QoS Routing in IP Networks
Using Multi-Constrained Computational Methods

A THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH

By
T. M. Fathelrahman
March 2008

Supervised by: Jaco Geldenhuys

Declaration

I the undersigned hereby declare that the work contained in this thesis is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Signature: ..... Date: .....

Summary

In this thesis, we consider the multi-constraints QoS routing problem in IP networks. Namely, we consider the problem of minimizing the path delays on IP networks. We use genetic algorithms to perform the optimization, some penalty function methods and the simulated annealing method for handling the problems constraints.

Our aim is to compare the performance of different penalty function methods and the simulated annealing method. The penalty function methods under consideration include penalty methods with non-stationary as well as stationary penalty coefficients. The basis for doing the comparisons are the maximum link and path delays, the maximum and average path length, and the CPU time.

We used four virtual networks as test examples. We found that, generally, the performances of the simulated annealing method, the dynamic and co-evolutionary penalty function methods are better than the performances of the adaptive, annealing and the static penalty function methods. Dynamic coefficients seem to have a slight edge over stationary coefficients. Simulated annealing turned out to be the slowest of the approaches investigated.

Afrikaans summary

Hierdie tesis ondersoek hoe om die multi-beperking QoS ("quality of service") roeteringsprobleem vir IP netwerke op te los. Meer spesifiek, die doel is om die netwerkpadvertragings te minimeer. Genetiese algoritmes word gebruik om die probleem deur middel van optimering op te los, en die multi-beperkings word hanteer met behulp van *boetefunksies*. Daar word ook gekyk na die tempersimulasie benadering ("simulated annealing").

Die doel van die tesis is om die boetefunksies en tempersimulasie te vergelyk. Beide konstante en nie-konstante boetefunksies word ondersoek en nuwe konstante boetefunksies word geformuleer deur die nie-konstante boetekoëffisiënte vas te pen. Al hierdie metodes word gemeet deur te kyk na die maksimum skakel- en padvertraging, die maksimum en gemiddelde padlengte, en die verwerkingstyd.

Vier virtuele netwerke word gebruik as 'n toetsraamwerk. Die uiteindelike gevolgtrekking is dat die verskillende boetefunksies rofweg dieselfde antwoorde produseer. Nie-konstante koëffisiënte presteer ietwat beter as konstante koëffisiënte. Die tempersimulasie was aan die einde van die dag, die stadigste benadering waarna gekyk is.

Acknowledgements

I would like to thank the Siemens-Telkom Centre of Excellence of ATM and Broadband Networks and their Applications, who funded my Masters project until I have completed it.

Thanks to my supervisor Dr Jaco Geldenhuys for supervising my thesis and Prof A.E.Krzesinski the head of our team.

Special thanks to my husband Dr. Eihab Bashier for his love and support, my parents and sisters for their continuous supports and prayers.

Finally my thanks also go to all my colleagues and friends who helped me during my stay at South Africa.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The last few decades have witnessed a rapid growth in Internet applications such as chat programs, video conferencing, social networking, remote document processing, and so forth. This has raised the problem of how to satisfy the quality of service (QoS) requirements of such applications. These requirements concern bandwidth, delay, packet loss, and the reliability of the network. Recently, several approaches have been proposed to provide different levels of QoS. These include Integrated services (IntServ), Differentiated Services (DiffServ), Asynchronous Transfer Mode (ATM), and Multi-Protocol Label Switching (MPLS) [29, 51].

One common key issue in the design of such services is how to identify a feasible route that satisfies multiple constraints while simultaneously achieving efficient utilization of network resources. This problem is known as *QoS (or constraint-based) routing* [29], and the constraints imposed by the QoS requirements (bandwidth, delay and packet loss) are referred to as *QoS constraints* [52]. Unfortunately, QoS routing problems are known to be NP-complete [25, 29, 33].

In IP networks, routing protocols such as OSPF (open shortest path first) and IS-IS (Intermediate System-Intermediate System) are used to route packets from their sources to their destinations. These routing protocols select paths based on static link weights that are configured by network operators and make information about the link weights available at each of the network routers. According to the link weights each router computes the shortest paths and creates a table to control the forwarding of the IP packets to the next routers [16].

Younis described different setups for QoS routing problems [52]. They include routing problems with bounds on bandwidth, problems with bounds on delay, problems with bounds on both bandwidth and delay, problems for the optimization of bandwidth and delay, problems for the optimization of a given

cost with bounds on delay, and multi-constrained routing problems. In essence, QoS routing has two components: the collection of QoS information and the computation of QoS routes [33]. The former means that the structure and state of the network must be completely known and available at each node [29]. The collection of data can be achieved by mechanisms such as the OSPF extension [33], and the information gathered may include metrics such as the delay, bandwidth, and packet loss rates.

QoS routing is usually described as a multi-constraint graph problem, where each edge (also known as a *link*) has associated weights, defined by one or more of the metrics mentioned above. The weights can be classified as either additive or non-additive. For additive parameters such as the delay, the cost of an end-to-end path is given by the sum of the links weights along that path. On the other hand, the cost of a path for a non-additive parameter is determined by a *bottleneck link*.

Korkmaz and Krunz has stated the multi-constraint optimal path problem as follows [29]:

Definition 1 *Multi-Constrained Optimal Path (MCOP) Problem: Consider a network that is represented by a directed graph $G = (V, L)$, where V is the set of nodes and L is the set of links. Each link $(i, j) \in L$ is associated with a primary cost parameter $c(i, j)$ and K additive QoS parameters $w_k(i, j)$, for $k = 1, \ldots, K$; all parameters are non-negative. Given K constraints $c_1, \ldots, c_k$, the problem is to find a path ρ from a source node s to a destination node d such that*

(i) $w_k(\rho) \stackrel{def}{=} \sum_{(i,j)\in\rho} w_k(i, j) \leq c_k$ *for* $k = 1, \ldots, K$, *and*

(ii) $c_\rho \stackrel{def}{=} \sum_{(i,j)\in\rho} c(i, j)$ *is minimized over all the feasible paths satisfying (i).*

When $K = 1$, the resulting problem is referred to as the *restricted shortest path (RSP)* problem, and it is NP-complete [29]. Other examples of QoS multi-constraint routing problems include the *multi-constraint path (MCP)* problem, which consists of finding a feasible path subject to multiple constraints on a given network without minimizing any objective function [25], and the *multiple-constraint shortest path (MCSP)* problem, which consists of finding the shortest path with respect to the hop count (i.e., path length) that satisfies the constraints [33].

1.1 Heuristic Solutions for Constrained Optimization

Precise, exhaustive approaches to solving constrained optimization problems are not feasible for large problems, and heuristic approaches often fare much better. In this thesis, genetic algorithms based on five different penalty function methods and the simulated annealing method are investigated.

Genetic algorithms are powerful techniques for solving optimization problems. A genetic algorithm (GA) starts with an initial random population of potential solutions called *genomes*. A GA works iteratively, and each iteration is known as a *generation*. In each generation a subset of the population — the *parents* — is selected according to some fitness function, and genetic operators such as crossover and mutations are applied to produce a new generation of solutions. The genetic operators ensure that the next generation is diverse enough to include a good, if not globally optimal, solution. The GA terminates when no improvement in the fitness of the best individual in the population can be achieved, or when the maximum number of generations is reached. Sometimes a local search technique is used together with a GA, and the resulting method is known as a *memetic algorithm* or a *hybrid genetic algorithm* [48].

For constrained optimization problems, a method for combining the constraints of the problem into the objective (goal) function is needed. *Penalty function methods (PFMs)* is one classical approach to do this. PFMs penalize infeasible solutions by applying penalty weights to the constraints and adding them to the objective function. Some PFMs place penalties on the amount by which the constraints are violated, while others consider both the amount by which the constraints are violated and the number of constraints violated. If the penalty factors are large enough, the solution of the constrained optimization problem converges to the solution of the resulting problem [7]. There are two classes of PFMs, namely non-stationary penalties and stationary penalties. In the non-stationary penalty methods, the penalty factors change at each iteration, while in the stationary penalty methods the penalty factors remain fixed during the whole optimization process.

In 1983, Kirkpatrick and others proposed the *simulated annealing method* for solving optimization problems [28]. The aim is to exploit the similarities between the physical process of heating a metal and then cooling it, and the combinatorial challenge of solving optimization problems. When a metal is heated to a high temperature and then slowly cooled to its freezing point according to some cooling schedule, the atoms have the freedom to find a configuration with low internal energy. The idea behind the simulated annealing method is to make an analogy between the value of the energy of a physical system and the value of the objective function of an optimization problem. At the freezing temperature both the internal energy of the physical system and the objective function of the optimization problem are supposed to achieved their minimum values.

1.2 Related Work

The field of OSPF and MPLS routing is large and important, and it is beyond the scope of this thesis to discuss its general background. An excellent overview of the role of GAs (or *evolutionary computation* as it is also known) in telecommunications problems is the survey by Kampstra, Van der Mei, and Eiben, which extends an earlier survey by Sinclair [26, 45]. Similarly, an enormous amount of research has been devoted to constrained optimization and its application to routing problems [41], and, over the last few decades, its combination with GAs and other heuristics approaches. Within this context, the subfield of penalty function methods has been surveyed by Coello [12]. Lastly, some of the work on the application of general heuristic techniques to QoS routing problems has been surveyed by Kuipers et al. [30, 31].

A lot of work over the last decade has focused QoS multicast routing with GAs [2, 10, 19, 23, 40, 42, 43, 46, 53, 54].

Chen and Dong [10] considered a constrained QoS multicast routing problem. Their approach to solve the problem was by using a generalized fuzzy-constrained fuzzy optimization model. They found that their algorithm is efficient.

Haghighat et al. [19] investigated various problem encodings and variations of the genetic operations, and compare their performance to other, more established algorithms. They use a stationary penalty function, but suggest that non-stationary penalties may lead the algorithms to converge faster. An important difference between their work and the approach described in this thesis is that *multicast* routing differs significantly from *unicast* routing. For the former, a solution takes the form of a Steiner tree, while in the latter case the algorithm produces a link-weight function.

Randaccio and Atzori [42] have address the problem of group multicast routing making use of genetic algorithms. Their solution is divided into two steps. Firstly a set of possible solutions are generated for each session in isolation. Secondly the combinations of these are evaluated by means of a cost function that weights the transmission delay and the network resource utilization concurrently.

Roy and Das [43] developed a protocol to optimize multiple QoS parameters and to solve constrained routing problem in a wireless network. Their simulation is shown to be capable of discovering a set of QoS-based near-optimal multicast routes within a few iterations.

Siregar et al. [46] considered the problem of minimizing the number of split-capable nodes in the network for a given set of multicast requests. They used a genetic algorithm to exploit the combination of alternative shortest paths for the given multicast requests in order to minimize the number of required split-capable nodes. They experimented with their algorithm in two real networks in Japan, and their

simulation showed that this algorithm can reduce more than 10% of split-capable nodes compared with other routing algorithms.

Zhang and leung [54] proposed an orthogonal genetic algorithm for multimedia multicast routing, where in their genetic algorithm they incorporated an experimental method called orthogonal design into the crossover operation. Their result showed that for practical problem sizes the orthogonal genetic algorithm can find near-optimal solutions within a moderate number of generations.

With regard to unicast routing with GAs, Xunxue, Qin, and Qing [50] study a slight variation of the MCOP problem. In their approach, two objective functions (based on the *cost* and *delay* metrics) are minimized using a GA, multi-objective optimization, and the concept of a Pareto optimum. They compare their approach to QoS routing with a (non-genetic) bandwidth-delay-constrained path algorithm [47] and a single-objective genetic algorithm [49], and show that their algorithm produces an average packet loss rate that is significantly less than that of the other algorithms and, moreover, improves as the network size grows, while the other two algorithms deteriorate. Their algorithm also outperforms the others when it comes to the network blocking rate.

To the best of our knowledge, the only previous application of GAs to multi-constrained QoS routing using PFMs was reported in our previous work [15]. There we compare the adaptive penalty method and the self-adaptive penalty method for the optimization of the delay metric of Balon et al. [4, 5], which describes the weighted mean link delay on an $M/M/1$ queue. The problem is of the form

$$F_1 = \sum_{\ell \in L} \frac{f_\ell}{C_\ell - f_\ell} \tag{1.1}$$

subject to the constraints

$$f_\ell \leq C_\ell \quad \text{for } \ell \in L, \tag{1.2}$$

where f_ℓ is the number of flows on link ℓ and C_ℓ is the total link bandwidth. The results we obtain show that the adaptive penalty methods fare better than the self-adaptive penalty methods.

When it comes to the use of simulated annealing for QoS routing, it is worthwhile to answer the question of how the constraints of the problem can be handled. Cui et al. [13] proposed a method to find an optimal path using the simulated annealing method. Their method first translates multiple QoS weights into a single metric and then looks for a feasible path and iterates to find the optimum cost without losing the feasibility. The technique of looking for a feasible path by simulated annealing is also used by Liu and Feng in [34].

1.3 Thesis Overview

The problem that the thesis is solving is stated as follows:
Given an IP network with a known structure. That is for each two adjacent nodes the total bandwidth of the link between them is known. Also given sets of source-destination pairs with the required bandwidths to transmit the packets from their sources to their destinations.

This thesis proposes that before moving packets from a source node to a destination node, the optimal link weights shall be computed and assigned to the network links such that the following properties are satisfied:

1. either the weighted mean delay metric (equations (1.1)-(1.2)) described by Balon et al. [5], or the mean delay metric (1.3)-(1.4) below

$$F_2 = \sum_{\ell \in L} \frac{1}{C_\ell - f_\ell} \tag{1.3}$$

   subject to:

$$f_\ell \leq C_\ell \quad \text{for } \ell \in L, \tag{1.4}$$

   described by Elwalid et al. [14] is minimized,

2. the network links are efficiently utilized.

In this thesis, genetic algorithms based on five different penalty function methods and the simulated annealing method are used to solve the two optimization problems, where the output of these optimization methods is the set of optimal link weights to be assigned to the network links before transmitting packets from their source to their destination.

Our objective is to compare between the performances of the five different implementations of genetic algorithms and the simulated annealing method described by Coello [12] on solving the two QoS routing problems.

Chapter 2 contains three parts. The first introduces the basic concepts of GAs, including the operators, parameters, advantages, and disadvantages. The second part presents the formulations for different PFMs and the third part gives some background about the simulated annealing method.

Chapter 3 discusses how GAs can be used to adjust a network's links and how PFMs work to penalize the infeasible solutions that violate the constraints. An implementation of PFMs with both non-stationary coefficients and stationary coefficients is described. The PFMs under consideration are the dynamic, annealing, adaptive, and self-adaptive (co-evolutionary) methods. It also discusses the implementation

of the simulated annealing method to solve the two QoS routing problems —weighted mean link delay and mean link delay— that are considered in this thesis, when the technique of Cui et al. [13] will be considered to handle the problem constraints.

The performance of the various methods is compared in Chapter 4, and conclusions and recommendations are presented in Chapter 5.

Chapter 2

Preliminaries and Background

To sketch how genetic algorithms work, we consider the simple network given in Figure 2 below.

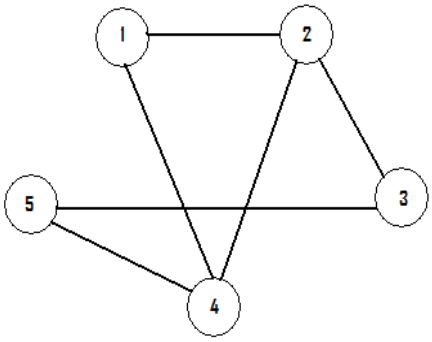

Figure 2.1: A simple network consisting of 5 nodes and 6 links.

Figure 2 above shows an example of a small network. There are five nodes connected to each other as shown. Each of the connections, say from node n_1 to node n_2, has a maximum capacity that we denote by TBW(n_1, n_2). From time to time there is a traffic demand: a request to transmit packets from a source node to a destination node, and with each demand we associate a required bandwidth. The job of the network administrator is to assign weights to the connections between the nodes so that the mean delay (or whatever metric is important to the owners of the network) is minimized. The weights can be adjusted before any given traffic demand is handled, and the core question of this thesis is how the weights should be adjusted and how the algorithms we study can help with the adjustments.

