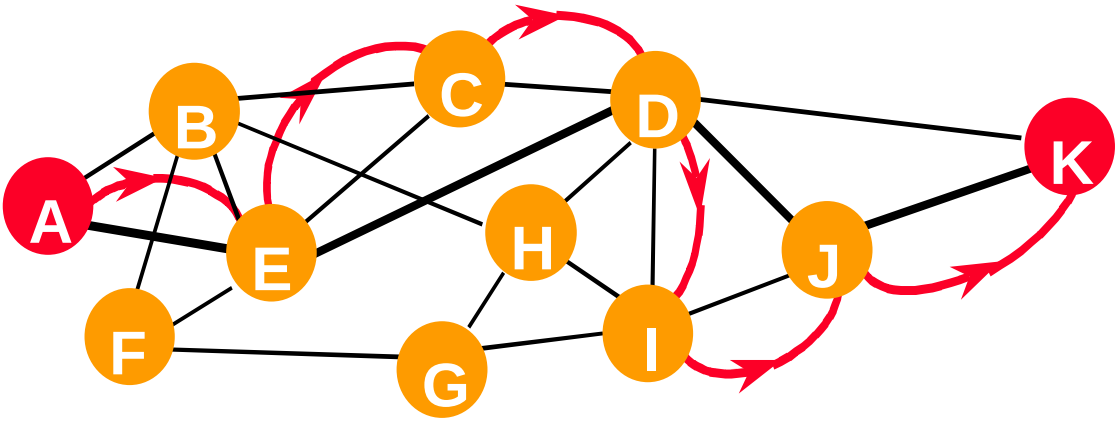
Hop-by-hop Destination Based Routing with Quality of Service Constraints

An evaluation of SAMCRA in a connectionless and connection-oriented environment



Fernando A. Kuipers (9486794)

Supervisor: Prof. dr. ir. P. Van Mieghem

Delft University of Technology

Information Technology and Systems

Telecommunications and Traffic-Control Systems (TVS) group

Assignment number: A-878

June, 2000

Preface

This Master's thesis has been conducted at the Telecommunications and Traffic-Control Systems (TVS) group of the Department of Electrical Engineering, Faculty of Information Technology and Systems, Delft University of Technology. It targets the problems surrounding quality of service (QoS) routing, where it mainly focuses on the algorithmic aspect of path selection based on multiple constraints.

This thesis may be of interest to anyone having an affiliation with algorithms, because although the setting of this thesis lies in the telecommunications area, the proposed algorithm(s) may be of use in any other field.

The readers whom are mainly interested in an overview of QoS-algorithms are directed to paragraph 2.1. The NP-completeness surrounding QoS routing is discussed in appendix A.

Acknowledgements

I would like to thank Prof. dr. ir. Piet Van Mieghem (Delft University of Technology) and dr. ir. Hans De Neve (Alcatel CRC Antwerp) for their insightful comments and suggestions during this work. They are the founders of the algorithms TAMCRA and SAMCRA and provided the necessary means for efficiently conducting my research.

Abstract

This thesis provides an overview of the most encountered quality of service (QoS) routing algorithms. In particular the algorithm SAMCRA is discussed in dept since its running-time in practice is polynomial and it guarantees exactness. This exactness is essential when routing is performed in a hop-by-hop manner, because this guarantees that no paths with loops are made. In the second part, routing deployed in a hop-by-hop manner is investigated, because this forms the basis of IP networking. At each hop along the path SAMCRA computes the next hop towards the destination, ignoring the previous path history (as in current IP routing). The results are evaluated and compared to the exact results. Although the results are considered good, the guaranteed exactness is lost. However since SAMCRA is exact, this means that there simply does not exist an algorithm that is able to guarantee QoS in an IP network. To guarantee QoS in a hop-by-hop manner, we need an active network. The performance of SAMCRA in these networks is discussed in the third part of this thesis. Two routing approaches are discussed here, the first does not always find the (exact) shortest path but does guarantee the requested QoS and the second one is exact but more complex.

Keywords: routing, quality of service (QoS), IP, multiple constraints routing, TAMCRA, SAMCRA, NP-completeness, active networks.

Contents

1. Introduction: Guaranteed Quality of Service

The past decade has shown a large-scale evolution in the communications sector. The two main contributions to this evolution came from the Internet, a connectionless data-network that was used for non-real-time communication, and the increasing interest in telecommunications, real-time communication based on a connection-oriented network. Today we still see this evolution progressing, however the big distinction between the data-world and the telecom-world is disappearing. The data-world is showing an increasing interest in supporting real-time communication, whereas the telecom-world is investing in an infrastructure that can support data-communication. To predict the future is always dangerous, but most likely it will consist of the integration of the data-world and the telecom-world, so that we will end up in a world where (broadband) information is ubiquitous, delivered timely, at relatively low costs with high quality. These features can not be delivered without the incorporation of guaranteed quality of service. Guaranteed quality of service (QoS) will accomplish that the user can request his QoS-constraints (e.g. bandwidth should be at least 2 Mb/s or the delay may not exceed 100 ms). For instance, if the user wants to see a movie on-line (streaming video) he needs, and therefore requests a connection with a large bandwidth and a small delay/jitter.

Routing deployed in today's Internet is focussed on connectivity and typically supports only one type of datagram service called "best effort", which basically treats every user in the same way. Current Internet routing protocols, e.g. OSPF [Moy, 1998] and RIP [Malkin, 1994], use a shortest path algorithm, e.g. Dijkstra's algorithm [Dijkstra, 1959] to retrieve the best route. This algorithm will find the shortest path based on a single arbitrary metric, e.g. administrative weight or hop-count. As a consequence this algorithm can not be used for QoS-routing. QoS-routing needs algorithms that compute paths that satisfy a set of end-to-end QoS-constraints. There are a number of complex and interrelated issues involved in the design of QoS routing algorithms and their use in a network. These issues were first addressed in the telecom world, where routing is performed in a connection-oriented manner. ATM with its PNNI protocol probably illustrates this best. To accommodate the multiple constraints routing problem, H. De Neve and P. Van Mieghem developed a routing algorithm to be integrated in the PNNI protocol stack [De Neve ea., 1998]. This algorithm which

later on evolved into TAMCRA [De Neve ea., 2000] is a promising algorithm for the use in a guaranteed QoS routing protocol.

Besides the telecom world also the Internet community (data-world) has realised the importance of being able to provide a certain quality of service. The IETF has proposed a signalling protocol (RSVP [Braden ea., 1997]) and traffic classes (guaranteed service and controlled load in the IETF's Integrated Services), but apart from a framework document on QoS Routing [Crawley ea., 1998] and the specification for an experimental protocol [Apostolopoulos ea., 1999], no QoS routing protocol has been standardised yet.

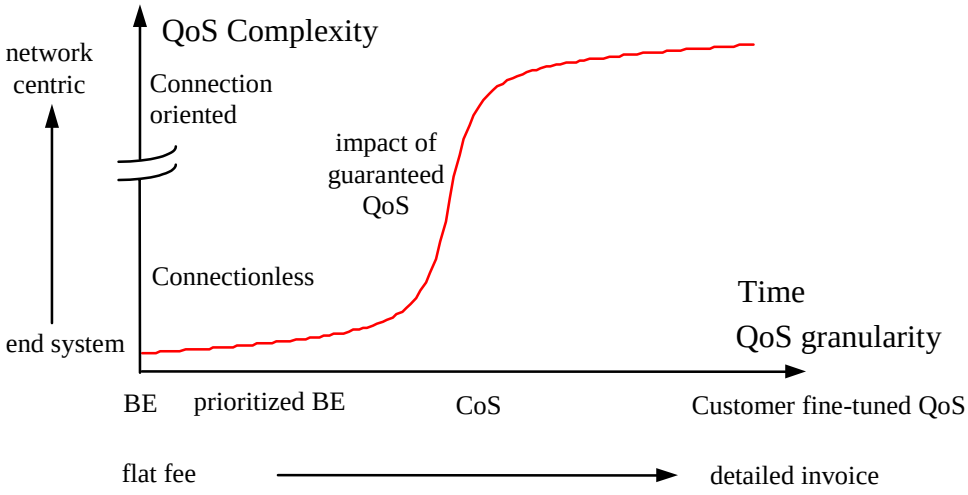


Figure 1. The complexity of QoS provisioning as a function of time or the granularity of QoS differentiation.

After the work in the IETF Integrated Services and RSVP group, it became clear that the complexity of guaranteeing QoS (as illustrated in figure 1) was an overwhelming problem especially because offering guarantees to the huge number of flows currently present in the Internet backbones seemed hardly feasible [Van Mieghem ea., 1999]. In order to cope with this scaling and aggregation problem, it was suggested to relax the notion of QoS guarantee. This idea was supported by the argument that the majority of flows can live without QoS guarantees, or at least that their generating applications may adapt to variable network conditions. These considerations have provided great impetus to the Differentiated Services IETF group, that is concerned with methods for providing differentiated classes of service for Internet traffic, to support various types of applications and specific business requirements. A "service" defines some significant characteristics of packet transmission in one direction across a set of one or more paths within a network.

These characteristics may be specified in quantitative or statistical terms of throughput, delay, jitter, and/or loss, or may otherwise be specified in terms of some relative priority of access to network resources [Blake ea., 1998].

The main goal of this thesis is to evaluate the feasibility of “hop-by-hop destination based only” guaranteed QoS routing. Although QoS routing naturally asks for on-demand, end-to-end and explicit routing, the focus will be on the algorithmic problem of multiple constraints routing in a connectionless environment. Therefore, throughout this thesis, guaranteed QoS refers to the guarantee that a path within the QoS-constraints is found *if such a path exists*!

The thesis is divided into three parts. Chapter 2 will describe a multiple constraints routing algorithm, TAMCRA, and its exact, self-adaptive version SAMCRA. In chapter 3, SAMCRA is evaluated in a “hop-by-hop destination based only” context. The version of hop-by-hop SAMCRA applied in this chapter uses the QoS routing constraints in a static manner. This opens the possibility for hop-by-hop SAMCRA to be used in an extended OSPF protocol. Chapter 4 evaluates the dynamic/active hop-by-hop SAMCRA algorithm. This algorithm dynamically adjusts the QoS constraints, but can not be used in today’s routing protocols and requires an active networking protocol.

2. The TAMCRA and SAMCRA algorithms

This chapter will focus on the algorithmic aspect of QoS routing. This amounts to finding an algorithm that can find a path in a given network, based on the current state of the network and the QoS constraints requested. A network can be modelled as a graph $G(V,E)$, consisting of $|V|$ vertices (representing nodes, e.g. routers and switches) connected by $|E|$ edges (representing the communication links between the nodes). Every edge $e \in E$ connecting the nodes u and $v \in V$, specified by a link vector $l(u,v)$, has m associated additive link metrics $l_i(u,v) > 0$ with $i = 1,...,m$. Traditional shortest path algorithms assume $m = 1$, but for the QoS routing problem, $m > 1$. A path P from source s to destination d through the network is a sequence of vertices $s=v_0,v_1,...,v_n=d$ such that $(v_{i-1},v_i) \in E$ for $i = 1,...,n$. The link vector $l(P)$ represents the m costs of the path and totally depends on the link vectors of the individual links on the path. The multi-constrained path (MCP) problem can be stated as follows: Given the m maximum allowable constraints $L_i > 0$, $i = 1,...,m$, find a path P from s to d obeying all the constraints: $l_i(P) \leq L_i$, $i = 1,...,m$.

If we look at the type of constraints handled by multiple constraints algorithms, we can make the following distinction:

1. Algorithms that use min/max constraints. Whether the requested constraints can be supported depends on one single link on the path that has the lowest/highest metric (the weakest link in the chain). If for instance there is a request for a minimum bandwidth of 2 Mb/s, we can simply filter out every link that does not have sufficient capacity to serve the requested 2 Mb/s. Using a simple algorithm, we can now calculate the shortest path through the remaining topology ([Wang ea., 1996] presents an algorithm based on this principle). As shown, this kind of multiple constraints routing is not considered complex and will therefore not be further evaluated in this thesis.
2. Algorithms that use additive constraints. The problem of finding a path that is subject to more than one additive constraint is known to be NP-complete (see appendix A) and therefore intractable for large networks. Most algorithms therefore try to transform the multiple constraints into a single constraint with the use of for instance a linear equation, see [Jaffe, 1984]. The most promising

algorithm seems to be TAMCRA or its successor SAMCRA. These algorithms will be further explained below and briefly compared with Jaffe's algorithm.

The amount of constraints requested (*m*) is usually larger than one, so we will have to abort the currently most used shortest path algorithms that can only handle one metric (like Dijkstra's algorithm). We therefore need an algorithm that can handle multiple constraints. A fair amount of algorithms are capable of handling more than one constraint. However the majority of these algorithms is not generic in the sense that they can not be used for arbitrary QoS-parameters. They try to solve the MCP problem for specific parameters, e.g. delay constrained – least cost routing (DCLC: find a path obeying a delay-constraint, which has a minimal cost. This DCLC problem is an optimisation problem, whereas the MCP problem would look for a delay- and cost-constrained path). An overview of the most encountered multiple constraints algorithms is given, partly to indicate that there are only few algorithms that can solve the MCP problem for any number of arbitrary constraints.

2.1 An overview of QoS-algorithms

The overview is divided into two sections, the first section presents the non-generic algorithms. All the algorithms in this class focus on specific QoS-parameters and are not able to solve the true MCP problem. They are restricted to a fixed number of constraints. The second section presents the algorithms designed to solve the true MCP problem. These generic algorithms are able to compute a path from source to destination based on an arbitrary number of constraints.

2.1.1 Non-generic algorithms:

- The Constrained Bellman-Ford (CBF) algorithm proposed by Widyono [Widyono, 1994] solves the delay-constrained least-cost problem (some other algorithms to solve the DCLC problem are also discussed in this paper). The CBF algorithm finds the minimum cost path that does not violate the delay constraints. The algorithm performs a breadth-first-like search (see [Cormen ea., 1997] for an explanation on breadth-first algorithms), discovering paths of monotonically increasing delay while recording and updating lowest cost paths to each node it visits. The algorithm is exact, but the price for exactness is paid in running-time, which may grow exponentially.

- In [Salama ea., 1997] the authors propose to use a distributed heuristic to solve the DCLC problem. This heuristic, called the delay-constrained unicast routing (DCUR) algorithm, requires a cost vector and a delay vector to be kept at each node. The cost vectors and delay vectors are similar to the distance vectors of existing distance-vector-based protocols such as RIP. In addition to the cost vector and delay vector, each node maintains a routing table and an invalid-link table. DCUR is a source-based algorithm that constructs a delay-constrained path connecting source *s* to destination *d*. The path is constructed one node at a time, from *s* to *d*. Any node *v* at the head of the partially-constructed path can choose to add one of only two alternative outgoing links. One link belongs to the least-cost path from *v* to *d* and the other to the least-delay path. Each node *v* maintains a routing table of all paths (either partially or fully constructed) which pass through it. Routing table entries are created while establishing a connection for a session and deleted when the session terminates.

  The DCUR algorithm may return costly paths when using tight delay constraints, because in that case the algorithm will hardly follow the unconstrained least-cost path.

  Another distributed algorithm is presented in [Sun ea., 1997].
- Ma and Steenkiste [Ma ea., 1997] show that if all the QoS-parameters are related to each other, the MCP-problem can be solved in polynomial time. They illustrate this for networks that use WFQ-like scheduling algorithms ([Bennet ea., 1996], [Demers ea., 1989], [Golestani, 1994], [Zhang, 1990]) to support end-to-end delays. These WFQ-like scheduling algorithms isolate each guaranteed session from other sessions to ensure a guaranteed share of link resources and hence create a relationship between the bandwidth, delay, delay-jitter and/or buffer-space. This relationship is enough to solve the MCP-problem (for these parameters) in polynomial time (see appendix A).
- Wang and Crowcroft [Wang ea., 1996] propose an algorithm to solve the bandwidth-delay-constrained path problem by filtering out the links that can not accommodate the requested bandwidth (the links are not actually removed, but their delay is set to ∞). After this a shortest path algorithm (Dijkstra) is used to find the path with minimum delay. If this minimum delay is found, it is determined whether it obeys the delay-constraint. If this is not the case, no

suitable path exists.

This is a simple and exact way to solve a simple problem. The problem becomes difficult when there are more than one additive constraints (e.g. delay *and* jitter).

