

[Westbrook 95] J. Westbrook & D. Yan. *Linear bounds for on-line steiner problems*. Information Processing Letters, vol. 55, pages 59–63, 1995.