2.1 Genetic Algorithms (GAs)

The last decade has experienced an increasing interest in the use of evolutionary optimization strategies to solve nonlinear programming problems. Evolutionary algorithms (EAs) use concepts from real-world genetics to "evolve" solutions to problems according to the principles of Darwinian evolution theory. They are based on an evolutionary paradigm where each iteration of the algorithm transforms one population of individuals into a new generation, using some pre-defined fitness measure to determine which solutions survive and which are eliminated [15].

In applications of evolutionary algorithms, a potential solution is encoded as a list of values called a *genome*. We shall use the terms "genome", "solution", and "individual" interchangeably in the rest of this chapter. Each problem generally has its own genome encoding, although several alternative representations are usually available for a given problem. The fitness measure or fitness function determines how "good" the solution represented by some genome is. The appropriate fitness function is determined by the problem and by the genome representation [17, 39].

GAs rely on three main genetic operations to produce the next generation of the population: reproduction, crossover and mutation. Reproduction (also known as selection) chooses a subset of the individuals in the current generation that will be used to form the next generation. Crossover combines two genomes from the current generation to produce two different genomes in the next generation. Crossover attempts to combine good solutions to find potentially better solutions. Mutation is the random permutation of one of the values in a genome. The next section discusses these operations in more detail [12, 17].

The introduction of new solutions in each iteration of the algorithm ensures that the evolutionary process will not get stuck at a local optimum, and the semi-random, fitness-proportionate way in which candidates for the genetic operations are selected favours the survival of good solutions. The termination condition for a GA is usually problem-specific, although for practical reasons there is often a limit on the total number of iterations.

Although GAs can find acceptably good solutions to problems by examining and manipulating a set of possible solutions, they are not guaranteed to find a globally optimal solution to a problem. GAs have been deployed in the literature to solve real-life problems using different strategies. The selection of the appropriate strategy to be used for a specific application is, naturally, an important factor in the success of an evolutionary algorithm [15].

2.1.1 Genetic Operations

As mentioned above, there are three important operations that are performed by GAs to produce to the next generation.

Reproduction

The reproduction operation randomly selects members of the current generation and copies them, unchanged, to an intermediate generation. The individuals can be selected in various ways, and four typical techniques are described below. The first three make use of weighted selection: each solution i is given a weight w_i, and a uniform random number is used to pick a solution. If one solution has double the weight of another, the heavier solution is twice as likely to be selected.

1. **Random selection:** The simplest method for selecting individuals assigns a unit weight $w_i = 1$ to each genome.
2. **Roulette wheel selection:** A more intuitive way to ensure that fitter solutions are selected to produce the next generation is to weight the genomes according to their fitness: $w_i = \mathit{fitness}_i$.
3. **Rank selection:** The individuals in the population are first ranked according to their fitness. If the population contains n individuals, the fittest individual i is weighted $w_i = n$, the second fittest individual j is weighted $w_j = n - 1$, and so on, until the least fit individual k is weighted $w_k = 1$.
4. **Tournament selection:** This last method works slightly differently, because it does not make use of straightforward weighted random selection. The entire population of n individuals is randomly divided into subgroups of size m. In each subgroup the fittest individual is selected, and the $\lceil n/m \rceil$ subgroup "winners" form the parents that produce the next generation.

Crossover

Once the current generation has been reproduced, the individuals in the intermediate generation are randomly paired off. Some pairs remain unchanged, but others — with probability p_c — are subjected to the crossover operation. The purpose of crossover is to enable the pair to exchange parts of their genomes. This allows the features of two fit parents to be combined to produce two even fitter offspring that will replace them in the intermediate generation. The crossover operation can be performed in different ways:

1. **Single point crossover:** A single position P in the genome is selected randomly. (We assume that all the genomes are the same length, but it is not difficult to adapt this operation for handling genomes of varying length). One offspring inherits the head of the first parent (the part of the genome before position P), and the tail of the second (the part after position P), while another offspring inherits the tail of the first and the head of the second.

2. **Dual point crossover:** Two positions P_1 and P_2 are selected randomly, and the parts of the parent genomes between P_1 and P_2 are exchanged to produce the offspring.

3. **Uniform crossover:** Each value in the genome of the offspring is taken from one of the parents' genomes at the corresponding position. Which of the parents' values is used is determined randomly, with equal probability that the value will come from either parent. If the parents' genomes disagree in k positions, then 2^k different offspring are possible, but only two new genomes are produced.

Mutation

As a last step in the process, each value of each genome in the intermediate generation is subjected to mutation with a small probability p_m. This is done by replacing the value of the genome with a new, random value.

2.1.2 GA Parameters and Outline

The first step in applying a GA for a specific problem is to select an appropriate genome encoding and fitness function. Once this has been decided, the basic parameters that determine the behaviour of the GA are the crossover probability p_c, the mutation probability p_m, the population size N, and the maximum number of iterations *MaxGen*. These choices represent a trade-off between how quickly the GA converges, and how "good" the convergence is. If the population size is too small, only a small part of the search space may be explored, and the GA may be unable to find satisfactory solutions. On the other hand, if the population is large and the fitness function is complex, the GA may take a long time to converge. Similarly, if p_c and p_m are too small, the GA will be slow to discover new solutions; if they are too large, the GA may jump from solution to solution without coverging within a reasonable time. Here then, is an overview of a general GA:

1. Find an encoding for the problem and a strategy for evaluating the fitness of a given individual.

2. Determine the crossover probability p_c, the mutation probability p_m, the population size N, and the maximum number of generations $MaxGen$.

3. Generate an initial random population of size N.

4. Set $CurGen := 1$.

5. While $CurGen \leq MaxGen$:

   (a) Select individuals for the intermediate generation, based on their fitness.

   (b) Apply crossover to the intermediate generation with probability p_c.

   (c) Mutate the intermediate generation with probability p_m to form the next generation.

   (d) Set $CurGen := CurGen + 1$.

2.1.3 Local Search Methods

GAs start with an initial random population of solutions. Using the given fitness function a new set of solutions is computed by applying the genetic operators. Those new solutions must have better fitness than their predecessors. The process terminates when no improvement in the solution can be obtained, or when the maximum number of generations has been exceeded.

In contrast to this, more traditional gradient-descent methods start from an initial guess for the optimal solution. They then follow the gradient of the objective function towards a fitter solution until reaching the locally optimal solution that is closest to the initial guess. Such methods use the so-called *Hessian matrix* to compute gradients. At the optimal solution, the Hessian matrix must be positive definite, while the gradient is zero.

The process terminates when the directional derivative is less than a given tolerance ε_1, and the maximum constraint violation is less than another given tolerance ε_2.

Traditional methods are also known as *local search* methods because they move from one solution to a fitter solution within the neighbourhood of the first, according to some well-defined metric. GAs, on the other hand, can jump more wildly from one solution to another because of the crossover and mutation operations, and the neighbours of a solution form a much larger set. Intuitively, GAs have a larger opportunity for finding the globally optimal solution since many solutions are explored in parallel, and the GA returns the fittest solution it encounters. However, due to its stochastic nature it cannot guarantee that the global optimum will be found. It is possible to integrate the traditional methods into a GA by applying a local search to each member of the population after a new generation has been

generated. Such a combination is then known as a *memetic algorithm* or *hybrid* GA, but this technique is not applied in this thesis.

2.1.4 Advantages and disadvantages of GAs

GAs have been used in a variety of problems in a number of different fields. Some of their most important advantages include the following:

- GAs are mathematically much simpler than local search methods since they do not require the (potentially expensive) calculation of gradients.
- GAs are suitable for problems with large search spaces and particularly with a large number of local optima, in which traditional methods tend to get trapped.
- GAs can explore many solutions in parallel.
- GAs tend to be less domain-specific than traditional methods. Apart from the genome encoding and fitness function, the operation of a GA is fixed. The crossover and mutation probabilities and the population size are parameters that can be used to finetune the performance of a GA.

GAs do have some drawbacks, mainly related to the genome encoding and fitness function:

- It is important to ensure that the genome encoding is robust enough to tolerate the random changes produced by crossover and mutation, so that fatal errors and undesirable results are avoided. In practice, however, this is usually not difficult to achieve.
- It is not always easy to find an appropriate fitness function.
- Other heuristic approaches that use more domain-specific knowledge may be more difficult to implement, but may find good solutions much faster than a general-purpose GA.
- The best values of the crossover and mutation probability, the population size, and maximum number of generations are not always obvious and may have to be adjusted to suit specific problems. Unfortunately there are few practical guidelines available on how to do this.
- The implementation of the fitness function is an important factor in a GA's runtime efficiency.

2.2 Penalty Function Strategies

Because the genetic operations that are used to manipulate the genomes of a population are random, the application of a GA to a constrained optimization problem may produce infeasible solutions. This can be solved by preventing such solutions from entering the population (known as the *death penalty method*), or by repairing the infeasible offspring. Special repair algorithms can convert infeasible individuals back to feasible individuals. One drawback of using repair algorithms is that the problem of finding a feasible individual may sometimes be just as difficult as the optimization problem itself [36].

Another, very different approach is to modify the genetic operations so that they preserve specific properties. This idea is used in the GEnetic algorithm for Numerical Optimization for COnstraint Problems (*GENOCOP*) project, which solves linear constrained optimization problems for which the feasible region is convex [24]. But those modified algorithms suffer from many restrictions, they have a very limited performance, and work on specific assumptions.

A fourth approach is the use of penalty function methods (PFMs) which transform the constrained optimization problem into an unconstrained optimization problem by adding the constraints to the objective function.

There are two kinds of penalty that can be used:

1. penalties based on the amount by which the constraints are violated (according to the distance from the feasible region), and
2. penalties based on the number of constraints violated.

According to Joines and Houck, the first class of penalty functions usually outperforms the second class [24].

PFMs use coefficients to weight the different kinds of penalties before they are added to the objective function. When the coefficients are high, convergence is achieved more rapidly, but this may also eliminate infeasible solutions that serve as "stepping stones" to fitter solutions. The way in which the coefficients are determined, allows us to further classify PFMs:

1. **Stationary penalty methods:** Coefficients are computed before the GA is started, and remain constant throughout the generations. An example of this class is the *static* penalty method.
2. **Non-stationary penalty methods:** Coefficients are re-computed after every generation. Examples of this class are the *dynamic*, *annealing*, *adaptive*, and *co-evolutionary* (*self-adaptive*) penalty methods.

The combination of GAs and PFMs is one of the important methods that can be employed to solve constrained nonlinear programming problems. Many different PFMs have been described in the literature; the survey by Coello is a good source of information in this regard [12]. The methods vary in the computational cost they exact, in the quality of the solutions they produce, and in how they handle the constraints.

The optimization problem can be formulated as follows:

$$\min_{\mathbf{x}\in\mathbb{R}^n} f(\mathbf{x}) \tag{2.1}$$

subject to:

$$g_i(\mathbf{x}) \leq 0, \qquad i = 1, \ldots, m \tag{2.2}$$

$$h_j(\mathbf{x}) = 0, \qquad j = 1, \ldots, p \tag{2.3}$$

where m is the number of inequality constraints, and p is the number of equality constraints. When their exact nature is not important, we shall refer to the constraints simply as $c_k(\mathbf{x})$ for $k = 1, \ldots, n$, where $n = m + p$.

2.2.1 Static Penalties

Homaifar, Lai and Qi [22] propose that users define several levels of violation. The method entails the following steps:

1. Generate ℓ levels of violation for each constraint.
2. Generate a penalty coefficient R_{ij} $(i = 1, \ldots, n; j = 1, \ldots, \ell)$ for each constraint and each level of violation. Higher levels are associated with larger coefficients.
3. Generate a random population containing both feasible and infeasible individuals.
4. Evaluate the fitness of individuals as follows:

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + \sum_{i=1}^{n} R_{i\phi(c_i(\mathbf{x}))} \tag{2.4}$$

R_{ij} is the penalty coefficient corresponding to the i-th constraint and the j-th level of violation, $f(\mathbf{x})$ is the value of the (penalty-free) objective function, and $\phi(\cdot)\colon \mathbb{R} \to \{1, \ldots, \ell\}$ is a function that classifies the violation according to the user-defined levels. The disadvantages of this technique are that a large number of coefficients need to be stored, and, more importantly, determined by the user, a tedious and not-too-straightforward task.

Kuri-Morales [32] suggests another approach to static penalties, where the fitness of an individual is calculated by

$$\mathit{fitness}(\mathbf{x}) = \begin{cases} f(\mathbf{x}) & \text{if the solution is feasible} \\ K - \sum_{i=1}^{s} K/n = K - \frac{sK}{n} & \text{otherwise} \end{cases} \tag{2.5}$$

where K is a large positive constant, and s is the number of non-violated constraints and n is the number of all constraints. One problem with this approach is that it only uses information about the number of violated constraints, not the amounts by which the constraints are violated.

Finally, Hoffmeister and Sparve [21] use a penalty function of the form

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + \sqrt{\sum_{i=1}^{n} c_i(\mathbf{x})^2 H(-c_i(\mathbf{x}))} \tag{2.6}$$

where $H\colon \mathbb{R} \to \{0, 1\}$ is the Heaviside step function defined by

$$H(y) = \begin{cases} 1 & \text{if } y > 0 \\ 0 & \text{if } y \leq 0 \end{cases}$$

This method is based on the assumption that the infeasible solutions are always less fit than the feasible solutions, but Michaelewicz has shown that this is not always so [38].

2.2.2 Dynamic Penalties

Dynamic penalty methods are designed so that the penalty coefficients change from one generation to the next generation, and in most cases, the coefficients are set to increase as time evolves. Joines and Houck [24] suggest the following form for a dynamic penalty:

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + (tC)^{\alpha} SVC(\beta, \mathbf{x}) \tag{2.7}$$

where

$$SVC(\beta, \mathbf{x}) = \sum_{i=1}^{m} \phi_i^{\beta}(\mathbf{x}) + \sum_{j=1}^{p} \psi_j(\mathbf{x}) \tag{2.8}$$

and

$$\phi_i(\mathbf{x}) = \begin{cases} 0 & \text{if } g_i(\mathbf{x}) \leq 0 \\ |g_i(\mathbf{x})| & \text{otherwise} \end{cases}$$

and

$$\psi_j(\mathbf{x}) = \begin{cases} 0 & \text{if } -\epsilon \leq h_j(\mathbf{x}) \leq \epsilon \\ |h_j(\mathbf{x})| & \text{otherwise} \end{cases}$$

and t is the number of the current generation, α, β, and C are parameters specificied by the user, and ϵ is a small positive tolerance value. The authors state that the method is very sensitive to the parameters α and β, but do not mention its sensitivity with respect to C. However, they propose the values $\alpha = \beta = 2$ as optimal choices for the parameters. However, Michalewicz has tested the method on many examples and reports that

1. it grows too fast to be useful (because in some experiments the fittest individual found occurs in an early generation);
2. the system has little chance to escape from local optima; and
3. for some experiments the algorithms produce infeasible solutions.

Kazarlis and Petridis [27] and Coello [12] report on a detailed study of the dynamic penalty function

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + V(t)\psi(\mathbf{x}) \left(B + A \sum_{i=1}^{n} \delta_i w_i \phi(d_i(\mathbf{x})) \right) \tag{2.9}$$

where

$$\delta_i = \begin{cases} 1 & \text{if constraint } i \text{ is violated} \\ 0 & \text{otherwise} \end{cases}$$

and

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \text{ is infeasible} \\ 0 & \text{otherwise} \end{cases}$$

and A is a severity factor, B is a penalty threshold factor, w_i is a weight factor for constraint i, $d_i(\mathbf{x})$ is a measure of the degree to which solution $\mathbf{x}$ violates constraint i, $\phi_i(\cdot)$ is a function of this measure, $\psi(\mathbf{x})$ is a binary factor, and $V(t)$ is an increasing function of (the current generation number) with range $[0, 1]$. The authors experimented with linear, quadratic, cubic, exponential, and 5-part stepwise versions of $V(\cdot)$, and found that the best performance results from

$$V(t) = (t/T)^2 \tag{2.10}$$

where T is the total number of generations. They refer to this method as the *varying fitness function* technique. In their experiments they used the values $A = 1000$ and $B = 0$. As in the case of the Homaifar-Lai-Qi static penalties, this method requires a large number of user-defined parameters (A, B, and w_i) and it is not immediately clear how their values should be chosen.