- In [Guérin ea., 2000] Guérin and Orda focus on the impact of advance reservations on the path selection process. They describe possible extensions to path selection algorithms in order to make them advance-reservation aware, and evaluate the added complexity introduced by these extensions. In their basic advance reservation model, requests are for a given amount of bandwidth between times t_1 and $t_2 \geq t_1$. Path selection then attempts to find a path with sufficient bandwidth in that interval (by deleting all links that can not accommodate the requested bandwidth). They also extend this basic model to incorporate delay guarantees in a rule-based environment [Shenker ea., 1997]. Given an advance reservation request i, the problem now is to choose a path P between s^i and d^i and a rate r^i, such that $r^i \geq$ "requested bandwidth" is guaranteed in the specified interval and the delay of this path also meets the delay-constraint. The proposed algorithm to this problem resembles that of Wang and Crowcroft [Wang ea., 1996]:

  1. First the minimum bandwidth (r_l) of a link during the interval is computed.

  2. Next a $r \in \{r_l\}$ is chosen, provided that it is equal or larger than the requested bandwidth. Each link with an $r_l < r$ is removed and a shortest path (based on a rate-based weight) is computed. If this shortest path obeys the delay-constraint the algorithm stops, otherwise it picks a new $r \in \{r_l\}$ and repeats the procedure.

  3. If all $r \in \{r_l\}$ are examined and no solution has been identified: the connection is not feasible.

  The authors end by discussing a network environment where pricing and priorities are the mechanisms used to integrate support for both advance and immediate reservations. In particular, reservations are assigned priorities that determine the order in which they can be pre-empted in case of insufficient resources.

- Hassin [Hassin, 1992] presents a fully polynomial approximation scheme (FPAS) to solve the DCLC problem. His algorithm is based on a technique called rounding-and-scaling. The general idea is to first devise an optimal (non-/pseudo-polynomial) algorithm, whose complexity is proportional to the largest possible

value of the delay/cost. If the set of possible delay/cost values is scaled down to a small enough range, then the scaled problem can be solved optimally in polynomial time. The solution is then rounded back to the original delay/cost values with some bounded error.

A directed graph $G(V,E)$ with a vertex set $V = \{1,\ldots,n\}$ and an edge set E is given. Each edge $(i,j) \in E$ has a cost c_{ij} and delay d_{ij}. These numbers are assumed to be positive integers. Hassin uses the following exact algorithm:

Algorithm EXACT: Denote by $g_j(c)$ the delay of a quickest 1-j path whose cost is at most c, then:

$g_1(c) = 0, \qquad c = 0, \ldots, OPT$

$g_j(c) = \infty, \qquad j = 2, \ldots, n$

$$g_j(c) = \min\left\{ g_j(c-1), \min_{k|c_{kj} \le c}\left\{ g_k\left(c - c_{kj}\right) + d_{kj} \right\} \right\}, \; j = 1, \ldots, n \qquad c = 1, \ldots, OPT$$

Note that OPT, the cost of the DCLC path, is not known a priori, but it satisfies $OPT = \min\{c \mid g_n(c) \le D\}$, where D is the delay-constraint. To approximate OPT, Hassin starts with easily computable upper and lower bounds (UB respectively LB). For example $LB = 1$ and UB = sum of (n-1) biggest costs. If $UB \le (1 + \varepsilon)LB$ then UB is an ε-approximation to OPT. Suppose now that $UB > (1 + \varepsilon)LB$. Let B be a given value $LB < B < UB(1 + \varepsilon)^{-1}$. Now TEST($B$) can be applied to improve the bounds on OPT.

TEST(B):

1. Set c <- 0

   For all $(i,j) \in E$

   if $c_{ij} > B$, set E <- $E\backslash\{i,j\}$

   else, set c_{ij} <- $\lfloor c_{ij}(n-1)/B\varepsilon \rfloor$

2. If $c \ge (n-1)/\varepsilon$ output YES

   else use algorithm EXACT to compute $g_j(c)$ for $j = 2, \ldots, n$

   if $g_n(c) \le D$ output NO

   else set c <- $c + 1$ and repeat step 2.

If TEST(B) outputs YES then $OPT \ge B$, if TEST(B)=NO $OPT < B(1 + \varepsilon)$.

Hassin's FPAS is constructed as follows.

FPAS:

0. Set LB and UB to their initial values

1. If $UB \leq 2LB$ go to step 2.

   Let $B = \sqrt{LB \cdot UB}$

   if TEST(B) = YES, set $LB = B$

   if TEST(B) = NO, set $UB = B(1 + \varepsilon)$

   repeat step 1.

2. Set $c_{ij} <- \lfloor c_{ij}(n\text{-}1)/\varepsilon LB \rfloor$

   apply EXACT to compute an optimal D-constrained path

   output this path.

The error introduced is at most εLB.

- In [Ergün ea., 2000] Ergün ea. study a network model in which each network link is associated with a set of delays and costs. The costs *are a function* of the delays and reflect the prices paid in return for delay-guarantees. Such a cost structure can model a setting in which the service provider provides multiple service classes with different price and delay guarantee for each class. In this setting they study 2 problems:

  1. Constrained minimum cost path: We are to choose a *s-d* path and determine the delay bound to be required from every link along this path. The goal is to minimise the sum of the link costs along the path subject to the end-to-end delay constraint.

  2. Constrained minimum cost partition: *s-d* path P of p links is already chosen. We are to determine the delay bound to be imposed on every link along path P such that the sum of the link costs is minimised subject to the end-to-end delay constraint.

  For these two problems they present 3 polynomial-time ε-approximation algorithms that are based on Hassin's approximation algorithm [Hassin, 1992]. We will just briefly explain the two ε-approximations used to solve the PATH-problem. The first algorithm transforms the graph $G(V,E)$ into graph $G'(V,E')$ by replacing each link $e \in E$ by D (= delay-constraint) parallel links $e_1, \ldots, e_D \in E'$, where e_i has cost $c_e(i)$ and delay i. Then Hassin's algorithm is used to find the best path. This path is transformed to a path in $G(V,E)$ by replacing link e_i with link e with delay i and cost $c_e(i)$. The second algorithm works in a similar manner, but considers fewer links, which results in a smaller running time but larger error.

- Orda and Sprintson in [Orda ea., 2000] present a polynomial precomputation scheme that provides ε-optimal solution for the restricted shortest path problem (RSP basically is the same as DCLC with also a constraint on the cost). This precomputations scheme works in two phases. The algorithm in phase one, RSP-GEN, uses a cost quantisation approach. It considers only a limited number of "budget" values, namely 1, C_1, C_2, …, where $C_i = \delta^i$ for some $\delta > 1$. For each node $v \in V$ and for each i, the algorithm outputs a path from s to v, whose cost is at most C_i. The algorithm basically works as follows:

  The algorithm starts with a zero budget. The approximate weights $W_v[0]$, of the paths between s and v are set to infinity except of course $W_s[]$, which equals 0. Next on each iteration, the value of the budget is incremented and it is checked whether this higher cost yields a smaller weight. I.e. each node v with outgoing links (v,u) tests whether the best path to u found so far can be improved by going through v under the current budget restriction C_i and, if so, updates the best path for node u. The algorithm keeps iterating as long as the increasing budget restrictions do not exceed the bound C.

  The second phase is invoked at a source node s, upon a connection request between s and a destination node d, with a QoS requirement w. The scheme then determines the minimum C_i for which $W_d[C_i] \leq w$ and derives the corresponding output.

  Orda and Sprintson conclude their article by showing that their precomputation scheme reduces the computational complexity compared to "standard" approaches where for each connection request a suitable path needs to be computed.

- The majority of QoS schemes proposed so far require periodic exchange of link QoS state information among network routers to obtain a global view of the network state. During the interval where no information is exchanged, the network (due to its dynamic nature) changes and the global view of the network that the routers have, becomes outdated. The larger this interval becomes the faster the performance of global state QoS routing schemes degrades. Therefore Nelakuditi ea. [Nelakuditi ea., 2000] propose to use a localised approach to QoS routing (Guérin and Orda [Guérin ea., 1999] have also investigated how to cope with inaccurate information provided by global QoS routing schemes. Their goal

was to determine the impact of such inaccuracy on the ability of the path selection process to successfully identify paths with adequate available resources.). The fundamental question in the design of localised QoS routing schemes is how to perform path selection based solely on a local view of the network QoS state so as to minimise the chance of a flow being blocked as well as to maximise the overall system resource utilisation. Assuming that the only information available at the source are route-level statistics, such as the number of flows blocked, they propose to use adaptive proportional routing. Based on the available statistics, adaptive proportional routing attempts to proportionally distribute the load from a source to a destination based on their perceived quality (e.g. observed flow blocking probability). The adaptive proportional routing scheme proposed and evaluated is named proportional sticky routing (psr). The psr scheme can be viewed to operate in two stages:

1. Proportional flow routing
2. Computation of flow proportions

The proportional flow routing stage proceeds in cycles of variable length. During each cycle incoming flows are routed along paths selected from a set of eligible paths (this is done with a weighted-round-robin-like path selector). A path is selected with a frequency determined by a prescribed proportion. A number of cycles form an observation period, at the end of which a new flow proportion for each path is computed based on its observed blocking probability. This is the computation of the flow proportion stage.

2.1.2 Generic algorithms:

- Jaffe [Jaffe, 1984] gives two approaches to solve or approximate the MCP problem for $m = 2$. The first approach uses a non-deterministic algorithm to determine whether a path meets the constraints. The large running time of this algorithm makes this approach unattractive for use in real-time systems. In the second approach, the two metrics are transformed into a single metric by using a linear equation. A shortest path algorithm then finds a path based on this single metric. Different linear equations are evaluated and a bound for their worst-case overshoot is given.

  Although Jaffe only discusses the MCP problem for two constraints, his algorithms are easily extended to cope with multiple constraints ($m \geq 2$). A

generalisation of Jaffe's analysis towards multiple constraints is given in [Andrew ea., 1998]. Jaffe's "linear" algorithm is further discussed in 2.2.

- Chen and Nahrstedt [Chen ea., 1998a] target the complexity of the MCP problem by approximating the real values of the link metrics by bounded integer values and then use dynamic integer programming to solve it in polynomial time (see appendix A). The integer values are gained by the following formula:

$$l'_i(u,v) = \left\lceil \frac{l_i(u,v) \cdot x}{L_i} \right\rceil$$

where x is a positive bounded integer. x is a tuning parameter: the higher x, the higher the probability of finding the solution. The solution to this simpler problem is also a solution to the actual problem, which was proved as follows:

$$l'_i(u,v) = \left\lceil \frac{l_i(u,v) \cdot x}{L_i} \right\rceil \Leftrightarrow l'_i(u,v) \geq \frac{l_i(u,v) \cdot x}{L_i} \Leftrightarrow l_i(u,v) \leq \frac{l'_i(u,v) \cdot L_i}{x}$$

therefore

$$l_i(P) = \sum_{(u,v) on P} l_i(u,v) \leq \sum \frac{l'_i(u,v) \cdot L_i}{x} = \frac{L_i}{x} \sum l'_i(u,v) = \frac{L_i}{x} l'_i(P) \leq \frac{L_i x}{x} = L_i$$

However, the solution to the actual problem might not be a solution to the simpler problem, which deems this algorithm to be a heuristic.

- The algorithm in [Iwata ea., 1996] (and a similar in [Lee ea., 1995]) tries to find a path optimised for one of the metrics, instead of optimisation for a single cost of weighted QoS parameters. The obtained path is further evaluated as to whether it can guarantee the other user QoS requirements or not. If no path can be obtained, it chooses another metric for optimisation and tries to find another path optimised for this metric iteratively, until an appropriate path is found or until all attempts are exhausted and the call is blocked.

  As shall be illustrated in 2.2, this approach may, to some extend, be seen as a special case of Jaffe's linear approach.

- TAMCRA [De Neve ea., 2000] and its successor SAMCRA seem most promising to solve the multiple constraints problem, since they are flexible in the sense that they can handle any number of constraints with good/exact results. These algorithms are chosen as a basis for this thesis and will therefore be explained in depth.

2.2 Existing algorithms that deal with the multiple additive constraints problem

The algorithms in this class can be divided into two subclasses, namely the algorithms that choose one of the existing metrics to work with and the algorithms that define a new metric. The first type of algorithms chooses what seems to be the most important metric (out of *m* metrics) and applies a classical shortest path algorithm to it. The other *m*-1 metrics are ignored. Once the classical shortest path algorithm has found a path, the algorithm verifies if all the constraints are met. If this is not the case the algorithm chooses another metric with which it performs the cycle again. This cycle is repeated until a path is found which satisfies the constraints or until a certain specified number of metrics has been used. The problem with this approach is that there is no guarantee that the shortest path for one constraint lies close to the path that satisfies all constraints.

The second type of algorithms applies a function to the multiple metrics in order to achieve a new metric on which a classical shortest path algorithm might be used. Two main contributions in this area came from Henig [Henig, 1985] and Jaffe [Jaffe, 1984]. Henig used the concept of non-dominated paths to find a suitable path in a two-constraint problem. This concept of non-dominated paths (paths that are not dominated by another path) is very useful and is used in the TAMCRA algorithm (see 2.3). Basically, a path is called non-dominated when there does not exist another path whose costs (all of them) are smaller than those of the non-dominated path. Jaffe presented an algorithm to solve the two-constraint problem (constraints L_1 and L_2). He proposed to replace the two link metrics by a single link value, which is a linear combination of the original metrics:

$$l(u,v) = d_1.l_1(u,v) + d_2.l_2(u,v) \quad (1)$$

, where d_1 and d_2 are positive numbers.

This linear representation of the link metrics allows for Dijkstra to be used. Dijkstra's algorithm will now find a path for which

$$\begin{aligned} l(P) &= [d_1.l_1(s,k) + d_2.l_2(s,k)] + ... + [d_1.l_1(l,d) + d_2.l_2(l,d)] \\ &= d_1.[l_1(s,k) + ... + l_1(l,d)] + d_2.[l_2(s,k) + ... + l_2(l,d)] \\ &= d_1.l_1(P) + d_2.l_2(P) \end{aligned} \quad (2)$$

is minimal. Figure 2 visualises this by displaying the length of all possible paths between a source and destination node in a plane.

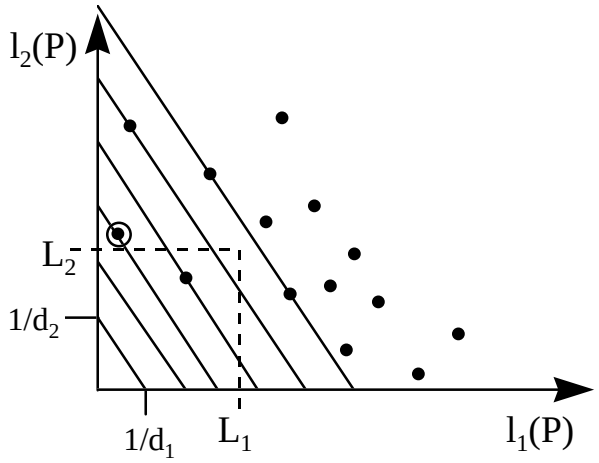


Figure 2. Distribution of the paths in the $l_1(P)$-$l_2(P)$ plane. Dijkstra's scanning procedure first encounters the encircled node, which accordingly is the one with minimal length.

The parallel lines in figure 2 represent a constant path length:

$$d_1.l_1(P) + d_2.l_2(P) = c \tag{3}$$

, where c is a constant.

All the path lengths above a certain line are larger than the length represented by that line. I.e. Dijkstra's algorithm scans the plane by increasing the parallel lines (with slope d_1/d_2[1]). The first node to intersect with such a line is considered to be the node with a minimal length. According to Jaffe, the values of d_1 and d_2 should be chosen such that:

$$d_1 / d_2 = \sqrt{L_2 / L_1} \tag{4}$$

, because it minimises the worst-case overshoot of the constraints. Figure 2 already showed some overshoot, because the path found by Jaffe's algorithm does not lie within the constraints (only one path satisfies these constraints). In a worst-case situation, as illustrated by figure 3, we could have only one path, satisfying the constraints, lying at the outer boundaries with length $l = d_1L_1 + d_2L_2$ and an other path with a length infinitesimal smaller lying outside the constraints. In that case Jaffe's algorithm would choose the path with the shortest length, thereby discarding the only path satisfying the constraints.

[1] If one uses the "1 out of m metrics" method, this is equivalent to assigning one d_i the value 1 and the rest 0. The scanning procedure is repeated for each $i = 1, \ldots, m$. For the example in figure 2, this method would not find a path within the constraints.