2.2.3 Annealing Penalties

The annealing penalty method is a variation of dynamic penalties, first described by Michalewicz and Attia [37, 39]. The constraints are divided into four subsets: linear equalities, nonlinear equalities, linear

inequalities, and nonlinear inequalities. A random starting point that satisfies the linear constraints is chosen, and the population evolves according to the formula

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + \frac{1}{2\tau}\sum_{i=1}^{n}\phi_i^2(\mathbf{x}) \tag{2.11}$$

where

$$\phi_i(\mathbf{x}) = \begin{cases} max\{0, g_i(\mathbf{x})\} & \text{if } 1 \leq i \leq m \\ |h_{i-m}(\mathbf{x})| & \text{if } m < i \leq n \end{cases}$$

and τ is a temperature that decreases in each generation. The best solution of each generation serves as the starting point of the next iteration. The algorithm terminates when τ reaches the freezing temperature τ_f. One problem with this approach is its sensitivity to the values of the parameters and in particular the cooling schedule for τ [28].

The fact that the initial population must satisfy the linear constraints is another weak point. Michalewicz describes the problems of this method and its poor performance compared to other approaches; he conjectures that the the linear constraints prevent the system from moving closer to the optimum [37].

Another formulation for annealing penalties has been proposed by Carlson et al. [9]. The fitness of an individual is computed as

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) \cdot \exp(-M/T) \tag{2.12}$$

where M is a measure of constraint violation, and T the temperature which is a function of the running time of the algorithm. As the execution proceeds, T approaches zero. The method starts with a very high temperature, and hence the penalty factor $\exp(-M/T)$ is quite low, but it is steadily increased to eliminate infeasible solutions. T (the cooling schedule) is defined as

$$T = \frac{1}{\sqrt{t}}. \tag{2.13}$$

Here t indicates the last temperature used in the previous iteration.

Finally, Joines and Houck [24] experimented with the penalty function

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + \exp((tC)^{\alpha} SVC(\beta, \mathbf{x})) \tag{2.14}$$

where t is the generation number, $SVC(\beta, \mathbf{x})$ is defined by Equation 2.8, $C = 0.05$, and $\alpha = \beta = 1$.

Two weaknesses of Joines and Houck's method are that the results of the exponential function sometimes overflows because the constraints are not normalized, and that the definition of the constant C cannot be fully justified.

On the other hand, the method is easy to implement and does not distinguish the linear from the non-linear constraints.

2.2.4 Adaptive Penalties

Bean and Hadj-Alouane propose a PFM where the penalty parameters are updated for every generation according to information gathered from the population [6, 18]. Individuals are evaluated by

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) + \lambda(t) \left(\sum_{i=1}^{m} g_i^2(\mathbf{x}) + \sum_{i=1}^{p} |h_i(\mathbf{x})| \right). \tag{2.15}$$

Adaptive penalties differentiate between three cases: (1) the best individual in a number of previous generations has always been feasible; (2) the best individual in a number of previous generations has never been feasible; and (3) the best individual in a number of previous generations has sometimes been feasible and other times not.

The penalty factor $\lambda(t)$ is updated at every generation t in the following way:

$$\lambda(t+1) = \begin{cases} \lambda(t)/\beta_1 & \text{in case \#1} \\ \beta_2\lambda(t) & \text{in case \#2} \\ \lambda(t) & \text{otherwise} \end{cases} \tag{2.16}$$

In case #1 (the best individual in the last K generations has always been feasible), the penalty form $\lambda(t+1)$ decreases, while in case #2 the $\lambda(t+1)$ increases. In the last case, the penalty does not change. As Coello has pointed out, one drawback of Bean and Hadj-Alouane's approach is that it is not clear show to choose the generation gap K and the values of β_1 and β_2 [12].

2.2.5 Co-evolutionary Penalties

For problems with only inequality constraints, Coello has developed another approach known as the *self-adaptive* or *co-evolutionary* PFM [11]. It is based on the formula

$$\mathit{fitness}(\mathbf{x}) = f(\mathbf{x}) - (\mathit{coef} \cdot w_1 + \mathit{viol} \cdot w_2) \tag{2.17}$$

where w_1 and w_2 are two integer penalty factors, the sum of all the amounts by which the constraints are violated is given by

$$\mathit{coef} = \sum_{i=1}^{m} \max\{0, g_i(\mathbf{x})\}^2 \tag{2.18}$$

and the integer parameter *viol* takes the value 0 initially, and is incremented by 1 for each constraint that is violated.

Coello further proposes the use of two populations, P_1 and P_2 with sizes M_1 and M_2, respectively. The second population P_2 encodes the integer weights (w_1 and w_2) that are used to compute the fitness of the individuals in the first population P_1. In other words, one population evolves solutions (as in a conventional GA), while the other population evolves the penalty factors.

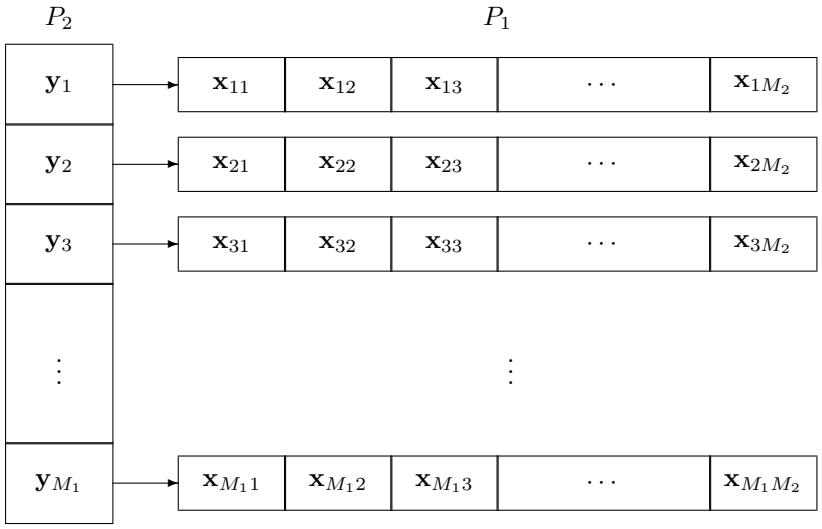

Figure 2.2: Graphical representation of the co-evolutionary PFM

The organization of this approach is illustrated in Figure 2.2. Note that there are in fact many P_1 populations; each individual in P_2 has its own corresponding P_1 subpopulation. To reach a solution, the following steps are repeated $gmax_2$ times:

1. For $i = 1, 2, \ldots, M_2$, the P_1 subpopulation that corresponds to individual $\mathbf{y}_i$ in P_2 is evolved for $gmax_1$ generations. The fitness function in Equation 2.17 is used with the weights w_1 and w_2 taken from the encoding $\mathbf{y}_i$.

2. The fitness for each individual $\mathbf{y}_i$ is evaluated using the formula

$$\mathit{fitness}'(\mathbf{y}_i) = \sum_{k=1}^{M_1} \left(\frac{F(\mathbf{x}_{ik})}{\mathit{count\_feasible}_i} \right) + \mathit{count\_feasible}_i \tag{2.19}$$

   where

$$F(\mathbf{x}) = \begin{cases} \mathit{fitness}(\mathbf{x}) & \text{if } \mathbf{x} \text{ is feasible} \\ 0 & \text{otherwise} \end{cases} \tag{2.20}$$

   and $\mathit{count\_feasible}_i$ is the number of feasible solutions in the subpopulation. If infeasible solutions are not excluded from the calculation of $\mathit{fitness}'(\mathbf{y})$, there is the danger that they will bias the GA towards regions where both w_1 and w_2 are low (and the penalty function becomes irrelevant).

Furthermore, the addition of the $count\_feasible_i$ term avoids stagnation in regions with few feasible solutions.

If any subpopulation contains only infeasible solutions, Steps 1 and 2 are repeated until every subpopulation has at least one feasible solution and every y_i has a fitness value.

3. The population P_2 is evolved for one generation.

As Coello himself points out, a difficulty of this approach is that it is not clear what the values of M_1, M_2, $gmax_1$, and $gmax_2$ should be. If these are not chosen carefully, the algorithm may not converge to a good solution. His experiments did show that the approach is more sensitive to changes in the parameters of P_1 than to changes in those for P_2.

2.3 The Simulated Annealing Method

The simulated annealing method was first developed by Kirkpatrick et al. [28], with the aim of exploiting the similarities between combinatorial optimization and statistical mechanics. It is, in essence, an extension of the Metropolis algorithm (which will be discussed in the next section) for the approximate numerical simulation of the behavior of a many-body system at a finite temperature.

First, however, we sketch the general setting of the simulated annealing algorithm. A simple model in statistical mechanics is a one-dimensional system that consists of N magnetic atoms. Each atom s_i points either to the north or to the south. These two possible states for an atom can be labeled with $+1$ and -1, respectively. Each adjacent pair of atoms s_i and s_{i+1} is coupled through a magnetic force J. Each configuration of all the s_j's is called a *micro-state* [8]. Associated with each configuration of the atoms is an energy E, given by

$$E = -\sum_{i=1}^{N-1} J s_i s_{i+1} = -J \sum_{i=1}^{N-1} s_i s_{i+1}$$

.

The least energy for a micro-state occurs when all N atoms point to the north, that is $s_i = +1$ for all i. In this case, the corresponding energy is $-(N-1)J$. On the other hand, the maximum energy for a micro-state occurs when all the atoms point south, when the corresponding energy is $+(N-1)J$. Hence, each adjacent pair of atoms contributes $\pm J$ to the total energy. The possible levels of energies are $-(N-1)J, -(N-3)J, -(N-5)J, \ldots, (N-5)J, (N-3)J, (N-1)J$. Each one of the energy levels is referred to as a *macro-state*.

At a given temperature, the atoms may change their positions in a restricted manner, and statistical mechanics can help us to describe the possible motions of the atoms. The probability that we find micro-state m with energy E_m at temperature T is given by

$$Pr(m) = \frac{\exp(-E_m/(k_B T))}{Z}$$

where $Z = \sum_q \exp(-E_q/(k_B T))$ defines the partition function, k_B is the Boltzmann constant, and q ranges over all the micro-states.

2.3.1 The Metropolis Algorithm

In 1953 Metropolis et al. introduced an algorithm (shown in Figure 2.3; it is also known as the Metropolis–Hastings algorithm) to provide an efficient simulation of a collection of atoms in equilibrium at a given temperature [20, 35]. Starting from an initial random state s_0 with energy E_0. The algorithm generates a sequence of states s_j with energies E_j, for $j = 1, 2, \ldots$, that tends toward the equilibrium state.

The Metropolis algorithm operates in a simple way: At state s with energy E, a random perturbation is introduced to yield a new state s', with energy E'. The change in the energy $\Delta E = E' - E$ is computed, and if $\Delta E < 0$ then the new state is accepted, otherwise it is accepted with probability $\exp(-\Delta E/(k_B T))$. If the new state it accepted, it replaces the previous state.

2.3.2 The Kirkpatrick Algorithm

The simulated annealing method for optimization problems as described by Kirkpatrick et al. imitates the physical process of annealing. This is accomplished by applying a chain of Metropolis processes at decreasing temperatures subject to some cooling schedule. In each iteration, at the given temperature the value of the objective function corresponds the energy of the system at that temperature, while the minimum value of the objective function corresponds to the energy of the equilibrium state at that temperature. (See Figure 2.4.)

Consider an unconstrained optimization problem with objective function

$$\min_{\mathbf{x} \in \mathbb{R}^n} f(\mathbf{x}) \tag{2.21}$$

The simulated annealing algorithm starts by choosing the initial temperature $T_0 > 0$, a temperature decreasing factor ΔT and a random initial configuration $\mathbf{x}_0 \in \mathbb{R}^n$. It applies the Metropolis algorithm

```
METROPOLIS (T)
1   choose a random initial state s with energy E
2   s' ← METROPOLISCORE(T, s, E)

METROPOLISCORE (T, s_0, E_0)
1   i ← 0
2   repeat
3     i ← i + 1
4     perturb the state s_i randomly to obtain a state s with energy E
5     compute the change in the energy ΔE ← E − E_{i−1}
6     if ΔE < 0 then
7       s_i ← s ; E_i ← E
8     else
9       generate a random number r ∼ U[0, 1] following the uniform distribution
10      if r ≤ exp(−ΔE/(k_B T)) then
11        s_i ← s ; E_i ← E
12      else
13        s_i ← s_{i−1} ; E_i ← E_{i−1}
14      endif
15    endif
16  until no perturbation decreases the energy
17  return s_i
```

Figure 2.3: The METROPOLIS algorithm

```
SIMULATEDANNEALING (f, T, ΔT)
1   choose a random initial state x_0 ∈ R^n
2   k ← 0
3   while T > ε do
4     k ← k + 1
5     x_k ← METROPOLISCORE(T, x_{k−1}, f(x_{k−1}))
6     T ← (1 − ΔT)T
7   endwhile
8   return (x_k, f(x_k))
```

Figure 2.4: The Kirkpatrick et al. simulated annealing algorithm

with $E_0 = f(\mathbf{x}_0)$ to obtain an approximate solution $\mathbf{x}_1$ and an objective function value $f(\mathbf{x}_1)$. State $\mathbf{x}_1$ is known as the ground state at temperature T_0. The temperature is then decreased by $\Delta T \cdot T_0$ and yields a lowered temperature $T_1 = (1 - \Delta T)T_0$. The Metropolis algorithm is applied again, now with initial state $\mathbf{x}_1$ at the temperature T_1 and a second approximate solution $\mathbf{x}_2$ for the minimizer of f is obtained. The process continues until the final temperature $T_N \leq \epsilon$ for some given limit ϵ is achieved (after iteration N). ϵ is known as the freezing temperature. At the end, the resulting ground state $\mathbf{x}_N$ is an approximate global minimizer of f.

Chapter 3

Implementation

3.1 Introduction

The purpose of this chapter is to explain some of the details of our application of GAs (in combination with the PFMs discussed in Chapter 2) and simulating annealing (SA) to the MCOP problem. We begin with an overview of the problem, and then discuss the algorithms at the hand of low-level pseudocode.

Given is a network (V, L), where $V \subset \mathbb{Z}^+$ is the set of the network nodes (routers, servers), and $L \subset V \times V$ is the set of the network links. For each link $\ell \in L$ we are also given the total bandwidth C_ℓ of the link. Furthermore, the input contains a set of demands in the form of source-destination pairs (s, d) where $s, d \in L$, along with the bandwidth $B_{s,d}$ required to transport the packets from their source to their destination along some path. Associated with each such path is a cost computed by a given cost function F.

In this work, we consider two different versions of the cost metric, given by the functions

$$F_1 = \sum_{\ell \in L} \frac{f_\ell}{C_\ell - f_\ell} \tag{3.1}$$

$$F_2 = \sum_{\ell \in L} \frac{1}{C_\ell - f_\ell} \tag{3.2}$$

subject to the constraint

$$f_\ell \leq C_\ell \quad \text{for all } \ell \in L \tag{3.3}$$

where L is the set of network links as before, and f_ℓ is the total number of source-destination pairs that make use of link ℓ. Metric F_1 in Equation 3.1 is known as the total queueing weighted mean link delay on the network, and metric F_2 in Equation 3.2 is the total queuing mean link delay on the network.

Computing the shortest paths on a given network from some source to some destination usually is done by using a shortest path algorithm such as Dijkstra's. The shortest path algorithms need to know the cost (weight) associated with each link on the network. These links weights are administrated by the network operator. Some of the algorithms require that the user specifies a number of parameters. Along with the pseudocode below we also give the values that we used for these parameters. In some cases the values are based on published studies; in other cases we have chosen values that were found to work well in practice.

3.2 The GA Framework

We now sketch the framework of our GA which is used, together with PFMs described in the subsequent sections, to find the link weights. In our GA implementation, a genome encodes the link weights as integers. In other words, each genome is a vector of length $|L|$, and the i-th entry of the genome is the weight associated with link ℓ_i for $i = 1, 2, \ldots, |L|$. The absolute magnitude of the weights is not important, only their relative values. Here, the weights are chosen to lie between 0 and 50.