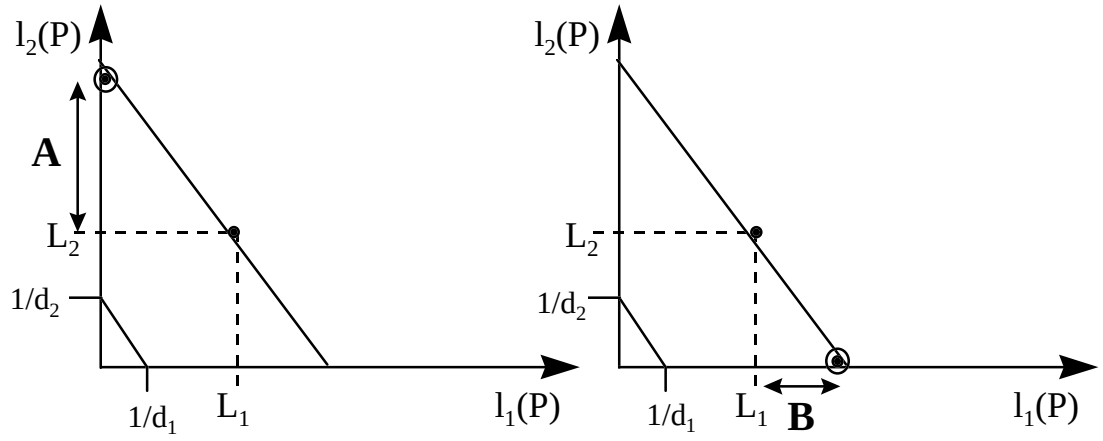


Figure 3: Two worst-case situations for the multiple constraints problem. There is only one path within the constraints with length vector (L_1,L_2). Another path is situated at the outer boundaries and has a length $l = d_1.l_1(P)+d_2.l_2(P)$ which is infinitesimal smaller than the length $l' = d_1.L_1+d_2.L_2$.

This illustrates that when scanning the solution space with a straight equilength line, this can result in a solution that lies outside the constraints even when there still exist solutions that do satisfy the constraints. Therefore it is desirable to reduce the area that is scanned outside the constraints. This can be done by choosing another slope as the one proposed by Jaffe or by using for instance curved equilength lines (see figure 4).

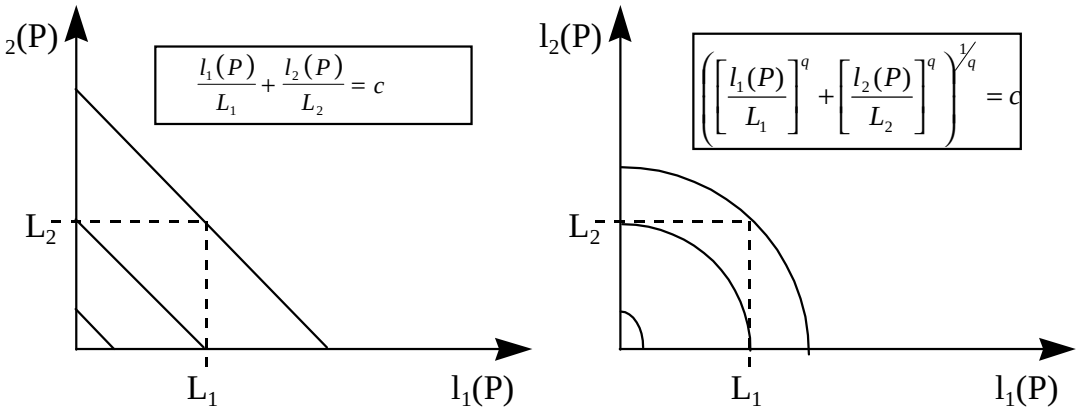


Figure 4. (a) straight equilength lines, (b) curved equilength lines

It is obvious that the curved equilength lines scan the area within the constraints in a more efficient manner. Such curved equilength lines obey another definition of the path length,

$$l(P)=\left(\sum_{i=1,\ldots,m}\left(\frac{l_i(P)}{L_i}\right)^q\right)^{1/q} \qquad (5)$$

Definition (5) is also known as Holder's q-vector norm [Golub ea., 1983] and is fundamental in the theory of classical Banach spaces, see [Royden, 1988].

Ideally the curved lines should match the shape of the outer boundaries, i.e. the scanning procedure resembles a cube that is growing from the bottom left corner towards the boundaries (the constraints) without ever crossing them (see figure 5).

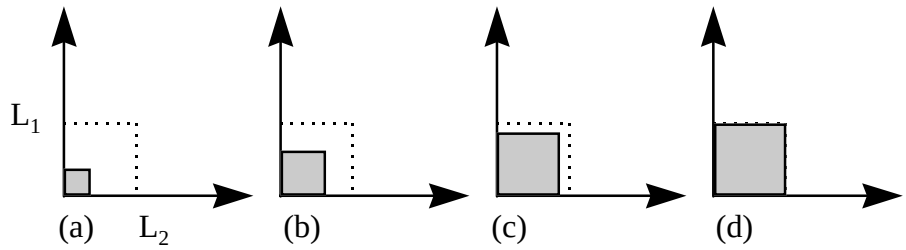


Figure 5: Scanning the constraint area when the length is defined as in (6)

This is achieved when $q \to \infty$ in (5). In that case the value of *l(P)* will be completely dominated by the largest term in the sum and expression (5) becomes equivalent to:

$$l(P) = \max\left(\frac{l_1(P)}{L_1}, \frac{l_2(P)}{L_2}, \ldots, \frac{l_m(P)}{L_m}\right) \qquad (6)$$

This equation can be interpreted as follows: if one of the components of the link vector *l(P)* (normalised with the constraints) is larger than one, the solution does not satisfy the constraints. On the other hand, if the maximum element in equation (6) is smaller than 1, then all the other elements also lie within the constraints and the solution therefore satisfies the constraints. Hence the problem of finding a path obeying the *m* constraints is "reduced" to the problem of finding the shortest path according to (6). (6) treats all *m* constraints as equally important, however (6) is extendable to any non-linear transformation *F* of the *m*-vector components to a positive real number, provided *F* satisfies the criteria of length. For example, some QoS parameters may be regarded more important than others and even min/max parameters may be considered. Of course the difficulty lies in the motivation of *F* (see appendix B). For TAMCRA (and SAMCRA), we limit the focus to the path length definition as given by (6). This definition of the path length already differs from the one used by Jaffe. This however is not the only difference. Jaffe's algorithm transforms, via equation (1), the link metrics into a single metric and then uses Dijkstra's algorithm to find the shortest path. Using the same concept, i.e. a combination of equation (6) and Dijkstra's algorithm, however does not guarantee that the shortest path will be found. This is a consequence from the non-linear path length definition (see corollary 1).

Corollary 1: *Sub-paths of shortest paths in multiple dimensions (m>1) are not necessarily shortest paths*

This is an important corollary that follows from non-linear path length definitions as (5) and (6).

For validating corollary 1, we use the following fundamental properties of the length of a vector p [Golub ea., 1983]:

- $l(p)>0$ for all non-zero vectors and $l(p)=0$ only if $p=0$ (i.e. all components of p are zero).
- for all vectors, p and q obey the triangle inequality:
  $$l(p+q) \leq l(p) + l(q) \quad (7)$$
- If p and q are non-negative vectors (i.e. all vector components are non-negative):
  $$l(p+q) \geq l(p) \quad (8)$$

Proof:

Consider two paths P_1 and P_2 for which $l(P_1) < l(P_2)$ and assume that, by adding a same link a to both paths, we can construct the paths P_3 and P_4.

Because the proof of this corollary requires the use of the triangle inequality, we focus on two cases: one where the equality sign holds (typically for $q=1$ in (5)) and the other for the inequality sign (typically for $q>1$ in (5)).

1. We have $l(P_3) = l(P_1+a) = l(P_1) + l(a)$ and $l(P_4) = l(P_2+a) = l(P_2) + l(a)$. Since $l(P_1) < l(P_2)$ it is easily verified that $l(P_3) < l(P_4)$, or in words, the sub-paths of a shortest path with a linear definition of the path length are again shortest paths.
2. Now we have the following inequalities:
   $l(P_3) = l(P_1+a) < l(P_1) + l(a)$
   $l(P_4) = l(P_2+a) < l(P_2) + l(a)$
   From this set of inequalities it can not be concluded whether $l(P_3) < l(P_4)$ *or* $l(P_3) \geq l(P_4)$. It suffices to show that the latter situation exists in order to prove the corollary. Such a situation occurs when we assign the vector [4, 4, 4] to P_1, [5, 2, 1] to P_2 and [2, 5, 1] to a. Given the constraints [10, 10, 10] and the length of a vector as defined by (6), we find:
   $l(P_1) = 0.4 < l(P_2) = 0.5$ (our initial assumption)
   However $l(P_3) = 0.9 > l(P_4) = 0.7$, which shows that the shortest path P_4

does not necessarily consist of shortest sub-paths (P_2 is not a shortest path!). QED.

2.3 TAMCRA: Tunable Accuracy Multiple Constraint Routing Algorithm

2.3.1 TAMCRA algorithm

Sub-paths of shortest paths in multiple dimensions (m>1) are not necessarily shortest paths. This means that if we want to find the shortest path between a source and destination node, we should not only consider the shortest sub-paths but also other paths, e.g. the second shortest paths. Two examples, provided by Hans De Neve and Piet Van Mieghem, illustrate this for a given network. The first example demonstrates what happens when you run the Dijkstra algorithm with a path length as defined in (6). The second example runs an extended version of this modified Dijkstra algorithm. This extended version keeps track of two paths.

Example 1:

Consider the graph shown in figure 6. Each link is characterised by three weights, each of which corresponds to an additive metric. The problem is to find a path between the source node *a* and the destination node *i* that satisfies all constraints which for this example were chosen to be respectively 14, 11 and 22.

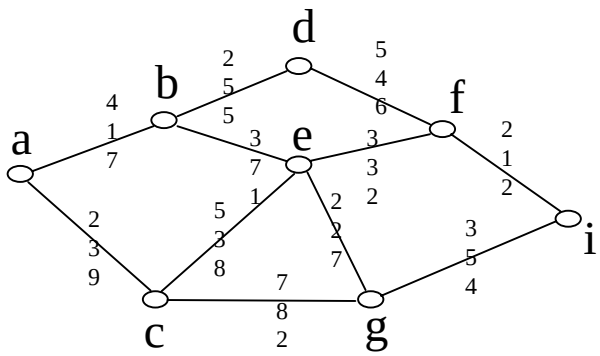


Figure 6: example graph with three metrics per link

The algorithm first calculates all individual path lengths to the nodes neighbouring the source node *a* and divides those path lengths by the constraint for the corresponding metric. This is shown in figure 7.

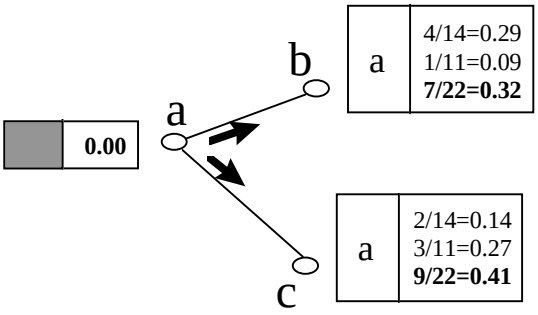


Figure 7: scanning the neighbours of node *a*

Each node keeps a table. The first column of the table indicates the previous node. This is necessary to retrieve the (reverse) path to the source node. The second column lists the three path lengths divided by their corresponding constraint. The algorithm only considers the largest of the three ratio's (in bold), which is the path length as defined in (6) and stores them in a queue. As in the Dijkstra algorithm, the minimum path length is selected from the queue and the corresponding node is marked (grey). In this case the shortest path length in the queue is 0.32 from node *b*. From this node the path lengths to its neighbouring nodes are calculated. Node *b* has three neighbouring nodes (*a*, *d* and *e*) but only those nodes are considered which are not the previous node or which are not marked. This is shown in figure 8.

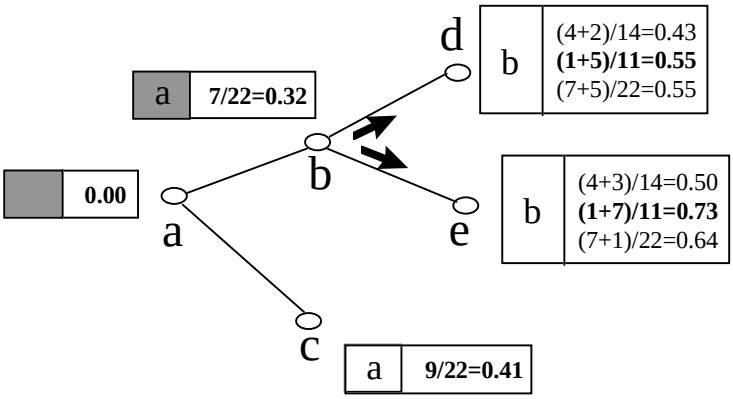


Figure 8: scanning neighbours of node *b*

The algorithm again selects the node with the smallest number in bold, which is 0.41 for node *c* in this case and calculates the paths to the neighbouring nodes. This time there is a new path to node *e* and only the path that leads to the smallest number in bold is maintained.

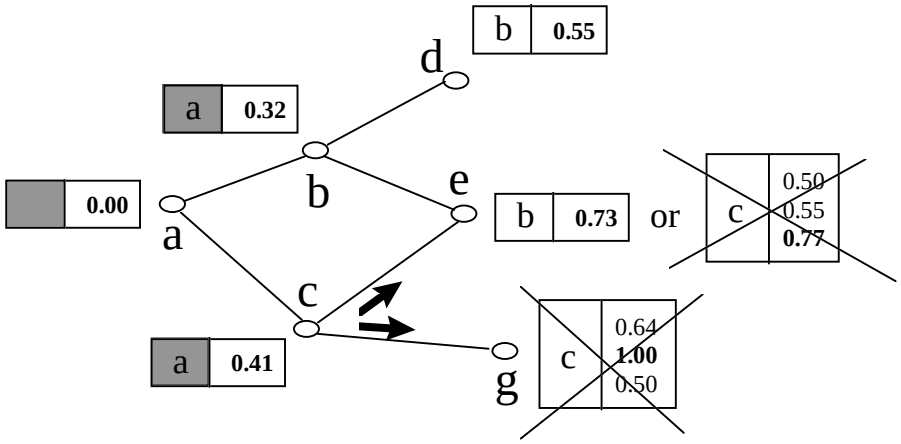


Figure 9: scanning neighbours of node *c*

In this case also the path to node *g* which is characterised by (0.64, 1.00, 0.50) can be discarded because the second constraint is already reached while node *g* is still at least one link away from the destination node *i*. This only leaves the choice between the bold numbers 0.55 for node *d* and 0.73 for node *e* which means the neighbouring nodes of node *d* are scanned next.

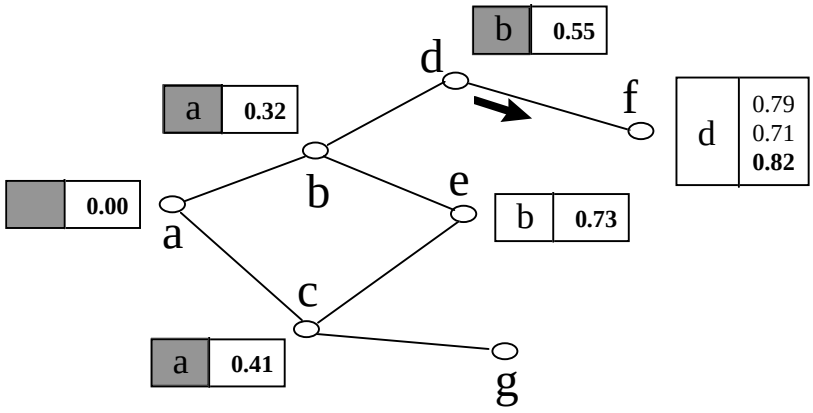


Figure 10: scanning neighbours of node *d*

This time the bold number of node *e* is smallest and its neighbouring nodes are scanned. This returns a path to node *g* which does lie within the constraints while there is a second path for node *f* but it is discarded because the path length (in bold) is larger than the existing path. In fact, its path length equals 1, which means it can be discarded anyhow, as we did with the first path to node *g* in Figure 9.