The implementation of the GA is shown in Figure 3.1, and consists of the routines PerformSimulation, PerformGenetic, GenerateRandomPopulation, and EvaluateFitness. The following variables play an important role in all of the routines:

- *Objective* is an integer flag and is either 1 or 2.
- The genomes of the current population is stored in the global variable *Population*. The i-th member *Population*[i] is a genome, and *Population*[i][j] is the weight of link ℓ_j as encoded by the genome. The number of genomes in the population is stored in *PopulationSize*, and the number of links (i.e., the value of $|L|$) is stored in *NumberOfLinks*. *Fitness* is another global variable that stores the fitness of the individuals in the current population. Genome *Population*[i] has fitness *Fitness*[i].
- *CurrentGeneration* is an integer counter that keeps track of how many populations have been generated. *NumberOfGenerations* is the number of generations for which the algorithm is allowed to evolve.
- *CrossoverProbability* and *MutationProbability* determine, as their names imply, the probability of crossover and mutation, respectively.

The *Network* variable stores the input network and is read from a file in line 1 of PerformSimulation. Each link ℓ of the network is a record with the following fields:

```
PerformSimulation (NetworkFile, DemandsFile, Objective)
1   Network ← read from NetworkFile
2   Demands ← read from DemandsFile
3   for d ∈ Demands do
4     w ← PerformGenetic(d, Objective)
5     P ← Dijkstra(d, w)
6     P.PathDelay ← 0.0
7     for ℓ ∈ P do
8       ℓ.Bandwidth ← ℓ.Bandwidth + d.Bandwidth
9       ℓ.NumberOfFlows ← ℓ.NumberOfFlows + 1
10      if Objective = 1 then
11        ℓ.LinkDelay ← ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
12      else
13        ℓ.LinkDelay ← 1/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
14      endif
15      P.PathDelay ← P.PathDelay + ℓ.LinkDelay
16    endfor
17    update statistics with P
18  endfor
19  calculate and display the statistics MaximumLinkDelay, AveragePathDelay,
      MaximumPathLength, AveragePathLength, and CPUTime

PerformGenetic (d, Objective)
1   GenerateRandomPopulation()
2   CurrentGeneration ← 1
3   while CurrentGeneration ≤ NumberOfGenerations do
4     EvaluateFitness(d, Objective)
5     Select()
6     Crossover(CrossoverProbability)
7     Mutate(MutationProbability)
8     CurrentGeneration ← CurrentGeneration + 1
9   endwhile
10  return GetFittest()

GenerateRandomPopulation ()
1   for i ← 1 . . . PopulationSize do
2     for j ← 1 . . . NumberOfLinks do
3       Population[i][j] ← Round(10 · Random())
4     endfor
5   endfor

EvaluateFitness (d, Objective)
1   for i ← 1 . . . PopulationSize do
2     P = Dijkstra(d, Population[i])
3     SumOfDelays ← 0.0
4     for ℓ ∈ P do
5       ℓ.Bandwidth ← ℓ.Bandwidth + d.Bandwidth
6       ℓ.NumberOfFlows ← ℓ.NumberOfFlows + 1
7       ℓ.LinkDelay ← ℓ.LinkDelay + ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
8       SumOfDelays ← SumOfDelays + ComputePenalizedObjective(ℓ, Objective)
9     endfor
10    Fitness[i] ← exp(−SumOfDelays)
11  endfor
```

Figure 3.1: The main routines of the GA

- $\ell.TotalBandwidth$ is the total bandwidth available on the link, and corresponds to C_ℓ. It is part of the input and is not changed during the run of the algorithm.
- $\ell.Bandwidth$ is the bandwidth currently in use on link ℓ. It is initially 0 but is steadily increased whenever a source-destination path that makes use of the link is computed by the GA.
- $\ell.NumberOfFlows$ is the number of source-destination paths that use link ℓ. It is initially 0.
- $\ell.LinkDelay$ is the value of one term of either Equation 3.2 or Equation 3.1, depending on the value of *Objective*.

The *Demands* variable stores the source-destination pairs and is read from an input file in line 2 of PerformSimulation. Each demand d is a record with three fields: $d.Source$ and $d.Destination$ are the source and destination of the demand; they are not used directly in the algorithm, but are needed inside the Dijkstra subroutine. The third field, $d.Bandwidth$ is used in the computation of the fitness function (in line 5 of EvaluateFitness) and the calculation of the overall performance of the GA (in line 8 of PerformSimulation).

Two more routines need to be mentioned. GetFittest (used in line 10 of PerformGenetic) returns the fittest genome found during the run of the GA. It is important to keep track of this individual, since as Rudolph has shown, the GA cannot converge to the global optimum value without this mechanism [44].

The Dijkstra(d, w) routine finds the shortest path between two nodes. The source and destination nodes are determined by the demand parameter d, and the w parameter is the link weights.

Lastly, pseudocode for the selection, crossover, and mutation operations is shown in Figure 3.2; their implementation is straightforward.

3.3 Implementation Using Non-Stationary Penalty Methods

The stationary PFMs are postponed until the next section; we deal with non-stationary PFMs first. The penalty function methods differ from each other in the way the penalized objective function is evaluated. Each penalty function method implements its own version of the ComputePenalizedObjective routine.

It is therefore necessary to now specify exactly which of the techniques described in Chapter 2 were selected and what parameters are used, and also to define how the ComputePenalizedObjective routine computes the fitness in each case.

```
SELECT ()
1   SumOfFitness ← 0.0
2   for i ← 1 ... PopulationSize do
3     SumOfFitness ← SumOfFitness + Fitness[i]
4   endfor
5   for j ← 1 ... PopulationSize do
6     r ← RANDOM() · SumOfFitness
7     WinningFitness ← 0.0 ; i ← 0
8     while WinningFitness < r do
9       WinningFitness ← WinningFitness + Fitness[i]
10      i ← i + 1
11    endwhile
12    NewPopulation[j] ← Population[i]
13  endfor
14  Population ← NewPopulation

CROSSOVER (CrossoverProbability)
1   I1 ← 1 ; I2 ← 2
2   while I2 ≤ PopulationSize do
3     if RANDOM() < CrossoverProbability then
4       p1 ← ROUND(PopulationSize · RANDOM())
5       p2 ← ROUND(PopulationSize · RANDOM())
6       if p1 > p2 then SWAP(p1, p2) endif
7       for i ← p1 ... p2 do
8         SWAP(Population[I1][i], Population[I2][i])
9       endfor
10    endif
11    I1 ← I1 + 2 ; I2 ← I2 + 2
12  endwhile

MUTATE (MutationProbability)
1   for i ← 1 ... PopulationSize do
2     for j ← 1 ... NumberOfLinks do
3       if RANDOM() < MutationProbability then
4         Population[i][j] ← ROUND(10 · RANDOM())
5       endif
6     endfor
7   endfor
```

Figure 3.2: The SELECT, CROSSOVER, and MUTATE routines

3.3.1 Dynamic Penalties

The dynamic penalty method of Joines and Houck [24] described at the start of Section 2.2.2 was selected because of its small number of parameters. For this problem we choose $\alpha = 2$, $\beta = 2$, and $C = 0.5$. The code is shown in Figure 3.3.

The fitness function for the weighted mean delay is described by the formula

$$F_1 = \sum_{\ell \in L} \left(\frac{f_\ell}{C_\ell - f_\ell} + (tC)^\alpha D_\ell^\beta(\mathbf{x}) \right) \tag{3.4}$$

and the mean delay objective function is described by the formula

$$F_2 = \sum_{\ell \in L} \left(\frac{1}{C_\ell - f_\ell} + (tC)^\alpha D_\ell^\beta(\mathbf{x}) \right) \tag{3.5}$$

where

$$D_\ell(\mathbf{x}) = \max\{0, f_\ell - C_\ell\}.$$

3.3.2 Annealing Penalties

The method of Michalewicz and Attia [37, 39] was selected as an annealing PFM, because — as in the case of the dynamic penalties — it is the simplest with the least number of parameters. The code appears in Figure 3.4.

The fitness function for the weighted mean delay is described by the formula

$$F_1 = \sum_{\ell \in L} \left(\frac{f_\ell}{C_\ell - f_\ell} + \frac{1}{2\tau} \max\{0, f_\ell - C_\ell\} \right) \tag{3.6}$$

and the mean delay objective function is described by the formula

$$F_2 = \sum_{\ell \in L} \left(\frac{1}{C_\ell - f_\ell} + \frac{1}{2\tau} \max\{0, f_\ell - C_\ell\} \right). \tag{3.7}$$

We choose $\tau = \frac{CurrentGeneration+1}{100}$.

3.3.3 Adaptive Penalties

The Bean and Hadj-Alouane method [6, 18] was selected as an adaptive PFM. (It is the only adaptive penalty method presented in Section 2.2.4.)

In this method, the weighted mean delay objective function takes the form:

$$F_1 = \sum_{\ell \in L} \left(\frac{f_\ell}{C_\ell - f_\ell} + \lambda(t)(f_\ell - C_\ell)^2 \right) \tag{3.8}$$

```
ComputePenalizedObjective (ℓ, Objective)
1   α ← 2 ; β ← 2 ; c ← 0.5
2   if ℓ.NumberOfFlows ≤ ℓ.TotalBandwidth then
3      D ← 0
4   else
5      D ← ℓ.TotalBandwidth − ℓ.NumberOfFlows
6   endif
7   if Objective = 1 then
8      LinkDelay ← ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
9   else if Objective = 2 then
10     LinkDelay ← 1/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
11  endif
12  PenalizedCost ← PenalizedCost + LinkDelay + (c · CurrentGeneration)^α · D^β)
13  return PenalizedCost
```

Figure 3.3: The ComputePenalizedObjective function for the dynamic PFM

```
ComputePenalizedObjective (ℓ, Objective)
1   if Objective = 1 then
2      LinkCost ← ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
3   else if Objective = 2 then
4      LinkCost ← 1/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
5   endif
6   PenalizedCost ← PenalizedCost + LinkCost + max{0, ℓ.NumberOfFlows − ℓ.TotalBandwidth}/(2τ)
7   return PenalizedCost
```

Figure 3.4: The ComputePenalizedObjective function for the annealing PFM

and the mean delay objective function takes the form:

$$F_2 = \sum_{\ell \in L} \left(\frac{1}{C_\ell - f_\ell} + \lambda(t)(f_\ell - C_\ell)^2 \right). \tag{3.9}$$

The implementation of the adaptive penalty function method differs from the implementations of the dynamic and annealing penalty methods. In the body of the routine PERFORMGENETIC we initialize a boolean vector *FittestIsFeasible* of dimension *NumberOfGenerations* and an integer variable *GenerationGap*. In each generation i, if the fittest individual is feasible, then *FittestIsFeasible*$[i]$ is set to true, and otherwise it is set to false.

In the first *GenerationGap* generations, the penalty factor λ remains constant. After that, it may change according to changes in the feasibily or infeasibility of the best individual of the past *GenerationGap* generations.

Three boolean variables are used to determine whether the best individual had changed its status from feasible to infeasible or vice versa in the past *GenerationGap* generations. The first variable (*OldFeasible*) becomes true if the best individual in generation $CurrentGeneration - GenerationGap + 1$ is feasible, and false otherwise. The second boolean variable (*Unchanged*), indicates whether the best individual changes its status from feasible to infeasible or vice versa. This variable is initially true. The third variable (*Feasible*) is an iterator that runs through the generations from $CurrentGeneration - GenerationGap + 1$ to *CurrentGeneration*. This variable initially (in generation $CurrentGeneration - GenerationGap + 1$) takes the same value as *OldFeasible*. In the next generations up to to *CurrentGeneration* it is compared to *OldFeasible*. If the variables are equal to it in all the generations under consideration, variable *Unchanged* remains true, otherwise it changes to false. Finally, if *Unchanged* is true, this means that the best individual has remained feasible or infeasible through all the past *GenerationGap* generations, depending on whether *OldFeasible* is true or false, respectively. If the variable *Unchanged* is false, then a change in the status (from feasible to infeasible or vice versa) of the best individual has occurred.

The code for the modified versions of PERFORMGENETIC and COMPUTEPENALIZEDOBJECTIVE are shown in Figure 3.5. We choose $\beta_1 = \beta_2 = 2$.

3.3.4 Co-evolutionary Penalties

Coello's co-evolutionary PFM [11] has already been described in considerable detail in Section 2.2.5. Our implementation follows the description faithfully. Because it is significantly more complicated than the other PFMs, we omit its pseucode here for the sake of brevity.

```
PerformGenetic (d, Objective)
1   GenerateRandomPopulation()
2   CurrentGeneration ← 1
3   while CurrentGeneration ≤ NumberOfGenerations do
4     if CurrentGeneration > GenerationGap then
5       Feasible ← OldFeasible ← FittestIsFeasible[CurrentGeneration − GenerationGap + 1]
6       Unchanged ← true
7       for i ← CurrentGeneration − GenerationGap + 2 do
8         Feasible ← FittestIsFeasible[i]
9         if Feasible ≠ OldFeasible then Unchanged ← false endif
10      endfor
11      if (OldFeasible) ∧ (Unchanged) then λ ← λ/β1 endif
12      if (¬OldFeasible) ∧ (Unchanged) then λ ← λ · β2 endif
13    endif
14    EvaluateFitness(d, Objective)
15    Select()
16    Crossover(CrossoverProbability)
17    Mutate(MutationProbability)
18    CurrentGeneration ← CurrentGeneration + 1
19  endwhile
20  return GetFittest()

ComputePenalizedObjective (ℓ, Objective)
1   if Objective = 1 then
2     LinkCost ← ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
3   else if Objective = 2 then
4     LinkCost ← 1/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
5   endif
6   PenalizedCost ← PenalizedCost + LinkCost + λ(ℓ.NumberOfFlows − ℓ.TotalBandwidth)
7   return PenalizedCost
```

Figure 3.5: The PerformGenetic and ComputePenalizedObject functions for the adaptive PFM

The weighted mean delay objective function takes the form:

$$F_1 = \sum_{\ell \in L} \left(\frac{f_\ell}{C_\ell - f_\ell} + (coef \cdot w_1 + viol \cdot w_2) \right) \tag{3.10}$$

and the mean delay objective function becomes:

$$F_2 = \sum_{\ell \in L} \left(\frac{1}{C_\ell - f_\ell} + (coef \cdot w_1 + viol \cdot w_2) \right). \tag{3.11}$$

We choose $Population1Size = 35$, $Population2Size = 5$, $GenerationMax1 = 10 = GenerationMax2$, and $coef = viol = 100$.

3.4 Implementation Using Stationary Penalty Methods

Section 2.2.1 discussed several static PFMs, one of which we discuss below. However, these are not the only choices for static penalties. In this section we explore the effect of fixing the penalty factors of the various penalty function methods that we used in the last section. Even though we have presented these methods as non-stationary, once we fix their penalty factors, they automatically turn into stationary methods.

Our aim is to find out how this can affect the performance of a given network and demand schedule. We expect this change to result in faster CPU time for computing the optimal paths, in particular for a large number of generations, because the computation of the penalty factors is performed only once, on starting the simulations.

3.4.1 Penalties On the Constraints Violations

The dynamic, annealing and adaptive penalty methods place penalties only on the amount by which the constraints are violated. For a larger constraint violation, a high penalty factor is used for a specific number of generations. After that the penalty factor may be increased or decreased depending on the method used. In dynamic penalty methods, the penalty factor increases with the evolution of time. In annealing penalty methods, the temperature is decreased with the evolution of time. In the adaptive penalty methods, the penalty factors depend on the status of the best individual in the last k generations. If it remains feasible in the last k generations, the penalty factor decreases, if it remains unfeasible, the penalty factor increases, and if it is feasible in some of the last k generations and infeasible for the rest, no change occurs in the penalty factor.

The above three scenarios give rise to constant penalty factors. Each can correspond on one of the three penalty methods.

```
ComputePenalizedObjective (ℓ, Objective)
1  if Objective = 1 then
2     LinkCost ← ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
3  endif
4  if Objective = 2 then
5     LinkCost ← 1/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
6  endif
7  PenalizedCost ← PenalizedCost + LinkCost + max{0, √((ℓ.NumberOfFlows − ℓ.TotalBandwidth)²)}
8  return PenalizedCost
```

Figure 3.6: The ComputePenalizedObjective function for the static PFM

Static Penalties

The fitness functions for the weighted mean delay is described by the formula:

$$F_1 = \sum_{\ell \in L} \left(\frac{f_\ell}{C_\ell - f_\ell} + \sqrt{H(f_\ell - C_\ell) \cdot (f_\ell - C_\ell)^2} \right) \tag{3.12}$$

and the mean delay by the formula:

$$F_1 = \sum_{\ell \in L}^{L} \left(\frac{1}{C_\ell - f_\ell} + \sqrt{H(f_\ell - C_\ell) \cdot (f_\ell - C_\ell)^2} \right) \tag{3.13}$$

The code for this PFM is shown in Figure 3.6.

Fixing Time in the Dynamic Penalties

By fixing t in Equation(3.4) as $t = T$, the coefficient $(tC)^\alpha = (TC)^\alpha = K$ is a constant value. The weighted mean delay objective function then takes the form

$$F_1 = \sum_{\ell \in L} \left(\frac{f_\ell}{C_\ell - f_\ell} + KH(f_\ell - C_\ell)^\beta \right)$$

and the mean delay objective function takes the form

$$F_2 = \sum_{\ell \in L} \left(\frac{1}{C_\ell - f_\ell} + KH(f_\ell - C_\ell)^\beta \right)$$

where H is the Heaviside function defined by

$$H(t) = \begin{cases} 0 & t \leq 0 \\ 1 & t > 0 \end{cases}$$

We choose $K = 100$ and $\beta = 2$.

Fixing Temperature in the Annealing Penalties

By setting $\frac{1}{2\tau} = T$ to be a constant, the penalized weighted mean delay objective function Equation(3.6) becomes

$$F_1 = \sum_{\ell \in L}^{L} \left(\frac{f_\ell}{C_\ell - f_\ell} + T \cdot \max[0, f_\ell - C_\ell] \right)$$

There are two cases to be considered

1. using a low temperature T, and
2. using a high temperature T.

Setting the Generation Gap Equal to the Maximum Number of Generations on the Adaptive Penalties

The first possible change on the penalty factor $\lambda(t)$, when using the adaptive penalty methods, can happen only when the number of the current iteration is equal to the generation gap. By setting the generation gap to be *NumberOfGenerations + 1* or any higher value, no change in λ can happen. Hence, λ remains constant in all the generations, with $\lambda(t) = \lambda_0$, where λ_0 is the initial penalty factor.

The weighted mean delay penalized objective function Equation(3.8) becomes

$$F_1 = \sum_{\ell \in L}^{L} \left(\frac{f_\ell}{C_\ell - f_\ell} + \lambda_0 (f_\ell - C_\ell)^2 \right)$$

3.4.2 Penalties On both the Constraints Violations and the Number of the Violated Constraints

Setting $GenerationMax2 = 1$ and $PopulationSize2 = 1$ in the Co-Evolutionary Penalties

If both the two parameters $GenerationMax2$ and $PopulationSize2$ are set to equal one, then the two random weights w_1 and w_2 will be generated at the begining and will not be allowed to evolve. The result of this setup is to keep the parameters w_1 and w_2 constant throughout the simulation.

The weighted mean delay objective function takes the formula:

$$F_1 = \sum_{\ell \in L}^{L} \left(\frac{f_\ell}{C_\ell - f_\ell} - (w_1(0) \times coef + w_2(0) \times viol) \right) \tag{3.14}$$

The mean delay penalized objective function takes the form:

$$F_2 = \sum_{\ell \in L}^{L} \left(\frac{1}{C_\ell - f_\ell} - (w_1(0) \times coef + w_2(0) \times viol) \right) \tag{3.15}$$

where $w_1(0)$ and $w_2(0)$ are the penalty factors that have been generated by the single individual in population 2 at the initial time.

3.5 Implementation Using Simulated Annealing

In this section we implement the simulated annealing algorithm to determine the optimal weights for the network links, which minimize the link delay when solving both the mean delay and the weighted mean delay optimization problems as defined in Equations 3.1, 3.2 and 3.3.

Here, we replace the routine **PerformGenetic** and the dependent routines which we implemented in earlier sections, by a routine **PerformSimulatedAnnealing**. The rest of the routines remain as is. **PerformSimulatedAnnealing** is actually broken down into three separate subroutines as in Figure 3.7, but we shall talk about all three routines as one. To initiate the correspondence between the metal annealing problem and finding the optimal link weights when solving both the mean delay and the weighted mean delay optimization problems in traffic networks, we map the state configuration of the metal annealing problem to the link weights. We also map minimizing the energy to minimizing the weighted mean delay (or mean delay) objective function.

The **PerformSimulatedAnnealing** routine receives a demand record d which contains the three fields: source, destination, and bandwidth. It starts with an initial temperature T_0 and a feasible initial random set of link weights for the network. The temperature decreases with a percentage ΔT of the temperature T. The amount ΔT is determined beforehand, and so is the freezing temperature ε.

The process starts at the highest temperature $T = T_0$. We copy the link attributes (*TotalBandwidth*, *UsedBandwidth*, *NumberOfFlow*, and *LinkDelay*) to a temporary structure which shall be used for the computations, before physical changes are made to the original links. We compute the shortest path P from the demand-source to the demand-destination, using the Dijkstra algorithm. Then the attributes of each link ℓ that belongs to the shortest path P are changed. This happens by adding the demand-bandwidth to the link bandwidth, incrementing the link's number of flows by 1 and computing the link delay. If the number of flows at the link ℓ is greater than the total link bandwidth, then the packets will never be delivered through that link. Hence the link delay will be infinite. Otherwise, the link delay can be computed by using either the weighted mean delay or the mean delay metrics. The total cost using the weights W is the value of the path delay.

Then, the **PerformSimulatedAnnealing** routine chooses one link at random and increases (decreases) its weight by one. According to the new link weights, the shortest paths are computed again using the Dijkstra algorithm and then the new value for the maximum link delay is computed and stored in another variable $Cost_2$. We compute the change in the value of the objective function $\Delta C = Cost_2 - Cost_1$. If $\Delta C < 0$, then the **PerformSimulatedAnnealing** routine accepts the new set of link weights. Otherwise it accepts it with a probability $e^{-\frac{y}{T}}$. The whole process of perturbing the link weights, computing the resulting path and path delay, and accepting or rejecting the new link weights configuration is repeated for a predetermined number of times $MaximumLoop$. This number should be large enough to give each link on the network the opportunity to switch its weight.

Lastly, the temperature is decreased to $T = (1 - \Delta T)T$ and the whole process is repeated again. The loop terminates when $T < \varepsilon$, the freezing temperature. Finally, the **PerformSimulatedAnnealing** routine returns the optimal link weights to the **PerformSimulation** routine in Figure 3.1.

```
PerformSimulatedAnnealing (d)
1   T ← 10^3
2   ΔT ← 0.08
3   for i ← 1 ... NumberOfLinks do W[i] ← Random() endfor
4   ε ← 1
5   while T ≥ ε do
6      Weights ← GroundState(d, W, T)
7      T ← (1 − ΔT)T
8   endwhile
9   return Weights

GroundState (d, W, T)
1   NewWeights ← W
2   Cost_1 ← ComputeCost(d, W)
3   for i ← 1 ... MaximumLoop do
4      RandomLink ← Round(Random() × NumberOfLinks)
5      W[RandomLink] ← W[RandomLink] + 1
6      Cost_2 ← ComputeCost(d, W)
7      ΔC ← Cost_2 − Cost_1
8      if ΔC < 0 then
9         NewWeights ← W
10        Cost_1 ← Cost_2
11     else if Random() < exp(−ΔC/T) then
12        NewWeights ← W
13        Cost_1 ← Cost_2
14     endif
15  endfor
16  return NewWeights

ComputeCost (d, W)
1   P ← Dijkstra(d, W)
2   PathDelay ← 0.0
3   for ℓ ∈ P do
4      ℓ.UsedBandwidth ← ℓ.UsedBandwidth + d.UsedBandwidth
5      ℓ.NumberOfFlow ← ℓ.NumberOfFlows + 1
6      if Objective = 1 then
7         ℓ.LinkDelay ← ℓ.NumberOfFlows/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
8      else if Objective = 2 then
9         ℓ.LinkDelay ← 1/(ℓ.TotalBandwidth − ℓ.NumberOfFlows)
10     endif
11     if ℓ.TotalBandwidth < ℓ.NumberOfFlows then ℓ.LinkDelay ← ∞ endif
12     PathDelay ← PathDelay + ℓ.LinkDelay
13  endfor
14  return PathDelay
```

Figure 3.7: PerformSimulatedAnnealing and related routines

Chapter 4

Performance Evaluation

To compare the performances of the genetic algorithms and the simulated annealing method that we have implemented, we used four networks. The first two networks, *USA1* and *USA2*, both contain 23 nodes and 76 links, but the link configurations are different and so are the traffic demands. For the *USA1* there are 106 traffic demands, and for the *USA2* there are 231. The third network, which we shall call *Europe*, contains 29 nodes, 92 links, and 100 traffic demands. The last network, *PAREN*, is based on the Pan-Africa Research and Education Network; it contains 31 nodes, 128 links, and 430 traffic demands. All the test networks are fictitious and had been generated based on the work by Arvidsson et al. [1]. The construction of each network model is based on the number of habitants in each city and also on the city area. Illustrative graphs for the three network models are shown in figures 4.1, 4.2 and 4.3 below.

To obtain optimal values for the parameters associated with the adaptive, annealing and dynamic penalty function methods, for each penalty method and each parameter we chose a range for that parameter and fixed the values of the other parameters. Then we ran simulations to determine a value for that parameter which optimizes the performance of the penalty method. For the parameter β in both the adaptive and dynamic penalty function methods we tried the values $0.5, 0.75, 1, \ldots, 2$. For the parameters λ, τ and C in the adaptive, annealing and dynamic penalty methods, we tried the values $10^j;\ j = 1, \ldots, 7$. Then, we chose the best value for the parameter in the chosen range. Those optimal values of the parameters in the mean delay and weighted mean delay models are shown in tables 4.1-4.3 below.

In figures 4.4 and 4.5 we plot the average values for the maximum link delay, average link delay, maximum path length and average path length against the penalty factor and the exponent β in the

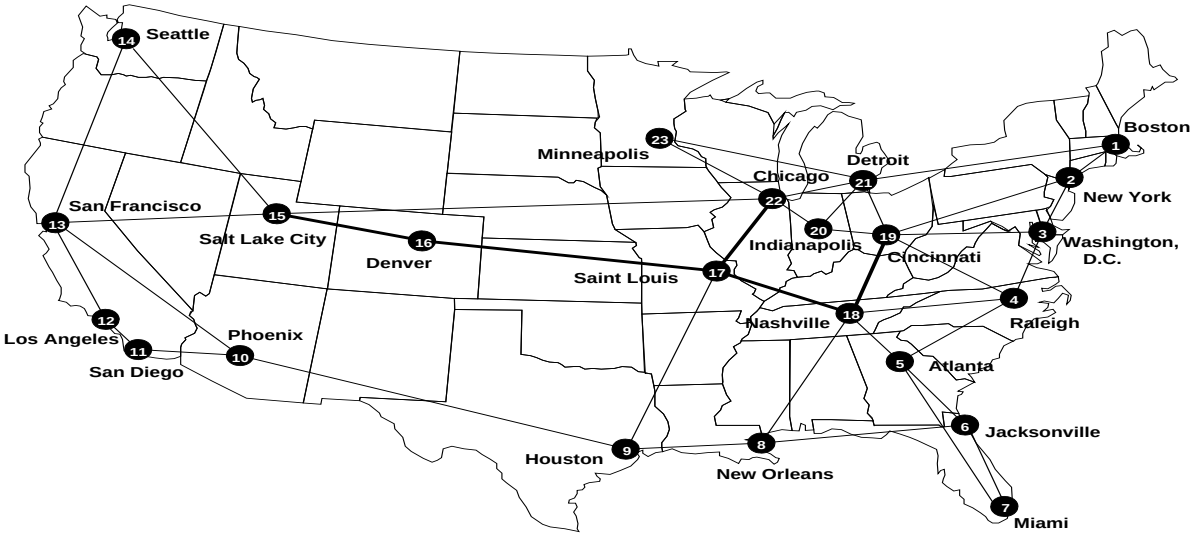

Figure 4.1: USA network.

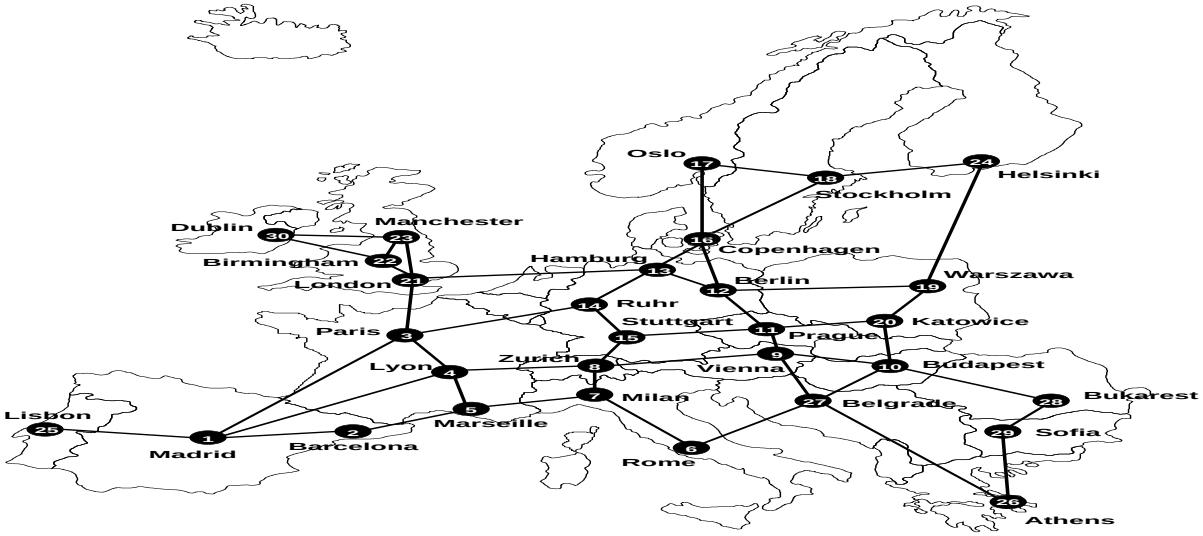

Figure 4.2: Europe network.

Table 4.1: Optimal parameter values in adaptive penalty method

	mean delay				weighted mean delay			
	USA1	USA2	Europe	PAREN	USA1	USA2	Europe	PAREN
β	0.5	1.25	1.25	1.0	1.75	2.0	1.75	0.5
λ	100000	1000	1000	100	1000	100000	1000	100

Table 4.2: Optimal parameter in annealing penalty method

	mean delay				weighted mean delay			
	USA1	USA2	Europe	PAREN	USA1	USA2	Europe	PAREN
τ	100	100	100	100	100	100	100	100

Table 4.3: Optimal parameter values in dynamic penalty method

	mean delay				weighted mean delay			
	USA1	USA2	Europe	PAREN	USA1	USA2	Europe	PAREN
β	1	2.0	1.5	1.5	1.0	1.0	1.5	1.5
C	100	10	100000	100000	10	100	1000000	10

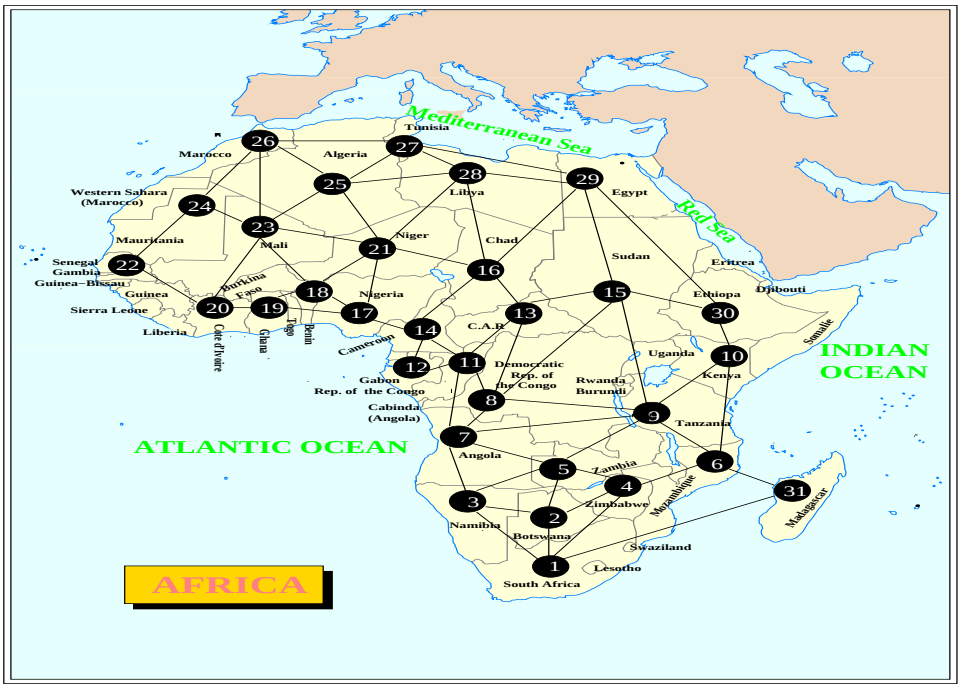

Figure 4.3: Africa network.