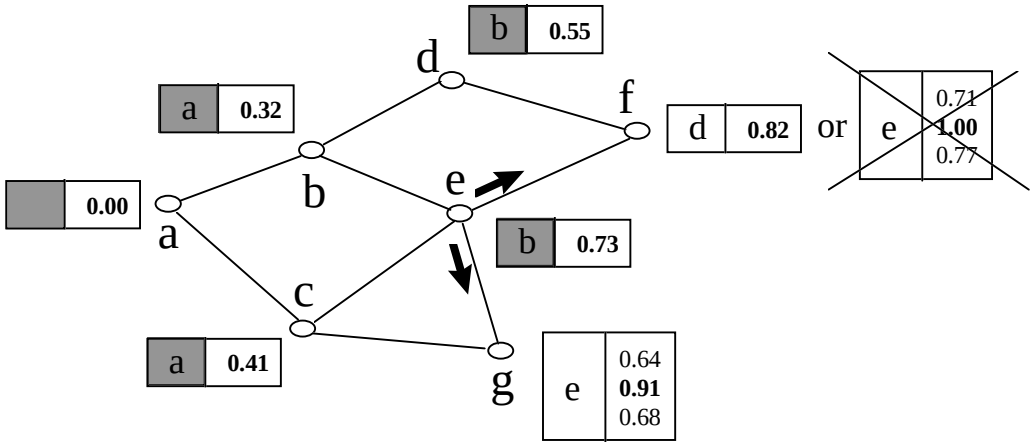


Figure 11: scanning neighbours of node *e*

Next the neighbouring nodes of node *f* are scanned which is node *i* only.

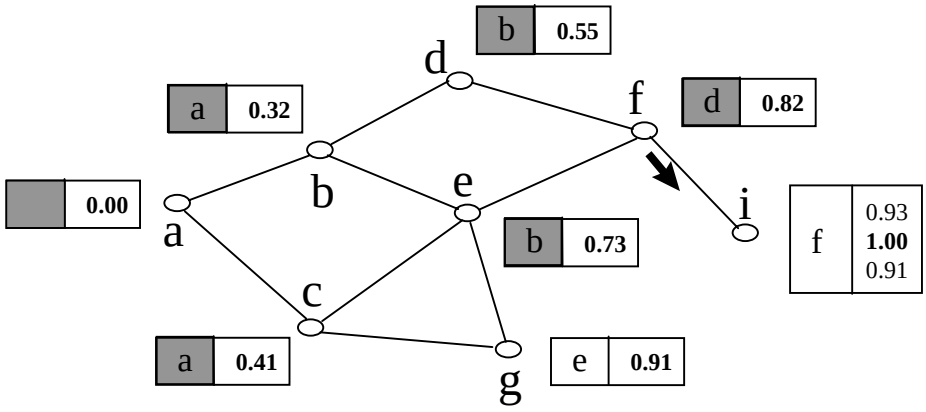


Figure 12: scanning neighbours of node *f*

The path length of node *g* is the now the smallest value left in the queue and again only node *i* is the neighbouring node which is considered, but the new path length for node *i* falls outside the constraint region. This is shown in figure 13.

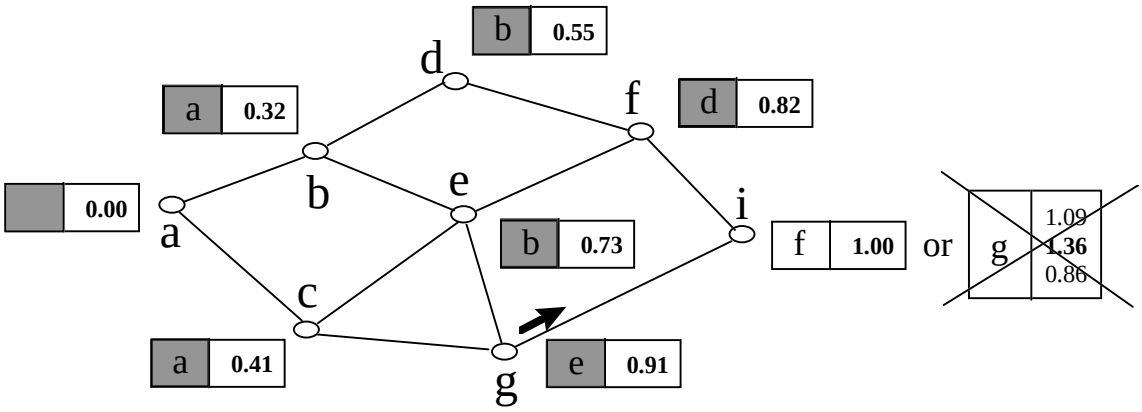


Figure 13: scanning neighbours of node *g*

The path length for node *i* is the only path length left in the queue so it is selected from the queue. As this path length belongs to the destination node, the procedure ends. The path is found by following the trail indicated by the pointer to the previous node: $i \rightarrow f \rightarrow d \rightarrow b \rightarrow a$. In this way we have retrieved a path which satisfies all constraints, using the Dijkstra algorithm and the path length as defined in (6).

Example 2:

An extended version of the modified Dijkstra algorithm. This example will use the same graph as in example 1. The only difference between this example and the previous one is that we allow each node to have two entries in the queue instead of one. This time we will only show the path length (i.e. the largest of the three numbers, which was the one in bold before). We start with figure 14; the first time that there are two entries at a node.

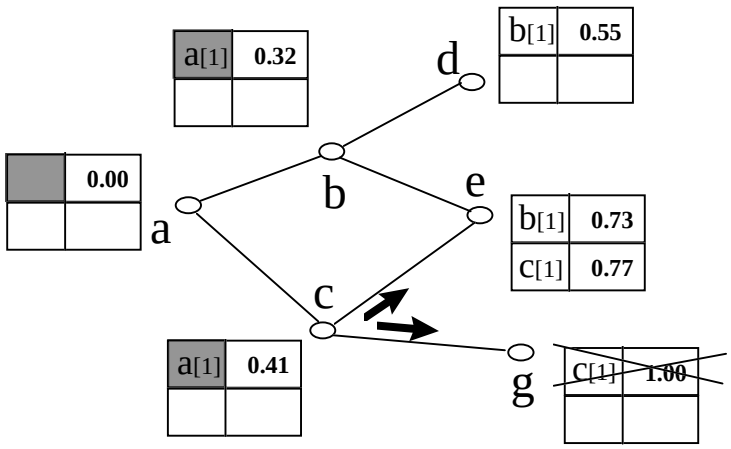


Figure 14: scanning neighbours of node *c*

Starting from this situation we proceed as before. Remember that the entries which have been marked grey are those which have been removed from the queue. There are now three path lengths in the queue: 0.55, 0.73 and 0.77. Node *d* has the smallest length in the queue and thus its neighbouring nodes are scanned, indicated by the arrows. Although node *d* has two neighbours and both neighbours still have an unmarked field in their table, node *b* is not scanned because it is the previous node of node *d*.

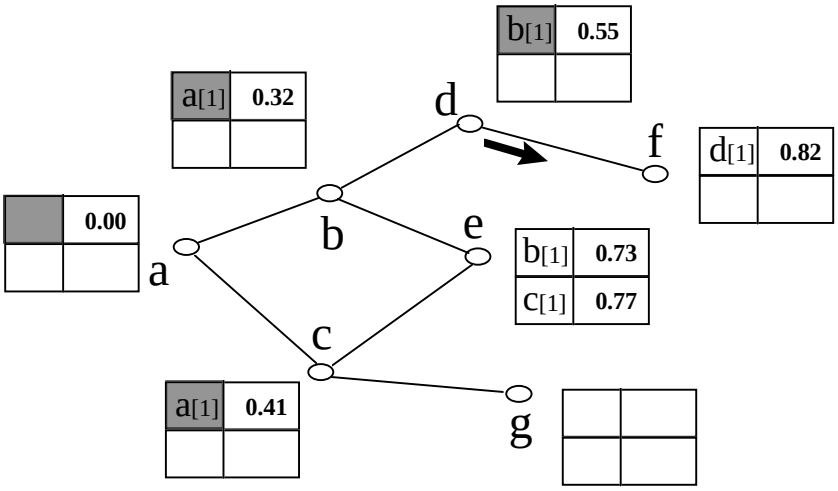


Figure 15: scanning neighbours of node *d*

When this is done, the shortest length in the queue belongs to the first entry in the table of node *e* (0.73). The previous node of the first entry of node *e* is node *b* so all the neighbours of node *e* are scanned except node *b*.

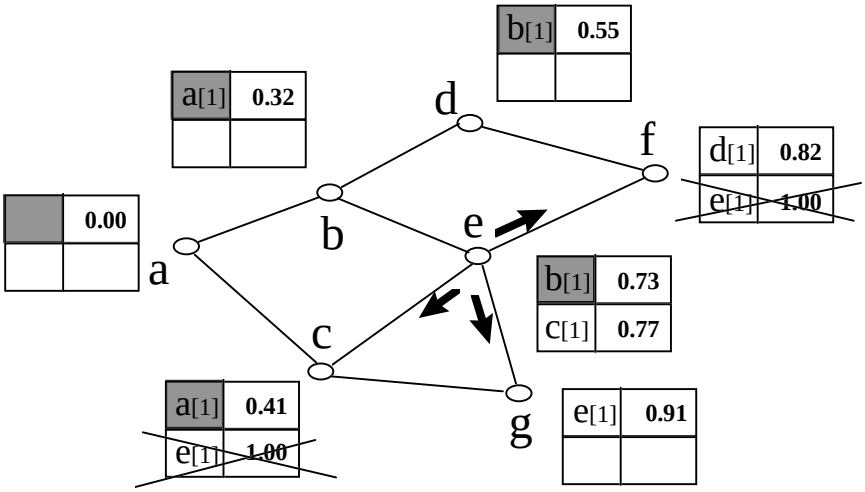


Figure 16: scanning neighbours of node *e*

Next, the second entry in the table of node *e* is the shortest path, which is still in the queue. For this entry, the previous node is node *c*. So we scan the neighbouring nodes of node *e* except for node *c*. This is shown in figure 17.

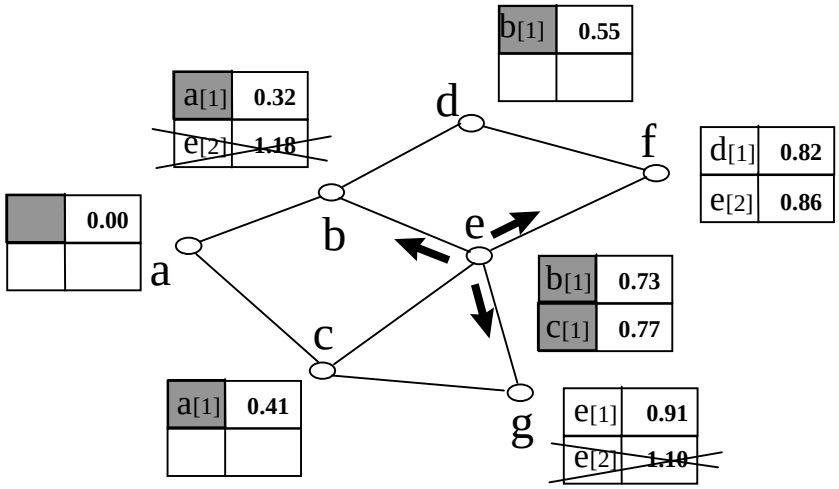


Figure 17: scanning neighbours of node *e*

Two of the three new path lengths can immediately be discarded because they exceed the constraints. The first entry of node *f* is the following path length, which is selected from the queue. The neighbouring node of node *f* is only node *i* because both entries of node *e* have been marked and because node *d* is the previous node of the first entry.

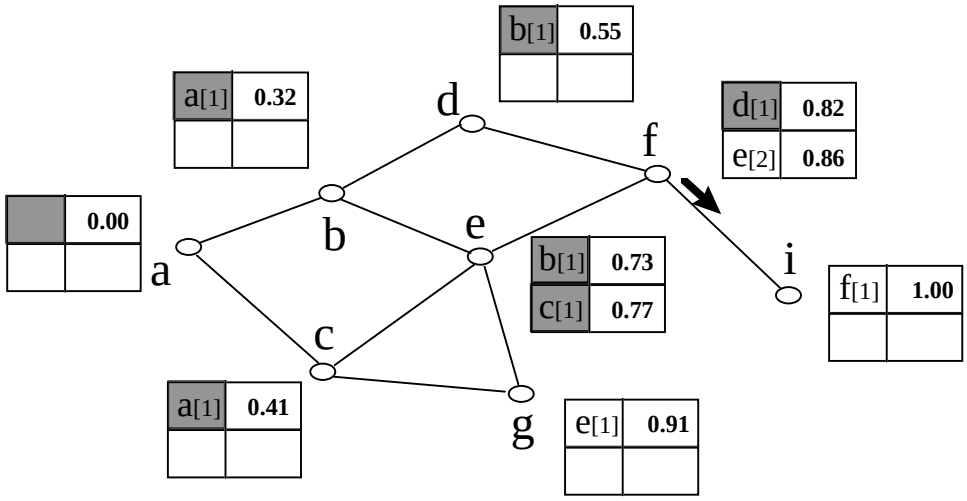


Figure 18: scanning neighbours of node *f*

Then the neighbouring nodes of node *f* are scanned again but this time referring to the second entry of the table. This means that besides node *i* also node *d* is considered because the previous node for the second entry of node *f* is node *e*.

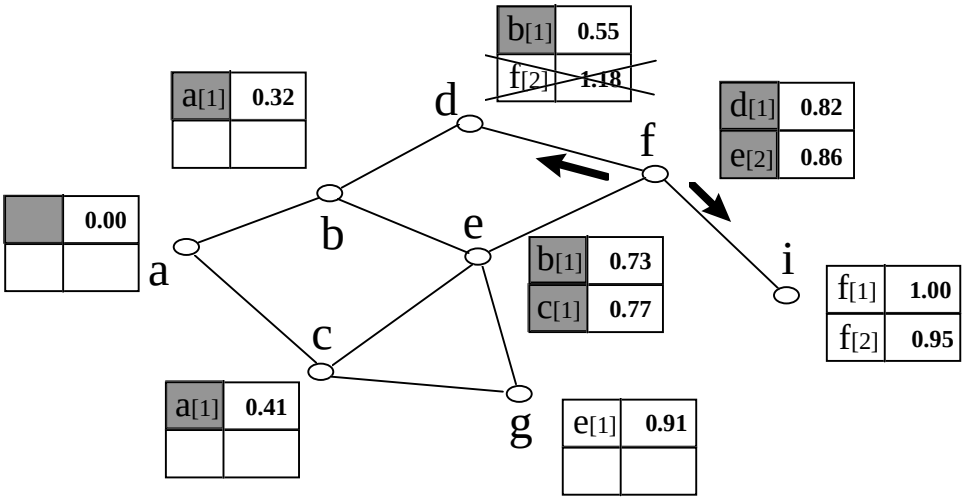


Figure 19: scanning neighbours of node *f*

Next, the adjacent nodes of node *g* are scanned but this does not yield a shorter path to the destination node.

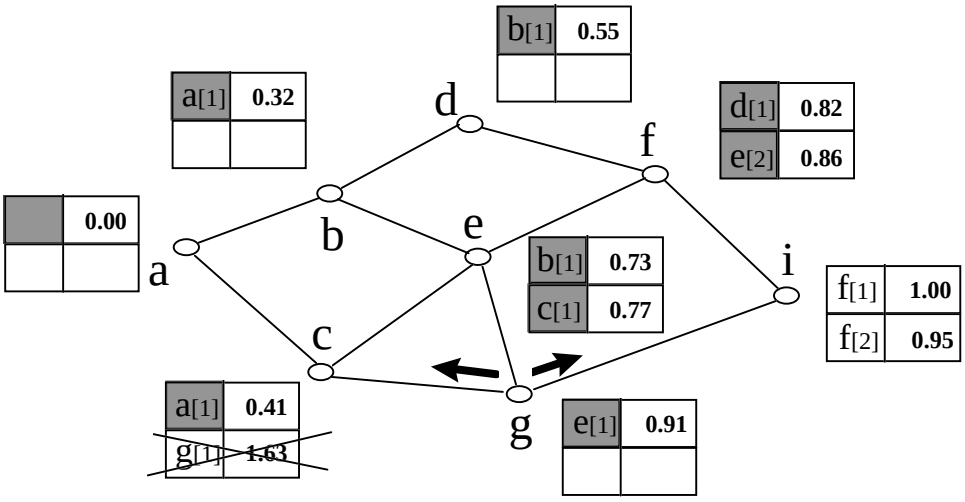


Figure 20: scanning neighbours of node *g*

The next path length which is selected from the queue is 0.95 belonging to the destination node *i*, so the algorithm stops. The path itself can be traced back to the source by following the pointers to the previous node.