PAREN (annealing penalty function method applied to the weighted mean delay) and USA2 (dynamic penalty function method applied to the mean delay), respectively.

The experiments were performed on a workstation with a dual core with 2.4 GHz processor and 3.0 gigabyes of memory. This is important, because one of the performance measures is time consumption. We also reported the maximum link delay, maximum path delay, maximum path length, and the average path length.

Through all the figures that will follow in the next sections, the networks 1, 2, 3 and 4 will denote the USA1, USA2, Europe and PAREN, respectively.

The results in this chapter are depicted graphically, but exactly the same information can be found in Appendix A in tabular form.

In sections 4.1 and 4.2 we summarize the results obtained for the weighted mean delay and the mean delay metrics using the adaptive, annealing, dynamic, co-evolutionary, static penalty function methods and the simulated annealing method. In section 4.3, we fix some of the penalty factors in the adaptive, annealing, dynamic and co-evolutionary penalty function methods obtaining stationary methods. Then, we compare between the performances of the stationary and non-stationary penalty function methods. Finally, in section 4.4 we summarize the results obtained by our simulations.

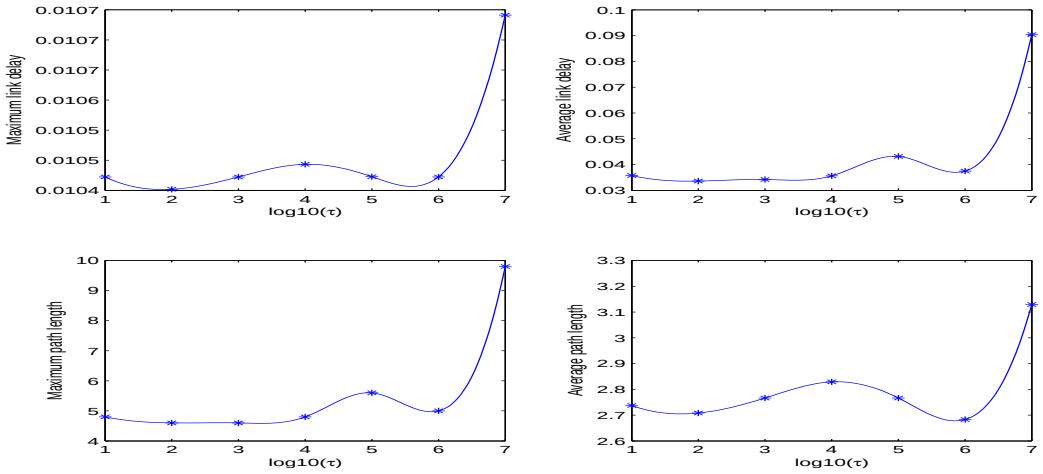

Figure 4.4: Results obtained by applying the annealing penalty function method to PAREN, for different values of τ in the weighted mean delay metric.

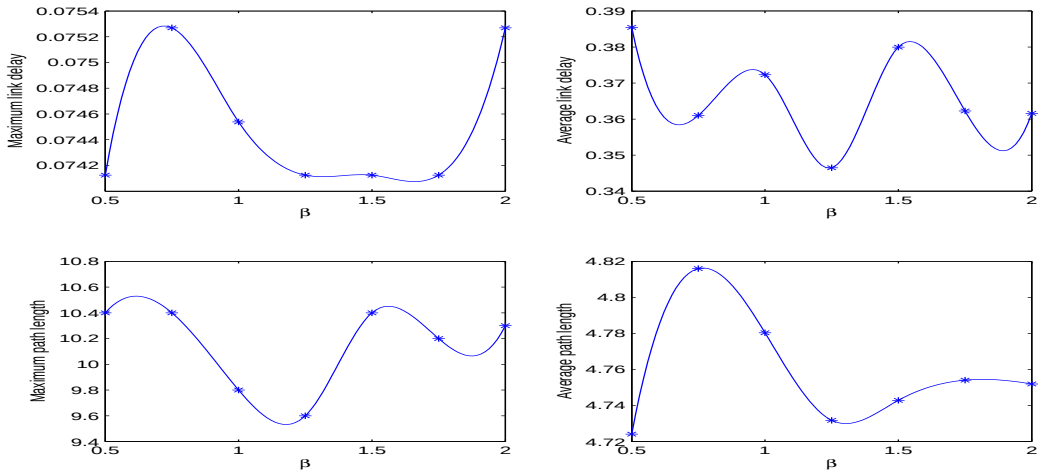

Figure 4.5: Results obtained by applying the dynamic penalty function method to the USA2 network, for different values of β in the mean delay metric.

4.1 The Weighted Mean Delay Metric

Tables A.1 and A.2 show the results obtained by the dynamic, annealing, adaptive, co-evolutionary, and static penalty methods, and by simulated annealing. The first column gives the network name, followed by the maximum link delay (in column MLD), maximum path delay (column MPD), maximum path length (MPL), average path length (APL) and the CPU time in seconds in the last column. The same information (apart from the standard deviation) is depicted graphically in Figure 4.6.

Even though the methods are stochastic, we only report the results of a single, typical run. In all the GA methods, except the co-evolutionary penalty method, we used a population size of 100 and the population evolved for 100 generations. For the co-evolutionary penalty method, we use 20 individuals in the first population and 5 individuals in the second population. In all the penalty function methods, the crossover probability was 0.40 and the mutation probability 0.05. For the SA method, we used a value of $MaximumLoop = 1000$.

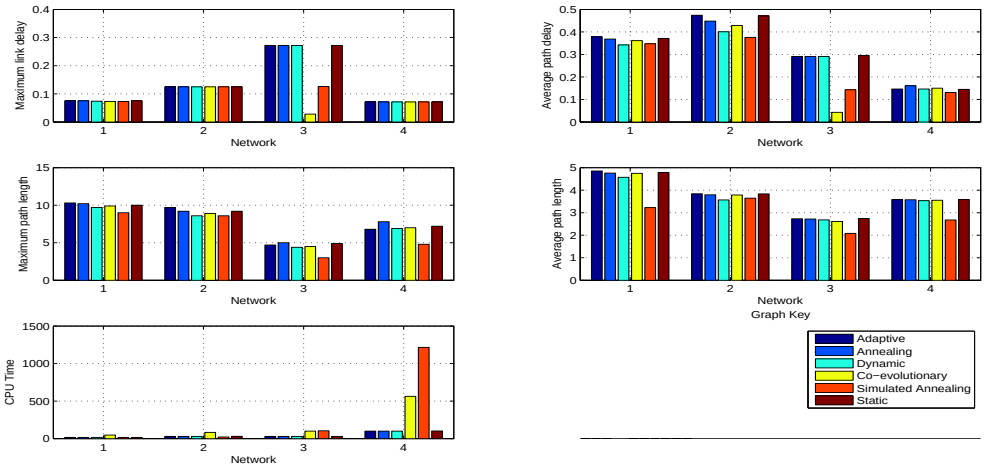

Figure 4.6: Weighted mean delay optimization. Plots from left to right are the adaptive penalty, annealing penalty, dynamic penalty, co-evolutionary penalty, simulated annealing and static penalty methods.

The first thing to note about these results is that the dynamic penalty, the co-evolutionary penalty and the simulated annealing methods perform better than the other penalty function methods in all criteria except the CPU-time. In most of the criteria except the CPU-time, the simulated annealing method performs better than the dynamic penalty and co-evolutionary penalty methods. However, there are the exceptions that the co-evolutionary penalty method has the least maximum link delay and the least average link delay for the Europe network. Also, the dynamic penalty method has the least average

path delay for the USA1 network and has the least average path length for the USA2 network.

Secondly, if the simulated annealing method is excluded from comparisons, then the dynamic penalty method over-performs the co-evolutionary penalty method in all criteria including CPU-time, except the exceptions that the co-evolutionary penalty method has the least maximum link delay and the least average link delay for the Europe network and has the least average path length for the Europe network.

Among the adaptive, annealing and static penalty function methods, we could not notice clear differences in the performances of these methods.

Thirdly, it is noticed that the co-evolutionary penalty method has the worst consumption of the CPU-time for USA1, USA2 and Europe networks. But, the simulated annealing time consumption is badly scaled in PAREN. It was 2.3 times slower than the co-evolutionary penalty, which, in turn, was 2.7 times slower than the next worst method.

4.2 The Mean Delay Metric

Tables A.3 and A.4 show the results obtained by applying the same methods (the dynamic, annealing, adaptive, co-evolutionary, and static penalty methods and the simulated annealing method) to the optimization of mean delay (as opposed to weighted mean delay). As before, the information is depicted graphically in Figure 4.7.

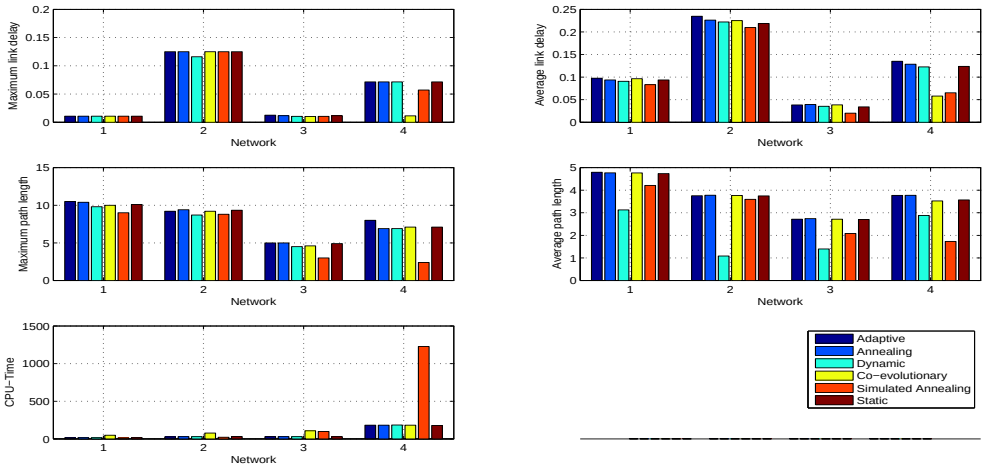

Figure 4.7: Mean delay optimization. Plots from left to right are the adaptive penalty, annealing penalty, dynamic penalty, co-evolutionary penalty, simulated annealing and static penalty methods.

Again we see that the simulated annealing, dynamic penalty and co-evolutionary penalty methods perform better than the adaptive, annealing and static penalty function methods. Both the simulated annealing and dynamic penalty methods perform better than the co-evolutionary penalty method, except in the cases of the maximum link delay and average link delay for PAREN. It is not clear which performance of the simulated annealing method or dynamic penalty method is better. While the performance of the simulated annealing in most of the first three criteria looks generally better than the performance of the dynamic penalty, the dynamic penalty method has the least average path lengths for the USA1, USA2 and Europe networks and has the least maximum link delay in the USA2 and Europe networks.

The same comment as before about CPU-time consumption is valid here. That is the co-evolutionary penalty function method has worst consumtion of the CPU-time for USA1, USA2 and Europe networks, whereas the simulated annealing consumes the most CPU-time for PAREN.

4.3 Stationary Penalty Function Methods

Recall that in Section 3.4 on page 34 we introduced four new stationary PFM by fixing the penalty factors of the dynamic, annealing, adaptive and co-evolutionary methods. In this section we present an evaluation of their performance.

4.3.1 Fixing the Time in the Dynamic PFM

We fix the time on the dynamic penalty method such that $C \cdot t = 1000$, with $\alpha = 1$. The results of minimization for both the weighted mean delay metric and the mean delay metric are shown in Table A.5 and in Figure 4.9.

From figures 4.8 and 4.9, it is clear that, the non-stationary dynamic penalty function method performs better than the stationary dynamic penalty method across the board.

4.3.2 Fixing the Temperature in the Annealing PFM

We fix the temperature on the annealing PFM, yielding a stationary penalty method. We used both low and high temperatures, specifically $T = 10$ and $T = 1000$. The results of minimization for both the weighted mean delay metric and the mean delay metric are shown in Table A.6, and in Figure 4.11.

From figures 4.10 and 4.11, it is obvious that the non-stationary annealing penalty method perform

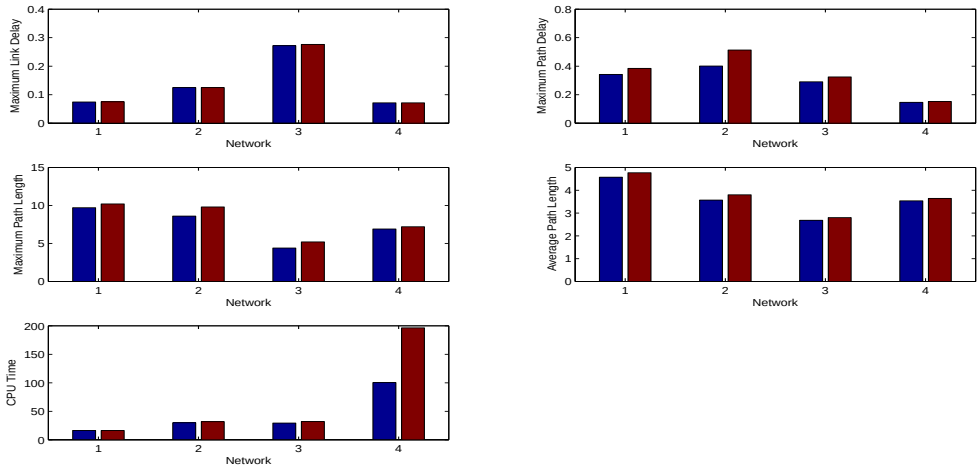

Figure 4.8: Stationary (right) and non-stationary (left) dynamic penalty method. Weighted mean delay optimization.

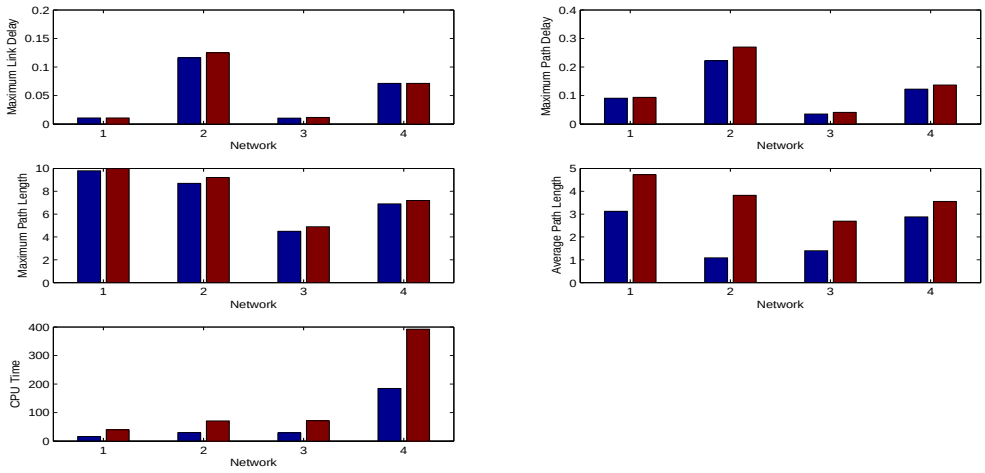

Figure 4.9: Stationary (right) and non-stationary (left) dynamic penalty method. Mean delay optimization.