The first example returned a path, meeting the constraints, with length 1.00, but the second example returned the true shortest path with length 0.95. This illustrates that we can find the shortest path if we only take enough paths into account. This is how TAMCRA works. TAMCRA has a tuneable integer *k* that represents the number of paths that are taken into account (*k* is the queue-size of each node). In example 1, *k* was equal to one and in example 2, *k* was taken equal to two. The larger

we choose *k*, the larger becomes the probability of finding the true shortest path. This "accuracy tuning index" *k* suggested to call this algorithm TAMCRA, a Tuneable Accuracy Multiple Constraints Routing Algorithm.

So far we have seen the two most important aspects of TAMCRA, i.e.: the non-linear definition of the path length given by (6) and the use of *k*-shortest paths. Besides these two aspects, TAMCRA also uses the concept of dominated paths as introduced by Henig [Henig, 1985] to increase the computing efficiency. Confining to $m = 2$ dimensions, we consider two paths P_1 and P_2 from a source to some intermediate node, each with a path length vector $(l_1(P_1), l_2(P_1)) = (x_1, y_1)$ and $(l_1(P_2), l_2(P_2)) = (x_2, y_2)$ respectively [Van Mieghem ea., 1999]. Figures 21a-b represent two possible scenarios for these two paths.

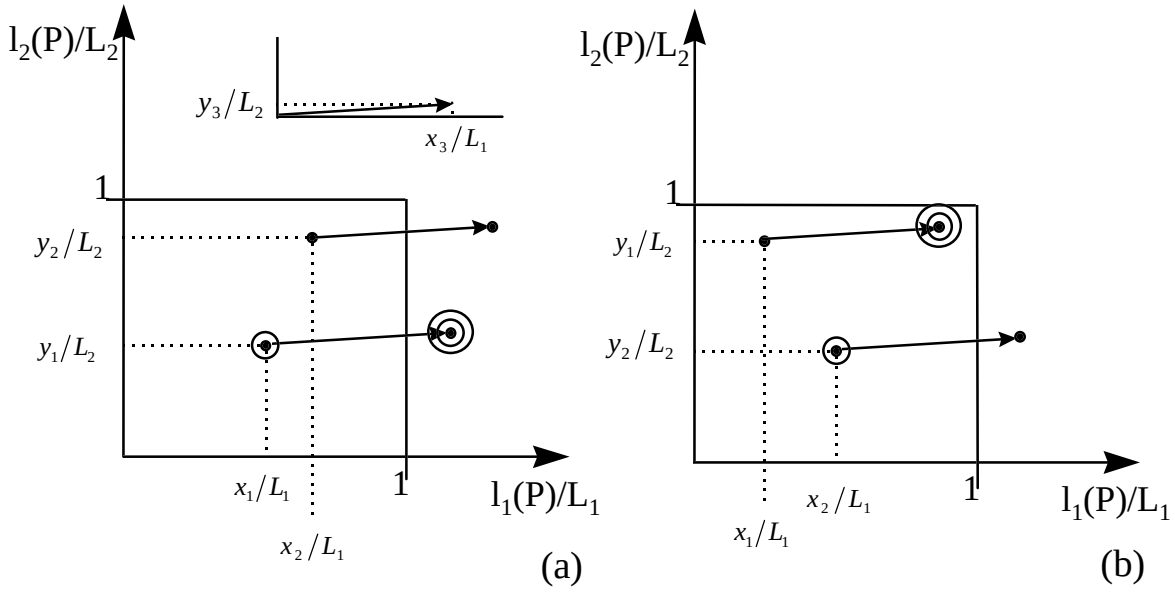


Figure 21: Representation of the two paths from the source node to some intermediate node. The length vector (x_3,y_3) represents the remaining part of the shortest path between source and destination. The shortest path is encircled twice.

In scenario (a), P_1 is shorter than P_2. Moreover, P_1 is shorter in both metric 1 (x) and in metric 2 (y). In that case, *any path from the source to the final destination node which makes use of P_1 will be shorter than any other path from this source to that destination which makes use of P_2.*

Indeed, if

$$x_1 < x_2 \text{ and } y_1 < y_2$$

then, conform the non-linear path length definition of (6) (or similarly any other function F, as illustrated in appendix B), also

$$x_1/L_1 < x_2/L_1 \quad \text{and} \quad y_1/L_2 < y_2/L_2$$

resulting in

$$(x_1 + x_3)/L_1 < (x_2 + x_3)/L_1 \quad \text{and} \quad (y_1 + y_3)/L_2 < (y_2 + y_3)/L_2$$

for any value of x_3, y_3 and thus

$$\max\left(\frac{x_1 + x_3}{L_1}, \frac{y_1 + y_3}{L_2}\right) < \max\left(\frac{x_2 + x_3}{L_1}, \frac{y_2 + y_3}{L_2}\right)$$

Hence, we certainly know that P_2 will never be a sub-path of a shortest path and therefore P_2 should not be stored in the queue. Using the terminology of Henig, P_2 is said to be dominated by P_1. Moreover if $x_1 = x_2$ and $y_1 = y_2$ then P_2 is identical to P_1. In that case both paths will yield the same results and it therefore suffices to only store one.

In scenario (b) both paths have crossing abscissa and ordinate points: P_2 has a smaller value for metric 2 (y) while P_1 has a smaller value for metric 1 (x), but $l(P_1) > l(P_2)$. In such scenarios, the shortest path (here P_2) between the source and some intermediate node is not necessarily part of the shortest path from source to destination. This is shown in figure 21-b by adding the path length vector (x_3,y_3) which completes the path towards the destination. It illustrates that P_1 (and not the shortest sub-path P_2) lies on that shortest path.

In summary, if two sub-paths have crossing abscissa-ordinate values, both values should be stored in the queue, else only the shortest should be stored. Although the demonstration was in $m=2$ dimensions, the same rule applies if there are m constraints. If path P_1 is already stored in the queue and a new path P_2 to the same intermediate node is found for which

$$l_i(P_1) \le l_i(P_2), \quad \text{for each } i=1,...,m$$

then path P_2 should not be stored in the queue. This implies that, for each new path P_n at an intermediate node, all m path length components $l_i(P_n)$ must be stored in order to be able to verify whether future, new paths to the same intermediate node are not dominated by the earlier paths.

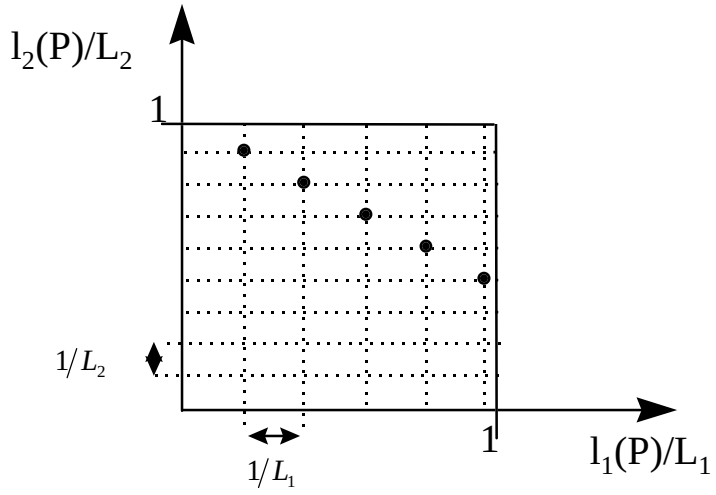


Figure 22: A worst-case situation in which L_1 partial paths are maintained in parallel

The worst-case amount of simultaneously stored paths is determined by the granularity of the constraints. In reality most protocols will only allocate a fixed number of bits per metric. In that case the constraints L_i can be expressed as an integer number of a basic metric unit. The worst-case number of partial paths which have to be maintained in parallel for each node are shown in figure 22. If $m=2$ and $L_1 < L_2$ as in figure 22, there can never be more than L_1 partial paths with crossing abscissa-ordinate values as in figure 21-a. Any additional partial path will amount to one of the L_1 partial paths as shown in either figure 23a or figure 23b.

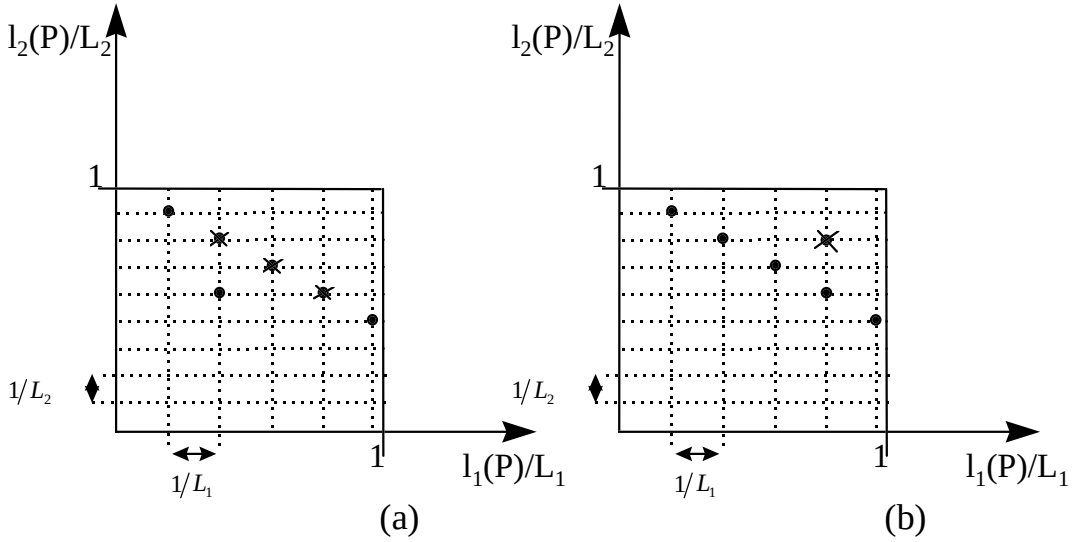


Figure 23: (a) only three partial paths should be maintained in parallel; (b) any longer path should not be maintained.

In both cases the number of paths which are stored in the queue is not increased. Thus, for m=2, the upper limit for k in the TAMCRA algorithm is

$$k_{\max} = \min(L_1, L_2)$$

If there are more than two constraints or if these constraints are very large, the maximum number of paths to be calculated is given by

$$k_{\max} = \min\left(\frac{\prod_{i=1}^{m} L_i}{\max_j \left(L_j\right)}, \lfloor e(N-2)! \rfloor \right) \tag{9}$$

The first argument of the min-operator in (9) refers to the number of relevant path vectors within the constraints surface. The second argument, where $\lfloor x \rfloor$ denotes the integer equal to or smaller than x and $e = 2.718..$, is a tight upper bound for the number of paths between 2 nodes in any graph with N nodes [Van Mieghem, 1998]. When the size of the queue in each node of the graph G is $k = k_{max}$, the multiple constraints routing problem is solved *exactly.*

Besides the concept of dominated paths, there is another way to increase the computational efficiency. This idea was formed during this thesis and therefore is not part of the original TAMCRA algorithm. This adapted version of TAMCRA takes into account the queue of the destination node, i.e. the solutions found so far. Each new entry is now first compared against the minimum entry in the destination-queue instead as done before with 1.0 (the constraint boundary). The idea behind this is that if we already have a path P_1 with length $l(P_1)$, we can discard all the other (sub)-paths P_2 with length $l(P_2) \geq l(P_1)$, because their length will never become smaller than $l(P_1)$ as the link vectors consist of non-negative additive metrics. Note that this modification of the original TAMCRA algorithm does not alter the results of the algorithm, but increases the performance. The algorithm will return the same paths, but it will store less k-shortest paths in the queue. Figure 24 shows this improvement in the use of the queue-size for 10^6 20-node networks with approximately 76 links and 2 link metrics.

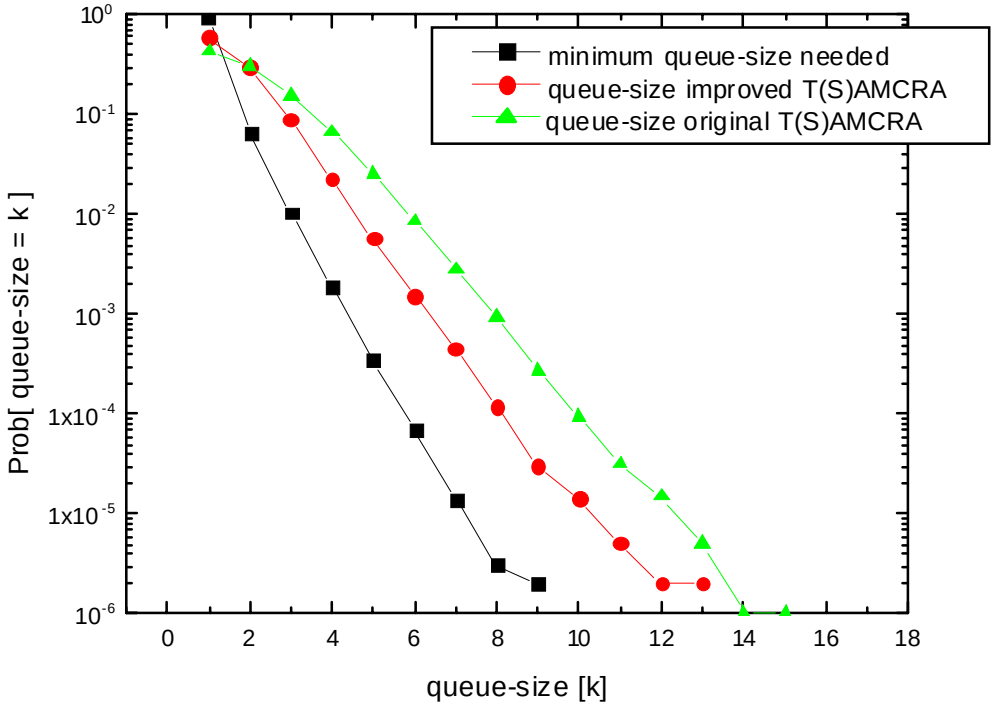


Figure 24. The queue-size that is used to retrieve the shortest path and the minimum queue-size needed (lower bound) to retrieve this path.

By tracking the minimum length in the end-queue, we can achieve a considerable decrease in the maximum queue-size (k) used by TAMCRA. This k is the maximum number of paths that TAMCRA stored in the queue of a node, given that the queue was large enough. As stated by formula (9), the worst-case k needed by TAMCRA to find the shortest path is k_{max} and therefore $k \leq k_{max}$. The plot also depicts the minimum queue-size k_{min}, that TAMCRA needed to retrieve the shortest path. When a $k \geq k_{min}$ is used, TAMCRA always retrieves the shortest path. The value of k_{min} differs per topology and was calculated by tracking the maximum number of entries per queue that were truly used by the nodes on the path. Thus the difference between the queue-size that was used (k) and the minimum queue-size (k_{min}) is that the first keeps track of the number of entries that were put into the queue, whereas the latter counts the number of entries that were extracted from the queue, because those were truly used for the calculation of the shortest path. We may consider k_{min} to be a lower bound on the queue-size and we will use this k_{min} to reflect the NP-completeness of the multiple constraints problem. Finally we can observe that there

still exists a gap between the k of the improved TAMCRA algorithm and k_{min}. In paragraph 2.4.3 some comments will be made on how this gap may be reduced.

2.3.2 TAMCRA meta-code

Considering the 4 aspects of TAMCRA, i.e. non-linear path length, k-shortest paths, non-dominated paths and tracking the destination-queue, the meta-code is listed as follows:

TAMCRA(G, s, d, L, k)
G: graph, s: source node, d: destination node, L: constraints, k: queue-size

```
1 counter = 0 for all nodes
2 endvalue = 1.0
3 s = source node, d = destination node
4 path(s[1]) = NULL and length(s[1]) = 0
5 s[1] = marked
6 put s[1] in queue
7 while(queue ≠ empty)
8     EXTRACT_MIN(queue) -> u[i]
9     u[i] = marked
10    if(u = d)
11          STOP
12    else
13          for each v ∈ adjacency_list(u)
14                if((v ≠ marked) and (v ≠ previous node of u[i]))
15                      PATH = path(u[i]) + (u,v)
16                      LENGTH = length(PATH)
17                      check if PATH is non-dominated
18                      if(LENGTH < endvalue and non-dominated)
19                            if(counter(v) < k)
20                                  counter(v) = counter(v) + 1
21                                  j = counter(v)
22                                  path(v[j]) = PATH
23                                  length(v[j]) = LENGTH
```

```
24                                    put v[j] in queue
25                              else
26                                    max_length(v) -> v[j]
27                                    if(LENGTH < length(v[j])
28                                          path(v[j]) = PATH
29                                          length(v[j]) = LENGTH
30                                          put v[j] in queue
31                              if(v = d)
32                                    endvalue = LENGTH
```

A few words to clarify this code:

Line 1 initialises the counter for each node to zero. This counter keeps track of the number of entries in the queue and therefore may not exceed *k* (see line 19). To start the algorithm with the source node, it is inserted into the queue (line 6). For our program, we chose to structure the queues of the nodes into a Fibonacci heap. For more information on how to construct and maintain Fibonacci heaps, see [Cormen ea., 1997]. The EXTRACT_MIN function in line 8 selects the minimum path length in the queue and returns the associated node *u* with its entry number *i* (= the *i*-th path stored in the queue at node *u*). The extracted element is marked in line 9. If the extracted node *u* equals the destination, the shortest path according to the algorithm can be retrieved. If this is not the case, the scanning procedure is invoked (lines 13 and 14). Line 15 describes how the path up to node *u* is extended towards the neighbouring node *v*. Line 17 performs a check on this path to see if it is not dominated by any other paths. The endvalue in line 18 keeps track of the smallest path length in the destination queue. The original TAMCRA algorithm used the value 1.0 instead of endvalue. Lines 19 to 24 describe how path(length)s for a node can be added as long as the maximum number specified by *k* is not attained. The procedure in lines 26 to 30 is called when there are already *k* non-dominated paths to a certain node in the queue. In that case, only the *k*-shortest paths are maintained. If necessary the endvalue is updated in lines 31 and 32.