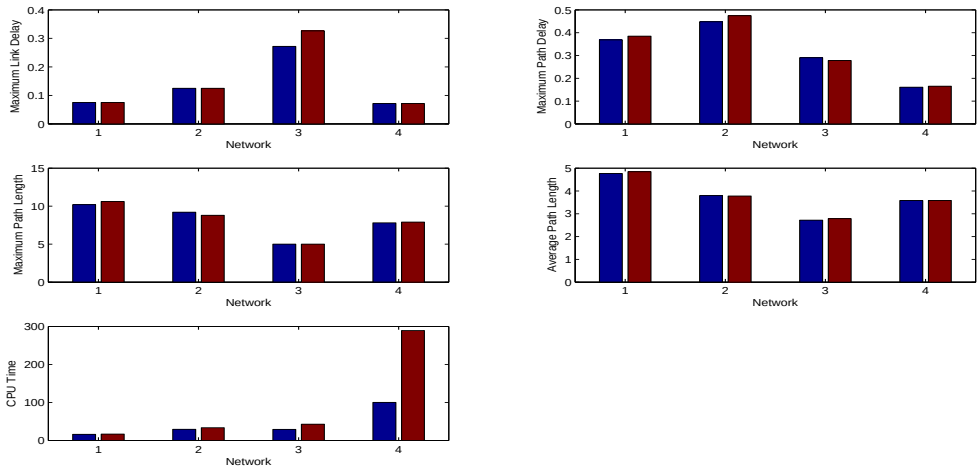

Figure 4.10: Stationary (right) and non-stationary (left) annealing penalty method. Weighted mean delay optimization

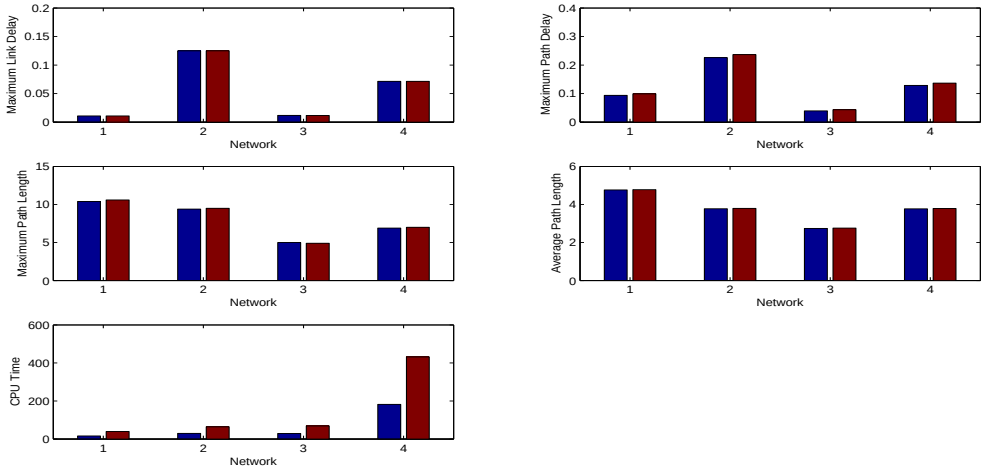

Figure 4.11: Stationary (right) and non-stationary (left) annealing penalty method. Mean delay optimization.

about the stationary PFM.

4.3.3 Setting *GenerationGap* = *NumberOfGenerations* in the Adaptive PFM

We set the variable *GenerationGap* to the value *NumberOfGenerations* in the adaptive PFM. We compared the stationary and non-stationary versions of the adaptive method; the results are shown in Table A.7 and in Figure 4.13.

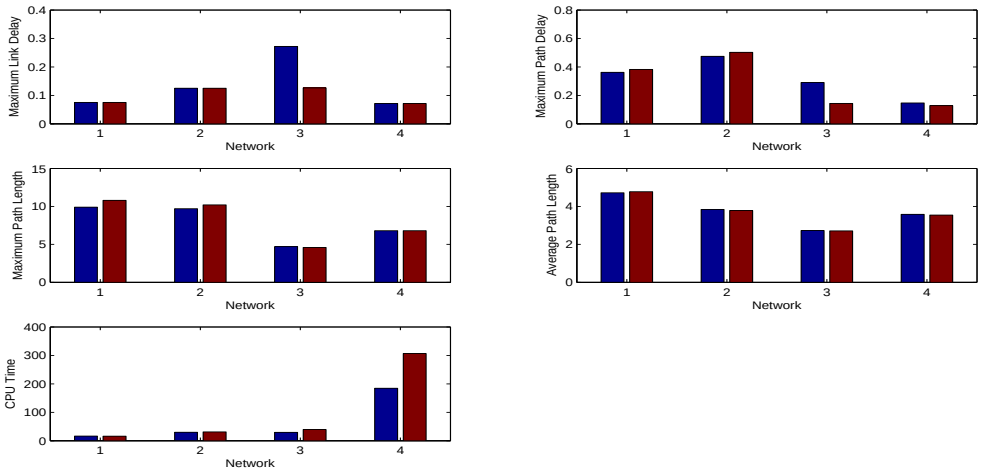

Figure 4.12: Stationary (right) and non-stationary (left) adaptive penalty method, *GenerationGap* = 5. Weighted mean delay optimization.

From Figure 4.13 we can see that the non-stationary adaptive penalty method performs about the stationary method. Hence, they are about the same.

4.3.4 Setting *GenerationMax2* = *PopulationSize2* = 1 for the Co-evolutionary PFM

We set *GenerationMax2* = *PopulationSize2* = 1 in the co-evolutionary penalty method. The results of minimization for both the weighted mean delay metric and the mean delay metric are shown in Table A.8 and in Figure 4.15.

From Figure 4.15, we see that the non-stationary co-evolutionary penalty method over-performs the stationary co-evolutionary penalty method in all criteria except the CPU time.

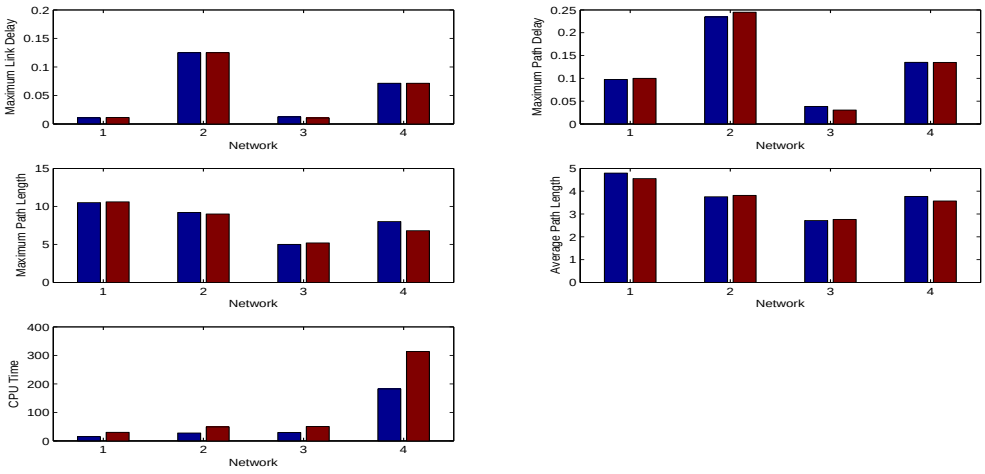

Figure 4.13: Stationary (right) and non-stationary (left) adaptive penalty method, $GenerationGap = 5$. Mean delay optimization.

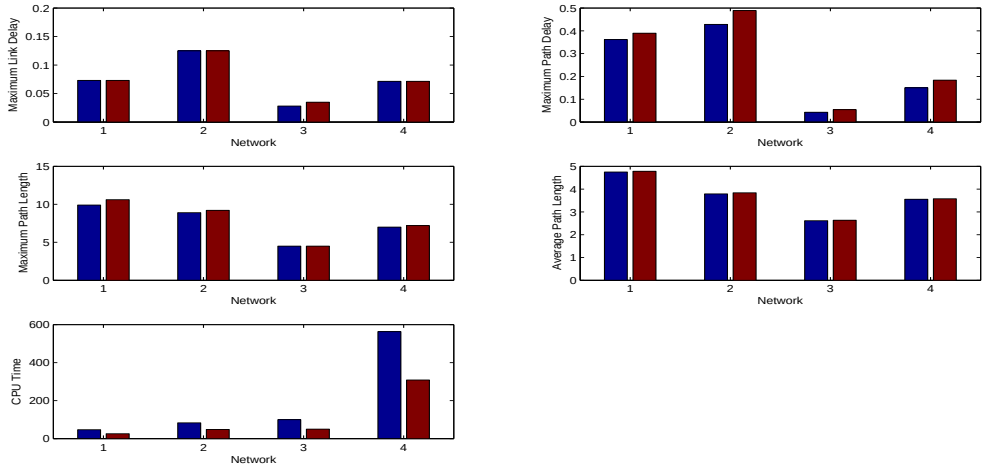

Figure 4.14: Stationary (right) and non-stationary (left) co-evolutionary penalty method, $GenerationMax2 = PopulationSize2 = 1$. Weighted mean delay optimization.

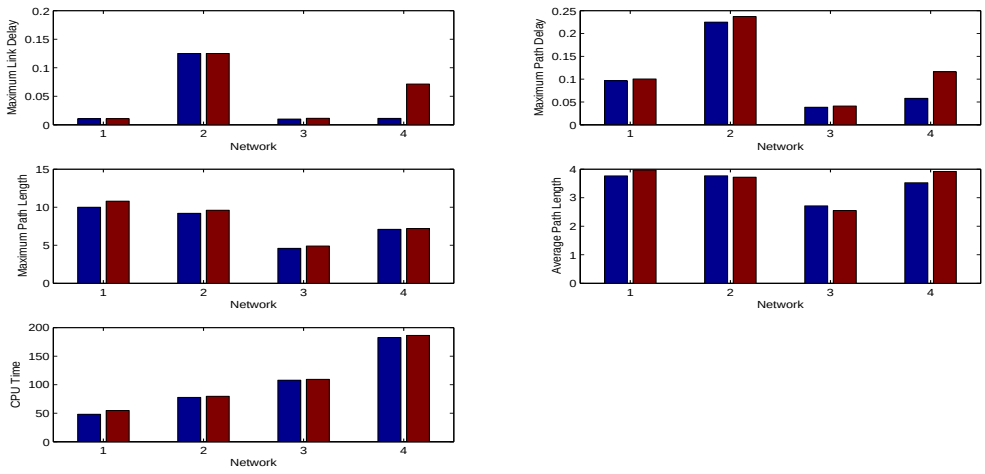

Figure 4.15: Stationary (right) and non-stationary (left) co-evolutionary penalty method, $GenerationMax2 = PopulationSize2 = 1$. Mean delay optimization.

4.4 Overview

- There is no clear winner among the non-stationary methods, but the simulated annealing, dynamic penalty and co-evolutionary penalty methods perform better than the adaptive, static and annealing PFMs.
- The simulated annealing provides better results in most of the criteria with less CPU-time for the USA1 and USA2 networks, but it consumes markedly more time for PAREN.
- The time required for the PAREN network is clearly much more than for any of the other three networks. This seems to be due to the fact that it has more than twice the number of traffic demands when compared to the next largest network.
- The stationary penalty methods are only faster for co-evolutionary, and only significantly better in a few exceptional cases.

Chapter 5

Conclusions

In this thesis our problem has been to compute the optimal paths on IP networks when the demands are routed from their sources to their destinations. We meant by optimal paths the paths which minimize the maximum path delay with respect to the weighted mean delay and the mean delay metrics. The two optimization problems are subject to constraints on the network links. Our main method for performing the optimization has been the genetic algorithms whereas our main methods for handling the constraints of the two problems have been the penalty function methods and the simulated annealing method. We considered both non-stationary and stationary penalty function methods. Our objective has been to compare the performances of the methods under consideration. We have done our computations on four networks with different structures.

We ran many simulations to obtain the optimal values for the parameters in the different penalty function methods. For the penalty factors λ, τ and C in the adaptive, annealing and dynamic penalty function methods we chose the values 10^j, $j = 1, \ldots, 7$ and fixed the other parameters to obtain the optimal values. For the parameter β in both the adaptive and dynamic penalty methods we chose among the values $0.5, 0.75, 1.0, \ldots, 2$ to obtain the optimal values. Then, we ran our main simulations based on those optimal parameter values.

Our computations have shown clearly that the simulated annealing, dynamic and co-evolutionary penalty function methods perform clearly better than the other penalty methods (adaptive, annealing and static). The performance of the simulated annealing method seems to be better than the other penalty function methods in most of the criteria, in both the weighted mean delay and mean delay optimizations. For small networks (networks with only a few numbers of links) the simulated annealing consumes less CPU time than the penalty function methods, but for networks with a large number of links, the simulated

annealing has much more complexity.

We have noticed that, fixing some penalty factors on dynamic and co-evolutionary penalty function methods does clearly worsen the results, but for the annealing and adaptive penalty function methods, it does not change much in the results we had except in few cases. For the co-evolutionary penalty method, fixing penalty factors reduces a lot of the consumed CPU-time.

Coello [12] tested the penalty function methods under consideration (except the simulated annealing method) and found that the co-evolutionary penalty function method performs better than the other penalty function methods. On the other hand, Michalewicz [38] ran simulations using the adaptive, annealing, dynamic, and static penalty function methods and found that the dynamic penalty methods perform better than the rest. The results obtained in this thesis -if the simulated annealing method is excluded from the comparisons- agree with the ones obtained by Coello and Michalewicz in arranging the performances of co-evolutionary and dynamic penalty function methods above the other penalty function methods. But they do not agree with Coello in that the performance of the co-evolutionary is better than the performance of the dynamic penalty.

Coello [12] stated other methods for handling the constraints of constrained-optimization problems that are associated to the genetic algorithms. They include hybrid methods such as the Lagrange multipliers, fuzzy logic, immune system simulations, cultural algorithms and the ant colony optimization. As far as we know, these hybrid methods have not been used for the QoS routing problems, except the ant colony optimization [3]. Therefore, more future research can be done to investigate which methods are better for handling the QoS routing problems constraints.

In addition to that, the performance of the simulated annealing method may be improved by doing one or all of the following changes in the simulated annealing algorithm.

1. Instead of using a random trial to obtain a random state which may lead to decrease the objective function, it is worthy to investigate the procedure of improving generation of the next trial points by using a particular direction which will lead to a decrease in the value of the objective function ($f(s_{k+1}) < f(s_k)$);
2. using a better cooling schedule to decrease the temperature; and
3. using other optimization methods such as particle swarm optimization.

Appendix A

Performance Evaluation Tables

This appendix contains the tables referred to in Chapter 4. They contain exactly the same information depicted in Figures 4.6–4.15.

Dynamic penalty method, $\alpha = 1$, $\beta = 2$, $C = 1000$

Network	MLD	MPD	MPL	APL	Time
USA1	0.07412	0.34226	9.70000	4.57193	16.40000
USA2	0.12500	0.40084	8.60000	3.56794	29.90000
Europe	0.27198	0.29046	4.40000	2.68333	29.40000
PAREN	0.07143	0.14643	6.90000	3.53570	100.30000

Annealing penalty method, $T_0 = 1000$

Network	MLD	MPD	MPL	APL	Time
USA1	0.07527	0.36852	10.20000	4.76112	16.20000
USA2	0.12500	0.44803	9.20000	3.79557	29.50000
Europe	0.27198	0.29042	5.00000	2.71875	29.30000
PAREN	0.07143	0.16084	7.80000	3.57740	100.30000

Adaptive penalty method, $\lambda_0 = 100$, $GenerationGap = 5$

Network	MLD	MPD	MPL	APL	Time
USA1	0.07527	0.37818	10.30000	4.85462	16.30000
USA2	0.12500	0.47431	9.70000	3.84343	30.00000
Europe	0.27198	0.29072	4.70000	2.72917	30.10000
PAREN	0.07143	0.14643	6.80000	3.58260	100.50000

Table A.1: Weighted mean delay optimization

Co-evolutionary penalty method, *PopulationSize1* = 100,
PopulationSize2 = 10, *GenerationMax1* = 70, *GenerationMax2* = 10

Network	MLD	MPD	MPL	APL	Time
USA1	0.07298	0.36145	9.90000	4.74844	47.20000
USA2	0.12500	0.42830	8.90000	3.78747	83.00000
Europe	0.02804	0.04293	4.50000	2.61111	100.00000
PAREN	0.07143	0.15025	7.00000	3.55201	563.00000

Static penalty method

Network	MLD	MPD	MPL	APL	Time
USA1	0.07527	0.37132	10.00000	4.78268	16.40000
USA2	0.12500	0.47105	9.20000	3.83518	30.20000
Europe	0.27198	0.29574	4.90000	2.74583	30.10000
PAREN	0.07143	0.14514	7.20000	3.59250	102.40000

Simulated annealing method, $T_0 = 1000$, $\Delta T = 0.08$

Network	MLD	MPD	MPL	APL	Time
USA1	0.07298	0.34766	9.00000	3.22633	14.40000
USA2	0.12500	0.37596	8.60000	3.64578	22.00000
Europe	0.12621	0.14316	3.00000	2.08333	104.60000
PAREN	0.07143	0.13114	4.80000	2.67966	1216.0000

Table A.2: Weighted mean delay optimization (continued)

Dynamic penalty method, $\alpha = 1$, $\beta = 2$, $C = 1000$

Network	MLD	MPD	MPL	APL	Time
USA1	0.01073	0.09061	9.80000	3.12547	16.00000
USA2	0.11608	0.22204	8.70000	1.08658	29.70000
Europe	0.01048	0.03536	4.50000	1.39894	29.30000
PAREN	0.07143	0.12261	6.90000	2.88190	184.60000

Annealing penalty method, $T_0 = 1000$

Network	MLD	MPD	MPL	APL	Time
USA1	0.01074	0.09364	10.40000	4.76707	16.30000
USA2	0.12500	0.22606	9.40000	3.77732	29.70000
Europe	0.01168	0.03920	5.00000	2.73958	29.20000
PAREN	0.07143	0.12858	6.90000	3.77150	182.00000

Adaptive penalty method, $\lambda_0 = 100$, $GenerationGap = 5$

Network	MLD	MPD	MPL	APL	Time
USA1	0.01074	0.09755	10.50000	4.79408	15.10000
USA2	0.12500	0.23465	9.20000	3.75619	27.50000
Europe	0.01250	0.03839	5.00000	2.71458	29.40000
PAREN	0.07143	0.13508	8.00000	3.76920	182.20000

Table A.3: Mean delay optimization

Co-evolutionary penalty method, *PopulationSize1* = 100, *PopulationSize2* = 10, *GenerationMax1* = 70, *GenerationMax2* = 10

Network	MLD	MPD	MPL	APL	Time
USA1	0.01075	0.09646	10.00000	4.76534	48.20000
USA2	0.12500	0.22503	9.20000	3.76618	77.77778
Europe	0.01020	0.03858	4.60000	2.71429	108.00000
PAREN	0.01139	0.05804	7.10000	3.52400	182.60000

Static penalty method

Network	MLD	MPD	MPL	APL	Time
USA1	0.01074	0.09371	10.10000	4.73363	16.20000
USA2	0.12500	0.21872	9.33333	3.74891	30.10000
Europe	0.01161	0.03401	4.90000	2.70417	29.80000
PAREN	0.07143	0.12374	7.10000	3.56920	177.50000

Simulated annealing method, $T_0 = 1000$, $\Delta T = 0.08$

Network	MLD	MPD	MPL	APL	Time
USA1	0.01073	0.08319	9.00000	4.21076	13.20000
USA2	0.12500	0.20958	8.80000	3.59536	21.80000
Europe	0.01043	0.02013	3.00000	2.08333	97.80000
PAREN	0.05714	0.06533	2.40000	1.73204	1226.20000

Table A.4: Mean delay optimization (continued)

Weighted mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.07527	0.38433	10.20000	4.76507	16.20
USA2	0.12500	0.51344	9.80000	3.79903	31.80
Europe	0.27621	0.32447	5.20000	2.80000	32.00
PAREN	0.07143	0.15197	7.20000	3.64319	196.40

Mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.0108	0.0939	10.0000	4.7276	39.7000
USA2	0.1250	0.2702	9.2000	3.8240	70.8000
Europe	0.0117	0.0413	4.9000	2.6917	71.4000
PAREN	0.0714	0.1370	7.2000	3.5574	392.4000

Table A.5: Stationary dynamic penalty methods, $C \cdot t = 1000$

Weighted mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.07527	0.38465	10.60000	4.84418	16.80
USA2	0.12500	0.47452	8.80000	3.77827	33.60
Europe	0.32621	0.27788	5.00000	2.78333	42.80
PAREN	0.07143	0.16530	7.90000	3.58366	289.20

Mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.0108	0.0990	10.6000	4.7791	39.6000
USA2	0.1250	0.2367	9.5000	3.7957	64.0000
Europe	0.0116	0.0433	4.9000	2.7575	68.8000
PAREN	0.0714	0.1367	7.0000	3.7909	431.7000

Table A.6: Stationary annealing penalty method, $T = 100$

Weighted mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.07527	0.39564	10.40000	4.78377	16.600
USA2	0.12500	0.49852	9.40000	3.83177	31.40
Europe	0.29621	0.25074	4.90000	2.73333	39.20
PAREN	0.07143	0.15983	7.00000	3.57502	306.20

Mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.01121	0.09992	10.60000	4.55403	30.00
USA2	0.12500	0.24507	9.00000	3.81605	49.20
Europe	0.01047	0.03047	5.20000	2.76250	50.20
PAREN	0.07143	0.13495	6.80000	3.56901	313.40

Table A.7: Stationary adaptive penalty method, $\lambda_0 = 100$, $GenerationGap = 5$

Weighted mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.07298	0.38937	10.60000	4.78101	26.00
USA2	0.12500	0.48893	9.20000	3.83591	48.40
Europe	0.03485	0.05423	4.50000	2.63500	50.00
PAREN	0.07143	0.18373	7.20000	3.57594	308.80

Mean delay optimization

Network	MLD	MPD	MPL	APL	Time
USA1	0.01075	0.10047	10.80000	3.96793	54.80
USA2	0.12500	0.23722	9.60000	3.72068	79.80
Europe	0.01141	0.04115	4.90000	2.54894	109.40
PAREN	0.07143	0.11610	7.20000	3.92127	186.40

Table A.8: Stationary co-evolutionary penalty methods, $GenerationMax2 = PopulationSize2 = 1$

Bibliography

[1] A.E. Krzesinski A. Arvidsson, B.A. Chiera and P.G. Taylor. A distributed scheme for value-based bandwidth re-configuration. In *Proceedings International Workshop on Traffic Management and Traffic Engineering for the Future Internet (FITraMEn 08)*, Porto, Portugal, 2008.

[2] A. F. R. Araújo, C. Garrozi, A. R. G. A. Leitao, and M. M. Gouvea Jr. Multicast routing using genetic algorithm seen as a permutation problem. In *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 477–484, 2006.

[3] A. B. Bagula, H. A. C. de Villiers, J. du Toit, A. E. Krzesinski, M. Loubser, and J. G. van der Horst. Ant routing simulation. In *Proceedings of the Southern Africa Telecommunication Networks and Applications Conference (SATNAC'03)*, pages 317–322, 2003.

[4] S. Balon, F. Skivée, and G.Leduc. How well do traffic engineering objective functions meet TE requirements. In *Lecture Notes in Computer Science*, volume 3976, pages 75–86. Springer Berlin/Heidelberg, 2006.

[5] S. Balon, F. Skivée, and G. Leduc. Comparing traffic engineering objective functions. In *CoNEXT '05: Proceedings of the 2005 ACM conference on Emerging network experiment and technology*, pages 224–225, New York, NY, USA, 2005. ACM.

[6] J. C. Bean and A. B. Hadj-Alouane. A dual genetic algorithm for bounded integer programs. Technical Report TR 92-53, Department of Industrial and Operations Engineering, The University of Michigan, 1992.

[7] J. T. Betts. *Practical Methods for Optimal Control Using Nonlinear Programming.* Society for Industrial and Applied Mathematics, 2001.

[8] J. Cardy. *Scaling and Renormalization in Statistical Physics.* CUP, 1996.

[9] S. E. Carlson and R. Shonkwiler. Annealing a genetic algorithm over constraints. In *1998 IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, pages 3931–3936.

[10] Ping Chen and Tian lin Dong. A fuzzy genetic algorithm for QoS multicast routing. *Computer Communications*, 26(6):506–512, 2003.

[11] C. A. Coello. Use of a self-adaptive penalty approach for engineering optimization problems. *Computers in Industry*, 41(2):113–127, 2000.

[12] C. A. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: A survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12):1245–1287, 2002.

[13] Y. Cui, K. Xu, M. Xu, and J. Wu. *Optimal QoS Routing Based on Extended Simulated Annealing*, volume 2662 of *Lecture Notes in Computer Science*, pages 553–562. Springer Berlin/Heidelberg, 2003.

[14] A. Elwalid, C. Jin, S. H. Low, and I. Widjaja. Mate: MPLS adaptive traffic engineering. In *IEEE The Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies, Infocom 2001*, volume 3, pages 1300–1309, 2001.

[15] T. M. Fathelrahman and A. B. Bagula. On routing IP traffic using multi-constrained genetic optimization with penalty functions. Southern African Telecommunication Networks and Applications Conference (SATNAC), 2006.

[16] B. Fortz, J. Rexford, and M. Thorup. Traffic engineering with traditional IP routing protocols. *IEEE Communications Magazine*, 40(10):118–124, 2002.

[17] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Reading, 1989.

[18] A. B. Hadj-Alouane and J. C. Bean. A genetic algorithm for the multiple choice integer program. *Operations Research*, 45:92–101, 1997.

[19] A. T. Haghighat, K. Faeza, M. Dehghanb, A. Mowlaeia, and Y. Ghahremani. GA-based heuristic algorithms for bandwidth-delay-constrained least-cost multicast routing. *Computer Communications*, 27:111127, 2004.

[20] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[21] F. Hoffmeister and J. Sparve. Problem-independent handling of constraints by use of metric penalty functions. In L. J. Fogel P. J. Angeline and T. Back, editors, *Proceedings of the Fifth Annual Conference on Evolutionary Programming (EP'96)*, pages 289–294, San Diego, California, 1996. The MIT Press.

[22] A. Homaifar, S. H. Y. Lai, and X. Qi. Constrained optimization via genetic algorithms. *Simulation*, 62(4):242–254, 1994.

[23] Ren-Hung Hwang, Wei-Yuan Do, and Shyi-Chang Yang. Multicast routing based on genetic algorithms. *J. Inf. Sci. Eng.*, 16(6):885–901, 2000.

[24] J. Joines and C. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GAs. In David Fogel, editor, *Proceeding of the First IEEE Conference on Evolutionary Computation*, pages 579–584. IEEE Press, 1994.

[25] I. Juva. Analysis of quality of service routing approaches and algorithms. Master's thesis, Department of Engineering Physics and Mathematics, Helsinki University of Technology, 2003.

[26] P. Kampstra, R. D. van der Mei, and A. E. Eiben. Evolutionary computing in telecommunication network design: a survey. In revision, 2006.

[27] S. Kazarlis and V. Petridis. *Varying Fitness Functions in Genetic Algorithms: Studying the Rate of Increase of Dynamic Penalty Terms.* Parallel Problem Solving from Nature. Springer-Verlag, V-PPSN V, Amsterdam, The Netherlands, 1998.

[28] S. Kirkpatrick, J. C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[29] T. Korkmaz and M. Krunz. Multi-constrained optimal path selection. In *Proc. 20th Joint Conf. IEEE Computer and Communications Societies (INFOCOM 2001): Twenty years into the communications odyssey*, pages 834–843. 2001.

[30] F. A. Kuipers, T. Korkmaz, M. Krunz, and P. van Mieghem. Performance evaluation of constraint-based path selection algorithms. *IEEE Network*, 18(5):16–23, 2004.

[31] F. A. Kuipers, T. Kormaz M. Krunz, and P. van Mieghem. An overview of constraint-based path selection algorithms for QoS routing. *IEEE Communications Magazine*, 40(12):50–55, 2002.

[32] A. F. Kuri-Morales and C. V. Quezada. A universal eclectic genetic algorithm for constrained optimization. In *In Proceedings 6th European Congress on Intelligent Techniques and soft Computing, EUFIT'98*, pages 518–522, Aachen, Germany, 1998.

[33] Y. Li, J. Harms, and R. Holte. Fast exact multiconstraint shortest path algorithms. In *IEEE International Conference on Communications, 2007. ICC apos;07*, volume 24, pages 123–130. 2007.

[34] L. Liu and G. Feng. *A Novel Heuristic Routing Algorithm Using Simulated Annealing in Ad Hoc Networks*, volume 3746 of *Lecture Notes in Computer Science*, pages 849–857. Springer Berlin/Heidelberg, 2005.

[35] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equations of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[36] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* AI Series. Springer-Verlag, 1992.

[37] Z. Michalewicz. Genetic algorithms, numerical optimization, and constraints. In L. J. Eshelman, editor, *Proceedings of the sixth International Conference on Genetic Algorithms*, pages 151–158, University of Pittsburgh, San Mateo, California, 1995. Morgan Kaufmann Publishers.

[38] Z. Michalewicz. A survey of constraint handling techniques in computation methods. In *Proceedings of the 4^{th} Annual Conference on Evolutionary Programming*, pages 135–155. The MIT Press, 1995.

[39] Z. Michalewicz and N. F. Attia. Evolutionary optimization of constrained problem. In *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, pages 98–109, 1994.

[40] H. Moustafa and H. Labiod. Multicast routing in mobile ad hoc networks. *Telecommunication Systems*, 25(1-2):65–88, 2004.

[41] C. A. S. Oliveira and P. M. Pardalos. A survey of combinatorial optimization problems in multicast routing. *Computers & OR*, 32(8):1953–1981, 2005.

[42] L. S. Randaccio and L. Atzori. Group multicast routing problem: A genetic algorithms based approach. *Computer Networks*, 51(14):3989–4004, 2007.

[43] A. Roy and S. K. Das. Qm^2rp: A QoS-based mobile multicast routing protocol using multi-objective genetic algorithm. *Wireless Networks*, 10(3):271–286, 2004.

[44] G. Rudolph. Convergence properties of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, 1994.

[45] M. C. Sinclair. Evolutionary algorithms for optical network design: A genetic-algorithm/heuristic hybrid approach, 2001. University of Essex.

[46] J. H. Siregar, Yongbing Zhang, and H. Takagi. Optimal multicast routing using genetic algorithm for WDM optical networks. *IEICE Transactions*, 88-B(1):219–226, 2005.

[47] Zheng Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE journal of Selected Areas in Communications*, 14(7):1228–1234, 1996.

[48] F. Wu. A framework for memetic algorithms. Master's thesis, 2001.

[49] Fei Xiang, Junzhou Luo, Jieyi Wu, and Guanqun Gu. QoS routing based on genetic algorithm. *Computer Communications*, 22(15-16):1392–1399, 1999.

[50] Cui Xunxue, Li Qin, and Tao Qing. Genetic algorithm for Pareto optimum-based route selection. *Journal of Systems Engineering and Electronics*, 18(2):360–368, 2007.

[51] M. H. Yaghmae. Quality of service routing in MPLS networks using delay and bandwidth constraints. Available online at the citeseer website, 1999.

[52] O. Younis and S. Fahmy. Constraint-based routing in the internet: Basic principles and recent research. In *IEEE Communications Surveys & Tutorials 5*, 2003. Available online http://www.comsoc.org/livepubs/surveys/public/2003/sep/pdf/fahmy.pdf.

[53] M. S. Zahrani, M. J. Loomes, J. A. Malcolm, and A. A. Albrecht. Genetic local search for multicast routing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 615–616, 2006.

[54] Qingfu Zhang and Yiu-Wing Leung. An orthogonal genetic algorithm for multimedia multicast routing. *IEEE Trans. Evolutionary Computation*, 3(1):53–62, 1999.