2.3.3 TAMCRA's Complexity

If *V* is the number of nodes in the graph *G(V,E)*, the queue can never contain more than *kV* path lengths. When using a Fibonacci heap to structure the queue,

selecting the minimum path length among kV different path lengths takes at most a calculation time of the order $\log(kV)$ [Cormen ea., 1997]. As each node can at most be selected k times from the queue, the EXTRACT_MIN function in line 8 takes $O(kV\log(kV))$ at most. The for-loop starting on line 13 is invoked at most k times from each side of each link in the graph. Calculating the length takes $O(m)$ when there are m metrics in the graph while verifying the crossing condition takes $O(km)$ at most. Adding a path length in the queue takes O(1) while replacing a path length takes $O(k)$ at most because the largest of the k entries for that node must be found. The total complexity of the for-loop now becomes $2kE(O(m)+O(km)+O(k)) = O(kEkm)$. Adding the contributions yields a worst-case time-complexity with $k=k_{max}$ of

$$O\left(kV\log(kV)+k^2mE\right) \tag{10}$$

With a single constraint (m=1) and k=1, this reduces to the time-complexity of the Dijkstra algorithm given by $O\left(V\log V+E\right)$. This means that, for a fixed number of constraints m and a fixed value of k, TAMCRA scales like the Dijkstra algorithm as the topology $G(V,E)$ grows (i.e. for a varying number of nodes V and links E).

The complexity (10) is non-polynomial (NP-complete) when m and L_i are not bounded, as shall be further clarified under 2.4.3.

2.4 SAMCRA: Self-Adaptive Multiple Constraints Routing Algorithm

The previous paragraph clarified how the TAMCRA algorithm functions. It was shown that TAMCRA uses a tuneable integer k, which statically allocates the queue-size of each node. The smaller k is chosen, the larger becomes the probability of not finding the shortest path. To reduce the probability of not finding the shortest path to zero and to use the queue-sizes of the nodes in a more economical way, SAMCRA was created. SAMCRA, the Self-Adaptive Multiple Constraints Routing Algorithm, is the exact version of TAMCRA. SAMCRA adapts the queue-size of each node independently to its needs, i.e. all entries that satisfy the constraints and are not dominated are put in the queue. This allows SAMCRA to *always* find the shortest path.

2.4.1 SAMCRA meta-code

SAMCRA(G, s, d, L)
G: graph, s: source node, d: destination node, L: constraints

```
1 counter = 0 for all nodes
```

```
2 endvalue = 1.0
3 path(s[1]) = NULL and length(s[1]) = 0
4 put s[1] in queue
5 while(queue ≠ empty)
6     EXTRACT_MIN(queue) -> u[i]
7     u[i] = marked
8     if(u = d)
9           STOP
10    else
11          for each v ∈ adjacency_list(u)
12                if(v ≠ previous node of u[i])
13                      PATH = path(u[i]) + (u,v)
14                      LENGTH = length(PATH)
15                      check if PATH is non-dominated
16                      if(LENGTH < endvalue and non-dominated)
17                            counter(v) = counter(v) + 1
18                            j = counter(v)
19                            path(v[j]) = PATH
20                            length(v[j]) = LENGTH
21                            put v[j] in queue
22                            if(v = d)
23                                  endvalue = LENGTH
```

This code is very similar to the code of TAMCRA. The main difference is that now entries can be stored in the queue an infinite number of times (in principle, because SAMCRA will never need more than k_{max}), as long as the shortest path is not found. The lines in the TAMCRA-code that mark a node and maintain only the *k*-shortest paths therefore become obsolete.

2.4.2 Proof that SAMCRA is exact

We have claimed that SAMCRA always finds the shortest path (if this path lies within the constraints), but although this claim seemed justifiable no proof was given. This sub-paragraph will provide that proof using SAMCRA's meta-code as

listed in 2.4.1. First consider the “all-paths” algorithm that is constructed from some of the lines of SAMCRA’s meta-code. The line between lines 14 and 17 in the all-paths-code that checks for loop-freeness is not explicitly found in SAMCRA’s code, but in fact is part of line 15 in SAMCRA’s meta-code (as will be explained below).

All-paths(*G, s, d, L*)

```
1 counter = 0 for all nodes
3 path(s[1]) = NULL and length(s[1]) = 0
4 put s[1] in queue
5 while(queue ≠ empty)
6     EXTRACT_MIN(queue) -> u[i]
11          for each v ∈ adjacency_list(u)
13                PATH = path(u[i]) + (u,v)
14                LENGTH = length(PATH)
                  if(PATH is loop-free)
17                      counter(v) = counter(v) + 1
18                      j = counter(v)
19                      path(v[j]) = PATH
20                      length(v[j]) = LENGTH
21                      put v[j] in queue
```

Property 1: The all-paths algorithm examines all possible loop-free paths between source and destination and hence always finds the shortest path.

Proof:

The all-paths algorithm may be seen as an extended breadth-first search (BFS). The BFS algorithm basically works as follows: All links are characterised by a single metric that has the value 1. Initially all nodes are coloured white except for the source node, which is coloured grey. Grey means that the node has been discovered (put into the queue), whereas white means that this has not yet happened. The algorithm now chooses (extracts) a grey-coloured node with smallest length and discovers all its white neighbours, which are then coloured grey. Once all neighbours of a node are

discovered (they are all non-white) the node is coloured black. The BFS-algorithm continues until all nodes are black. In [Cormen ea., 1997] the breadth-first procedure is explained more thoroughly and it is proved that BFS discovers every node $v \in V$ that is reachable from the source s and that the path from s to v is the shortest path.

The all-paths algorithm behaves like the described BFS-algorithm except that instead of examining only the white neighbours of a grey-coloured node, it examines all of its loop-free neighbours (line 11). The neighbours that form a loop with the previously traversed path need not be examined, because those paths will never become shortest paths (see also lemma 1 and property 3). Because all-paths is an extension to BFS, we know that all nodes are at least visited once, i.e. there is at least one path to each node. However, since the all-paths algorithm is applied to graphs with multiple metrics, the first known (stored) path from s to d is not necessarily the shortest (corollary 1) as was the case with BFS. Since each node examines all of its loop-free neighbours all paths from source to destination can be written as

$s, v_i \in \text{adj}(s), v_j \in \text{adj}(v_i)\backslash\{s\}, \ldots, v_p \in \text{adj}(v_{previous})\backslash\{\text{path to } v_{previous}\}, \ldots, d$

, where $i,j,\ldots = 1,2, \ldots,$ *number of adjacent nodes* and adj(x) represents the set of adjacent nodes to x.

Since each node examines all its loop-free neighbours, all possible combinations of $i,j,\ldots$ are examined, which corresponds to all loop-free paths. If a path is not examined this means that in the combination of $i,j,\ldots$ some node, e.g. v_j, forms a loop with its previous path.

Once the list of all paths from s to d is known it is merely a matter of choosing the path with shortest length to retrieve the exact shortest path[2].

To prove that SAMCRA finds the same result as the all-paths algorithm, we will "justifiably" reduce the search-space of the all-paths algorithm to retrieve the search-space of SAMCRA.

A first feature encountered in SAMCRA's meta-code is that lines 8 and 9 cause the algorithm to stop when the extracted path belongs to the destination node. Since line 6

[2] The exactness of the all-paths algorithm has also been verified through simulations.

only extracts the path with the shortest length, all other path lengths left in the queue are equal or longer than the length of the extracted path and hence will never become the shortest path. The shortest path from source to destination has been attained and it is therefore useless to proceed with the algorithm. We have therefore justified lines 8 and 9.

A second feature is that SAMCRA discards all paths that exceed the constraints (line 16: `if(LENGTH ≤ 1.0)`). Since these paths can not accommodate the requested QoS, there is no need to examine them.

SAMCRA has two ways to remove loop-containing paths from the search-space. The first way is implemented via line 12. This line makes sure that the path does not return to the node it just came from. This way of preventing loops only eliminates loops between two adjacent nodes. Loops containing more nodes are targeted by only allowing non-dominated paths (lines 15 and 16).

Property 2: Paths that contain a loop are always dominated and therefore may be discarded.

Proof:

Consider a path P_a that was already stored at a node and a new path P_b to that node. If $l_i(P_a) \leq l_i(P_b)$ for $i = 1,\ldots,m$ then P_b is said to be dominated by P_a. Since $l_i(P_a) \leq l_i(P_b)$ for $i = 1,\ldots,m$ also $\max(\frac{l_i(P_a)}{L_i}) \leq \max(\frac{l_i(P_b)}{L_i})$ for $i = 1,\ldots,m$ (the path length definition). Hence, a dominated path will never become the shortest path and can be removed from the search-space. Moreover, by removing dominated paths, we have also eliminated the possibility of loop-containing paths to exist. This is best explained by examining figure 25.

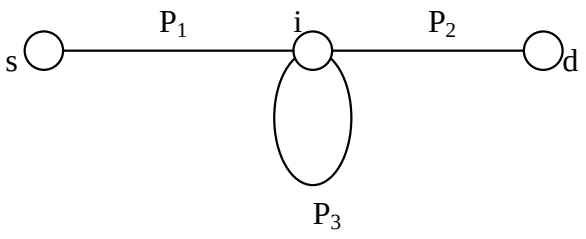


Figure 25. Example topology.

The loop-containing path P_1+P_3 is dominated by P_1 because $l_i(P_1) \leq l_i(P_1+P_3)$ for $i = 1,\ldots,m$ (see also 3.2 on routing loops).

The final search-space reduction is achieved by comparing each new entry against the minimum entry in the destination-queue. Indeed, if we already have a path P_a with length $l(P_a)$, we can discard all other (sub)-paths P_b with length $l(P_a) \leq l(P_b)$, because their length will never become smaller than $l(P_a)$.

We have argued that the search-space of the exact all-paths algorithm can be reduced to the search-space of SAMCRA without losing the shortest path and therefore proved that SAMCRA is indeed exact.

2.4.3 SAMCRA's Complexity

The calculation of the complexity of SAMCRA is analogous to the calculation performed for TAMCRA. The only difference is that for SAMCRA we do not need to replace a path length, which took O(k) at most and therefore need only O(1). This however does not alter the overall worst-case time-complexity:

If V is the number of nodes in the graph $G=(V,E)$, the queue can never contain more than kV path lengths. When using a Fibonacci heap to structure the queues, selecting the minimum path length among kV different path lengths takes at most a calculation time of the order of $\log(kV)$ [Cormen ea., 1997]. As each node can at most be selected k times from the queue, the EXTRACT_MIN function in line 6 takes $O(kV\log(kV))$ at most. The for-loop starting on line 11 is invoked at most k times from each side of each link in the graph. Calculating the length takes O(m) when there are m metrics in the graph while verifying the crossing condition takes O(km) at most. Adding a path length in the queue takes O(1). The total complexity of the for-loop now becomes $2kE(O(m)+O(km)+O(1)) = O(kEkm)$. Adding the contributions yields a worst-case time-complexity with $k=k_{max}$ of

$$O\left(kV\log(kV) + k^2 mE\right) \tag{10}$$

With a single constraint (m=1) and k=1, this reduces to the time-complexity of the Dijkstra algorithm given by $O(V\log V+E)$. When the constraints/metrics are real numbers, the granularity is infinitely small implying that the first argument in (9) is infinite and, hence, $k_{max} = O(V!) = O(\exp(V\ln V))$. In this case, the QoS routing

problem is clearly NP-complete. But, as argued before, in practice these metrics will have finite granularity, hence k_{max} is limited by the first, finite argument in (9) which does not depend on the size of the topology. Unfortunately this first argument is also NP-complete, which is best seen when we assign each constraint the same value: then $k_{max} = O(L^{m-1})$. It is obvious that this function is not polynomial in its input $O(m\log(L))$. However if m is chosen fixed, (10) becomes pseudo-polynomial. Moreover if L_i for $i = 1, \ldots, m$ would be upper bounded (by a constant or polynomial in $\log(L_i)$) (10) would become polynomial. These minor restrictions to m and L_i will almost always apply to practical applications. *Thus, in practice, the multiple constraints problem is not NP-complete at all!*

In any case, on (10), reducing k to its minimum value k_{min}, definitely should be a target. Two additional computations can be performed. Both computations consider previously stored paths, whereas before only new entries were examined. The first one occurs at line 15 (of the SAMCRA meta-code). Instead of just checking whether the new path is dominated by a previously stored path, the reverse is checked too and all dominated paths are removed. The removal of dominated paths increases the complexity with $2kE(k-1)O(logkV)$. The second way to decrease the queue-size k is to add an extra line (24) in the meta-code:

```
22    if(v=d and LENGTH < endvalue)
23          endvalue = LENGTH
24          remove all entries > endvalue
```

Since removing one entry takes $O(logkV)$, we have a worst-case scenario of $2kE(kVlogkV)$.

The total complexity of this k-optimised SAMCRA algorithm now becomes:

$$O(k^2VElog(kV) + k^2mE) \qquad (11)$$

Figure 24 showed that the difference between $k_{improved}$ and k_{min} is fairly small and the induced extra complexity of a "k-optimised SAMCRA" does not outweigh the decrease in k. In practice and in our simulations we use the meta-code of 2.4.1, because it performs better than the "k-optimised variant".

3. Static hop-by-hop QoS routing

3.1 Framework

This chapter evaluates the performance of SAMCRA in a connectionless environment. The routing decisions are made based on an extended OSPF protocol (see [Zhang ea., 1997], [Apostolopoulos ea., 1999] for a more in-dept discussion on QoS extensions to the OSPF protocol). The extended OSPF protocol differs from the currently used version (OSPFv2, [Moy, 1998]) because it incorporates different classes of service (CoS). These different classes of service each consist of *m* lower bounds on the constraints, i.e. if a boundary exceeds the corresponding requested constraint, that particular class of service can not accommodate the requested quality of service. The number of QoS-parameters (*m*) that is to be used, or the problem of maintaining the set of classes is not considered here. This extended OSPF protocol is only mentioned to situate the environment we try to model in this chapter. Since the extended OSPF protocol works with an exact number of QoS-parameters, the user can not request more constraints on these parameters. However if less than *m* constraints are provided, the remaining constraints are considered to be infinite (i.e. they can always be handled). Figure 26 gives an example of this extended OSPF protocol, for 3 different classes to one destination.

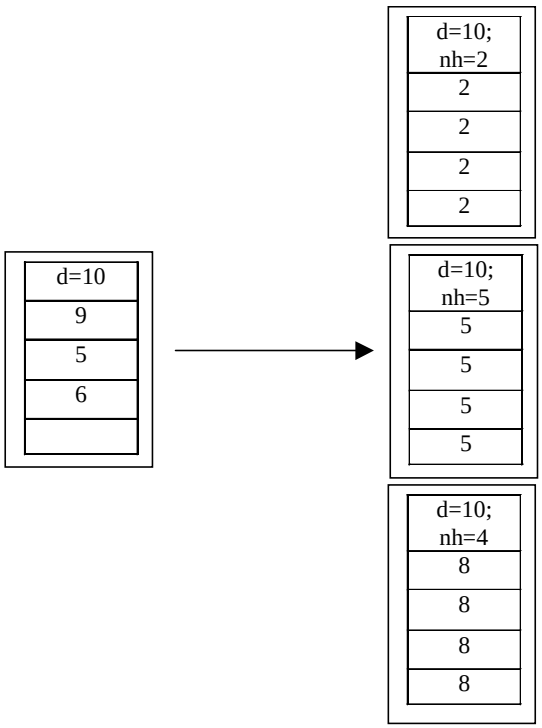


Figure 26. Example of how the extended OSPF protocol works. d=destination, nh=next hop

In the example displayed in figure 26, the user requests constraints on three different QoS-parameters, where the protocol uses four. The last constraint is therefore assumed to be infinite. The protocol now checks which of the classes is able to handle the requested constraints. The criteria used during this scanning procedure, is that all of the requested constraints should be larger or at least equal than the lower bounds of a class. It may occur that more than one class is able to handle the constraints. In this case the one with the largest lower bounds is chosen. If the class with the smallest (strictest) lower bounds would be chosen, all traffic would be sent to this class and its corresponding link would get congested.

3.2 Routing loops

The previous chapter provided an important corollary, namely: *When using a non-linear definition of the path length, the subsections of shortest paths in multiple dimensions are not necessarily shortest paths.* This corollary made it necessary for T(S)AMCRA to keep track of *k*-shortest paths. As long as enough paths are considered, the shortest path from source to destination could always be retrieved. This however is not always the case when we perform routing in a hop-by-hop mode. With “hop-by-hop destination based only” routing each hop calculates the shortest path to the destination and sends its packets to the next hop corresponding to this shortest path. Therefore the overall (end-to-end) path will consist of a sequence of shortest paths. As stated by corollary 1, this may result in a hop-by-hop path that is not equal to the shortest path. A question that might arise, is: Is hop-by-hop routing with a non-linear path length loop-free?

The answer lies in lemma 1 and property 3 [Van Mieghem ea., 1999].

Let P_k denote the shortest path from an intermediate hop k towards the destination computed by SAMCRA.

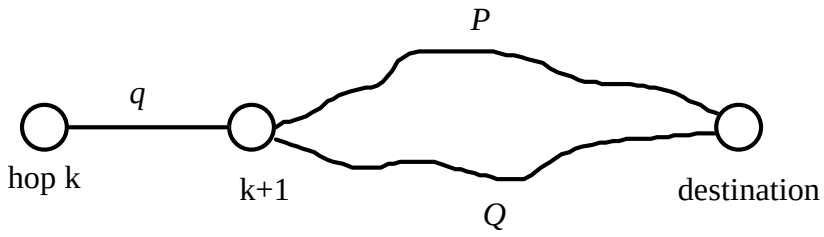


Figure 27. Construction of hop-by-hop paths via SAMCRA

Lemma 1: If P_{k+1} *is a subsection (or sub-path) of* P_k *there holds that* $l(P_k) \geq l(P_{k+1})$ *otherwise* $l(P_k) > l(P_{k+1})$

Proof:

If P_{k+1} is a subsection (or sub-path) of P_k, we have that $P_k = q + P_{k+1}$ where q is a single link vector (the first link vector on the shortest path from hop k to the destination). The first condition then is immediate from (8).

The second condition follows from the corollary that subsections of shortest paths are not necessarily shortest paths. Hence, we may have that, referring to figure 27, the path $P_k = q + Q$ is the shortest path from k to the destination while $P_{k+1} = P$ is the shortest path from hop $k+1$ to that same destination. Thus[3], $l(P + q) > l(Q + q)$ and $l(Q) > l(P)$. On (8) holds $l(Q + q) \geq l(Q)$. Combining all inequalities leads to $l(P + q) > l(Q + q) \geq l(Q) > l(P)$ or $l(P_k) > l(P_{k+1})$. QED.

Lemma 1 implies that the sequence of the lengths of the shortest paths from the different intermediate hops to the (same) destination is non-increasing: $l(P_1) \geq l(P_2) \geq \ldots \geq l(P_k) \geq l(P_{k+1}) \geq \ldots \geq 0$. The end-to-end path from source node s to destination node d computed by hop-by-hop SAMCRA consists of the node list $\{s, v_2, \ldots, v_j, \ldots, d\}$ where v_j is the second node in the node list of path P_{j-1}.

Property 3: *The end-to-end path computed by hop-by-hop SAMCRA is guaranteed to be loop-free*

Proof:

Property 3 is a consequence of lemma 1. Indeed, a loop means that at some hop k, the path has returned to a node n previously visited at hop j with $j < k$. Further, the presence of a loop implies that not all P_k are subsections of P_1 and that, in the

sequence of lengths of shortest paths between hop *j* and hop *k*, there must be at least one strict inequality sign (lemma 1) yielding $l(P_j) > l(P_k)$ with $j < k$. Let *P* denote the shortest path from node *n* to the destination. At hop *j*, we first arrive at node *n* and we have $P_j = P$. Since the loop returns during hop *k* at node *n* again, we obtain $P_k = P$ implying that $P_j = P_k$ and thus $l(P_j) = l(P_k)$ for $j < k$. This condition contradicts the previous one based on lemma 1. QED.

In conclusion, we have demonstrated that "hop-by-hop destination based only" multiple constraint routing with SAMCRA is loop-free. In case of an approximate QoS routing algorithm that can not guarantee to always find the shortest path towards the destination in every hop, the arguments used in the proof of property 3 and lemma 1 do not exclude the occurrence of loops. Hence, so far, we have shown that exactness is a sufficient condition. It remains to show that exactness is also necessary. Therefore, it suffices to illustrate with an example that loops can indeed occur if the *x-th* shortest path is chosen with $x > 1$.

Consider the topology in figure 28.

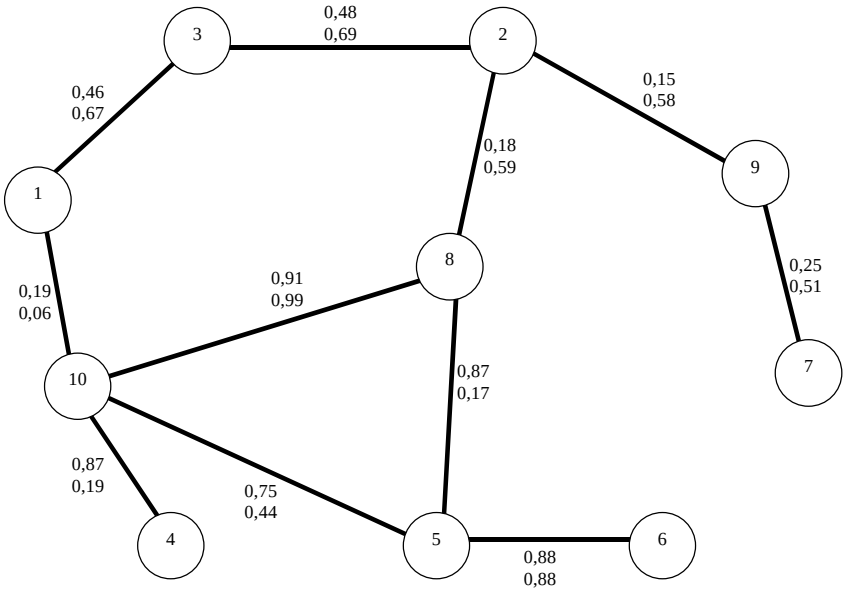


Figure 28. Topology on which hop-by-hop TAMCRA (with k=2) creates a loop.

[3] It is assumed that not both conditions $\{P_k = q + Q$ and $q + P$ are two shortest paths (equal in length)$\}$ together with $\{P_{k+1} = P$ and Q are two shortest paths (equal in length)$\}$ coincide such that the strict inequality sign in lemma 1 is justified.

Assume source node 1 wants to send its data to destination node 7, with both of the constraints equal to 9. For the path calculation TAMCRA with a maximum queue-size (*k*) of 2 is used.

Starting at node 1 TAMCRA (with *k*=2) finds the following path: 1-10-5-8-2-9-7
In this case the next hop is 10.
The data is now forwarded to node 10, were again a new path towards destination 7 is computed. TAMCRA now retrieves 10-1-3-2-9-7 as its shortest path towards 7.
The next hop in this case is node 1. Node 1 was the original source node, and therefore TAMCRA has created a loop 1-10-1-10-.... The shortest path in this case was not found, because the queue-size was too small. This resulted in a full queue at node 2, with the consequence that the shortest path 10-5-8-2-9-7 was cancelled. This shows that if the value of *k* is too small, hop-by-hop TAMCRA can create a loop. Hop-by-hop SAMCRA automatically uses the appropriate value of *k* and therefore never creates a loop.

3.3 Evaluation of the hop-by-hop path

We are now going to investigate the "badness" of QoS routing relying on a hop-by-hop approach only based on the destination. In particular, only the shortest path (according to SAMCRA's non-linear length definition (6)) is computed in each hop towards the destination (ignoring the previous history where it came from). The resulting end-to-end path computed "hop-by-hop destination based only" via SAMCRA is compared with the shortest end-to-end (source based) path satisfying all QoS constraints.

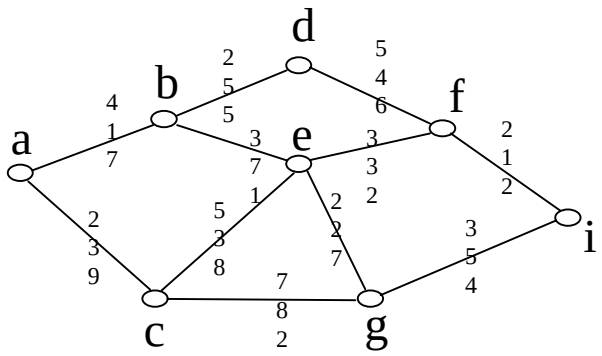


Figure 29. Example topology.

The example topology (figure 29) will be used to demonstrate how SAMCRA performs in a distributed, “hop-by-hop destination based only” mode, further referred to as *hop-by-hop SAMCRA*.

Each link is characterised by three additive QoS metrics (or vector components). Suppose that node *i* wants to send packets to node *a* according to a certain CoS. The CoS is characterised by a constraint for each of the graph metrics. In this example, the constraints are chosen to be 14, 11 and 22 respectively. Using SAMCRA, node *i* calculates the shortest path to node *a* as shown in 30.

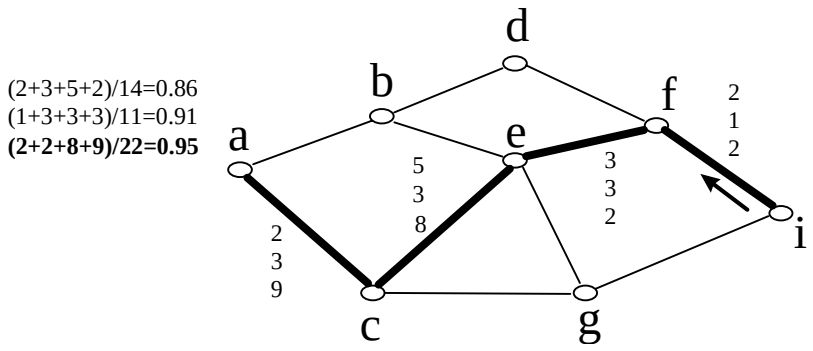


Figure 30. Shortest path from node *i* to node *a*.

The shortest path from node *i* to node *a* runs over nodes *f*, *e* and *c* and has a path length of 0.95. Thus, all constraints are satisfied. Node *i* will store in its routing table that the next hop for destination *a* and the class of service specified by constraints (14, 11, 22) is node *f*. Node *f* will construct its routing table in a similar manner: it uses SAMCRA to calculate the shortest path to destination *a* subject to the constraint vector (14, 11, 22). The result is drawn in figure 31.

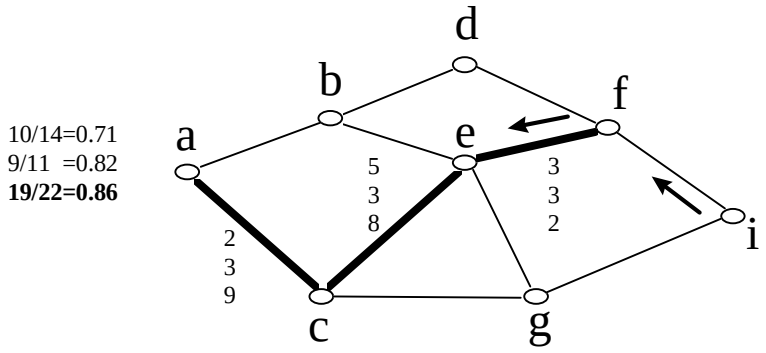


Figure 31. The constrained shortest path from node *f* to node *a*

The shortest path from *f* to *a* has a path length of 0.86 and coincides with the previous shortest path from node *i* to *a*. Node *f* stores node *e* as the next hop towards destination *a* for this class of service in its routing table. Every packet with this class

of service and destination *a* will be forwarded to node *e*, independent of the source node that has sent the packet. So far, no deficiencies have occurred for the packets originated at node *i* because the shortest path from node *i* to node *a* coincides with the shortest path from node *f* to node *a*.

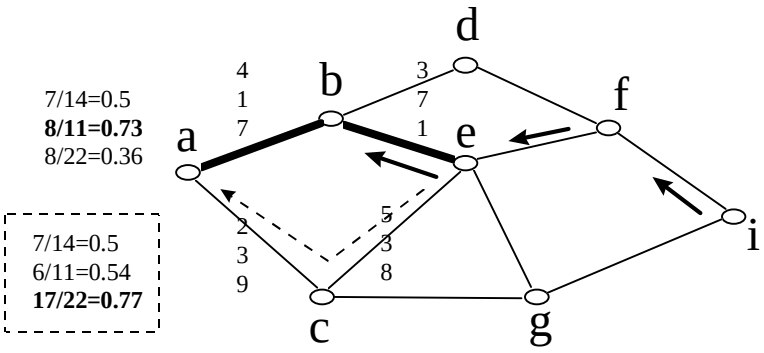


Figure 32. Constrained shortest path form node *e* to node *a*

However, this is no longer the case for node *e*! The shortest path from node *e* to node *a* for the class of service characterised by the constraint vector (14, 11, 22) is shown in figure 32.

The shortest path from node *e* to node *a* runs over node *b* and not over node *c*. The path over node *c* is shown with dashed lines and has a path length of 0.77 which exceeds the path length of the path *e-b-a* which is only 0.73. Thus node *e* will store in its routing table that all packets for destination *a* and the above specified CoS need to be forwarded to node *b*. Packets from node *i* will thus no longer follow the shortest path between node *i* and node *a*.

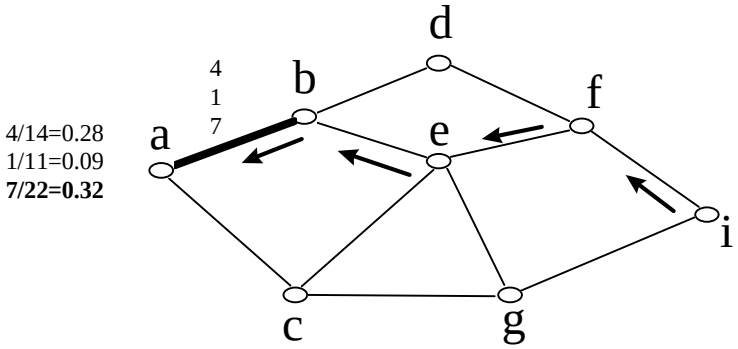


Figure 33. Shortest path from node *b* to node *a*.

Finally, at node *b* packets are forwarded according to the shortest path from node *b* to node *a*, which is the direct link between these two nodes. This is shown in figure 33.

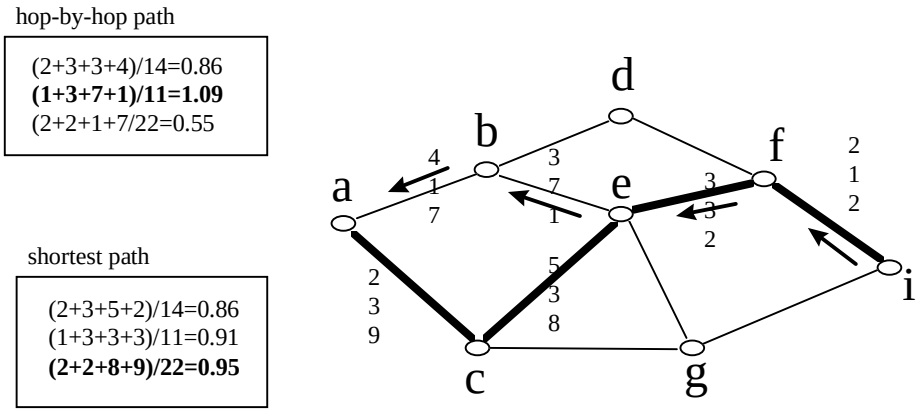


Figure 34. Comparison between the shortest path (in bold) and the hop-by-hop path (arrows).

Figure 34 shows both the end-to-end hop-by-hop path and the shortest path from node *i* to node *a* for the CoS specified by the constraint vector (14, 11, 22). The shortest path satisfies all constraints but the hop-by-hop path has an unacceptable path length of 1.09 > 1. In particular, the box in figure 34 indicates that the hop-by-hop path satisfies the constraints for the first and the third metric but violates the constraints for the second metric. Thus, although there is a path from node *i* to node *a* satisfying all constraints, in a distributed hop-by-hop mode, this example illustrates that the packets will follow a path which violates one of these constraints.

In this example, it can be verified that the hop-by-hop path turns out to be the third shortest path from node *i* to node *a* for the CoS specified by constraints (14, 11, 22). The second shortest path is *P(i, f, d, b, a)*. The path length of the second shortest path satisfying all constraints is exactly 1.00, which means that at least one of the constraints is precisely met.

The reason that the hop-by-hop path can return a non-shortest path that exceeds one or more constraints is because the length of the path traversed so far is not taken into account. Thus, each time a hop is made, the constraints get more relaxed. The solution for this problem requires an active networking protocol. Chapter 4 will further elaborate on this topic.

3.4 Results for static hop-by-hop QoS routing

To evaluate the performance of hop-by-hop QoS routing, numerous simulations were run. Each of these simulations calculated per random labelled weighted graph one path from source (node 1) to destination (node *N*, where *N* equals

the number of nodes), for a total of 10^6 graphs. There are numerous random methods available to model networks. All are variations on the same basic method: a set of vertices is distributed in a plane, and an edge is added between each pair of vertices with some probability *p*. The simulations were done, using the pure random method, where *p* is a fixed number. *p* is further referred to as the linkdensity, since it reflects the number of links in the network divided by the total number of possible links:

$p \sim \frac{2E}{N(N-1)}$, where *E* is the number of links and *N* the number of nodes. (12)

The weight of a link is characterised by a vector consisting of *m* link metrics. The *m* link metrics are assigned a uniformly distributed random number $\in [0,1]$. While the pure random method does not explicitly attempt to reflect the structure of real networks, it is attractive for its simplicity and therefore commonly used to study networking problems. Since the main goal of this thesis is to address the algorithmic aspect of hop-by-hop QoS routing, we did not want to restrict ourselves to "realistic"[4] networks. The 10^6 pure random topologies will most likely also include "realistic" topologies and will therefore give us a more thorough understanding of how the algorithm performs.

To simulate the hop-by-hop behaviour of SAMCRA in random networks a program was written in C. The main modules used for this program are gathered in appendix C. In this program a hop-by-hop path is computed and compared against a list of *x*-shortest paths, to see which hop-by-hop path is found. The list of *x*-shortest paths is an ordered list of shortest paths that is retrieved by not stopping when the end-node is extracted (see appendix C for more information). The *x*-shortest path with $x=1$ is the true shortest path from source to destination. The second shortest path is denoted by $x=2$, the third by $x=3$, etc. Once the nature of the hop-by-hop path is retrieved, it is compared against the shortest path ($x=1$). The results gathered with this approach are displayed below. First the results are given for a network of 100 nodes, with a linkdensity of $p = 0.04$ and 2-element link vectors. The average degree or the average number of links corresponding to this type of random networks resembles that of current internetworks. They have an average degree of 4. The degree of a node: $d(v_i)$, where $i = 1,\ldots,N$ denotes the number of adjacent (= linked) nodes. Since

[4] There is no rule to distinct one network as being realistic and another as being unrealistic. The problem of modelling an internetwork has been (and still is) studied in dept, e.g. [Calvert ea., 1997].

each link connects two nodes, the sum of the degrees over all nodes equals twice the amount of links:

$$\sum_{i=1}^{N} d(v_i) = 2E \qquad (13)$$

The average degree of a random network can now be formulated as:

$$d_{av} = \frac{2E}{N} \qquad (14)$$

Since E (the number of links) is not explicitly known for all random networks, the expected average degree can be found with:

$$E[d_{av}] = p(N\text{-}1) \qquad (15)$$

The expected average degree, together with the number of nodes, are the two main parameters[5] that affect the performance of the algorithm.

Once the results for the 100-node network are given and analysed, we proceed with the evaluation of the performance of the algorithm under certain conditions/parameters. These results are categorised into three parts: The first part uses different network-sizes to see its effect on the algorithm. The second part proceeds with changing the linkdensities and the final part is used to analyse the influence of different numbers of constraints/metrics.

3.4.1 On statistics

All simulations were done using 10^6 random graphs for each type of network. Gauss formulated two equations (16) and (17) that allow us to calculate E[x] and σ_x^2 for the end-result x, which is constructed out of the observations $a, b, c, \ldots$ that are "cursed" with coincidental errors:

$x = f(a, b, c, \ldots)$

If we know E[a], E[b], E[c], … and σ_a^2, σ_b^2, σ_c^2, … then E[x] and σ_x^2 can be calculated as follows:

$$E[x] = f(E[a], E[b], E[c], \ldots) \qquad (16)$$

$$\sigma_x^2 = \left(\frac{\partial f}{\partial a}\right)_{(E[a],\ldots)} \cdot \sigma_a^2 + \left(\frac{\partial f}{\partial b}\right)^2_{(E[a],\ldots)} \cdot \sigma_b^2 + \ldots \qquad (17)$$

These formulas hold for each type of probability density functions.

[5] A third parameter could be the variance of d_{av} (VAR[d_{av}]), which is a good indicator for finding realistic networks. In realistic networks, each node has approximately the same degree, which results in a low value for VAR[d_{av}].

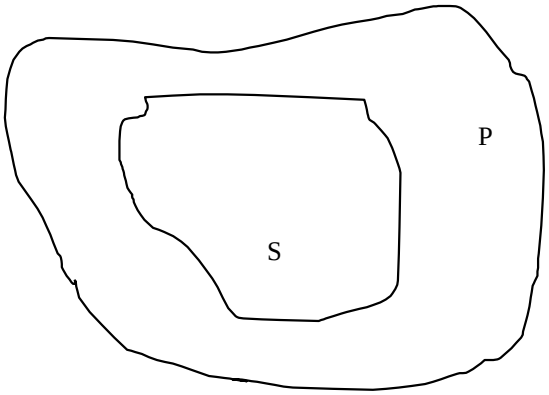


Figure 35. Subset S and entire set P of all possible observations.

Figure 35 shows the set of all possible observations. Considering we can not examine this entire set, we are restricted to a subset *S*, consisting of *n* observations a_i, i=1,…,n, thus:

$$\mathrm{E}[S] = \frac{1}{n}\sum_{i=1}^{n} a_i$$

$$\mathrm{E}[P] = \lim_{n->\infty} \frac{1}{n}\sum_{i=1}^{n} a_i$$

$$\sigma_P^2 = \lim_{n->\infty} \frac{1}{n-1}\sum_{i=1}^{n} (a_i - E[P])^2$$

Using equation (16) and (17), we gain the following results:

$\mathrm{E}[S] = \mathrm{E}[P]$

$$\sigma_S^2 = \frac{1}{n^2} n\sigma_P^2 = \frac{1}{n}\sigma_P^2$$

or

$$\sigma_S = \frac{\sigma_P}{\sqrt{n}}$$

This last expression indicates that the accuracy of the expected value increases with the root of the number of experiments/observations. Since we use 10^6 random graphs, we have $\sigma_S = 0.001\sigma_P$. This means that probabilities up to roughly 10^{-4} may be considered significant.

3.4.2 Evaluation of hop-by-hop SAMCRA in a 100-node network

The network configuration is put below the plots and should be interpreted as follows. *N* is the number of nodes, *p* is the linkdensity and *m* is the number of constraints (with the actual constraints between brackets).

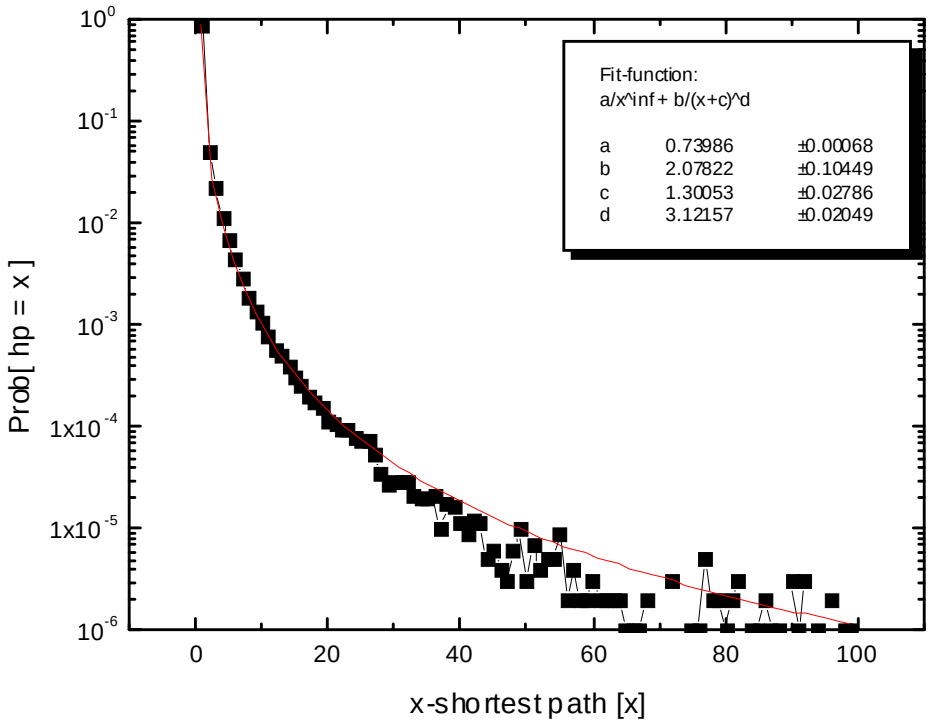


Figure 36. Probability that the hop-by-hop path (hp) equals the x-shortest path. N=100, p=0.04, m=2 (100,100).

Figure 36 displays the probability that the hop-by-hop path (*hp*) equals the *x*-shortest path. This probability distribution indicates the "badness" of the SAMCRA algorithm in a connectionless environment. The hop-by-hop path is considered to be optimal when the true shortest path is found (x=1). According to this probability distribution, the shortest path is found in 89.4 % of the cases. The badness of the hop-by-hop algorithm is therefore reflected by the remaining 10.6 % of the cases, where the algorithm finds a less optimal path (x>1). The probability distribution function shows a polynomial behaviour:

$$\text{Prob}[\ hp = x\ ] \approx \frac{a}{x^{\infty}} + \frac{b}{(x+c)^{d}} \qquad (18)$$

where a, b, c, d are constants which depend on the structure of the network, i.e. the number of nodes, the number of links and how they are distributed. The constant a only affects the cases where the shortest path (x=1) is found. In those cases the pdf

shows a disproportionate increase as opposed to the “nice” polynomial distribution $\frac{b}{(x+c)^d}$.

Since finding a *x*-shortest path with *x*>1 is indeed not optimal, the question arises how much worse the solution actually is. This can be indicated by comparing the length of the hop-by-hop path with the shortest path, as done in figure 37. Figure 37 displays the probability that the length of the hop-by-hop path (*lhp*) is *x* % longer than the length of the shortest path (*lsp*).

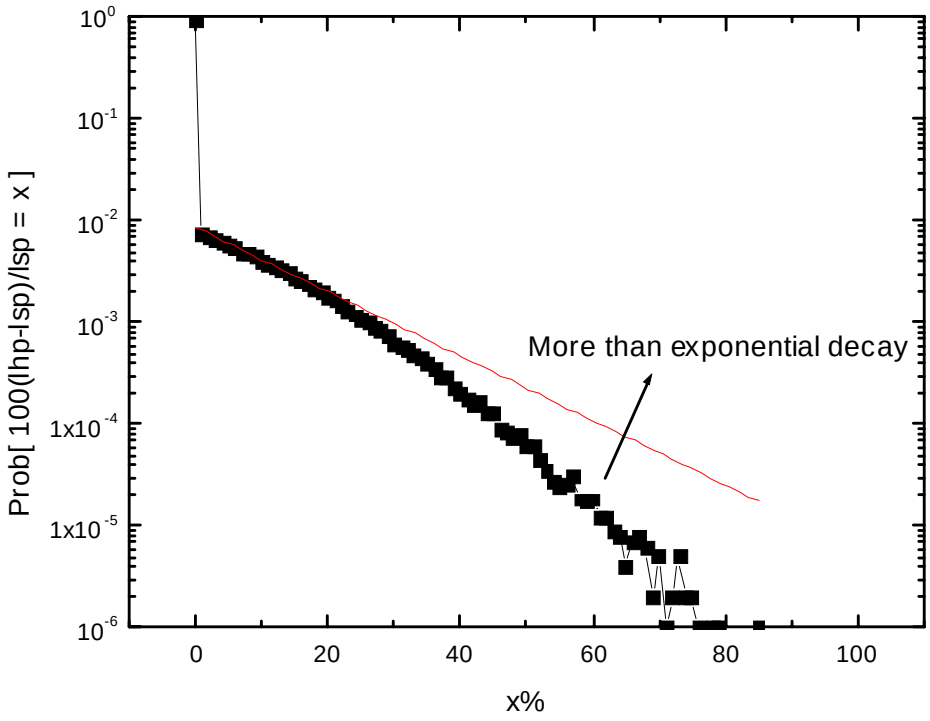


Figure 37. Probability that the hop-by-hop path is x% longer than the shortest path. N=100, p=0.04, m=2 (100,100).

The difference in length between the hop-by-hop path and the shortest path is strongly related to the *x*-shortest path that was found (= hop-by-hop path). The larger *x* in the *x*-shortest path, the larger becomes the difference in length. Since the probability of finding the *x*-shortest path strongly decreases for increasing *x*, a similar behaviour is observed in the difference in length: The probability of finding a hop-by-hop path that is *x* % longer than the shortest path strongly decreases for increasing *x*. This decrease occurs faster than the polynomial decrease observed in the previous plot. The

decrease is even more than exponential, which illustrates that retrieving a non-optimal path might not be as damaging as one might think, because the probability that the length of this path is near the length of the shortest path and within the constraints, is very high.

Property 4: If the hop-by-hop path only makes one wrong decision (compared to the true shortest path), its length never exceeds 100% of the length of the (exact) shortest path.

Proof:

For this proof, we again use the properties of the length of a vector as stated for corollary 1.

Consider the graph in figure 38, where the shortest path from *s* to *d* consists of paths P_a and P_b, with length $l(P_a + P_b)$. The shortest path from *h* to *d* is P_c. Hence, ignoring the past history, the hop-by-hop path makes a wrong decision at node *h* where it follows path P_c resulting in a hop-by-hop path consisting of P_a and P_c.

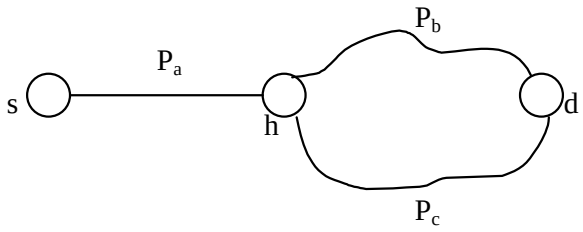


Figure 38. Two paths, exact (P_a+P_b) and hop-by-hop (P_a+P_c)

Since the hop-by-hop path does not equal the shortest path, we know that $l(P_a+P_b) < l(P_a+P_c)$ (we consider $P_c \neq P_b$). Further we know that $l(P_c) < l(P_b)$, otherwise the hop-by-hop path would make the correct decision and follow path P_b resulting in a difference of 0%. Using the properties of the length of a vector, we can arrange the lengths in increasing order as follows:

$l(P_c) < l(P_b) \leq l(P_a+P_b) < l(P_a+P_c) \leq l(P_a) + l(P_c) < l(P_a) + l(P_b) \leq l(P_a) + l(P_a+P_b)$

The relative difference in length therefore has an upper bound:

$$\frac{l(P_a+P_c)-l(P_a+P_b)}{l(P_a+P_b)} < \frac{l(P_a)+l(P_a+P_b)-l(P_a+P_b)}{l(P_a+P_b)} = \frac{l(P_a)}{l(P_a+P_b)} \leq 1 (= 100\%)$$

QED.

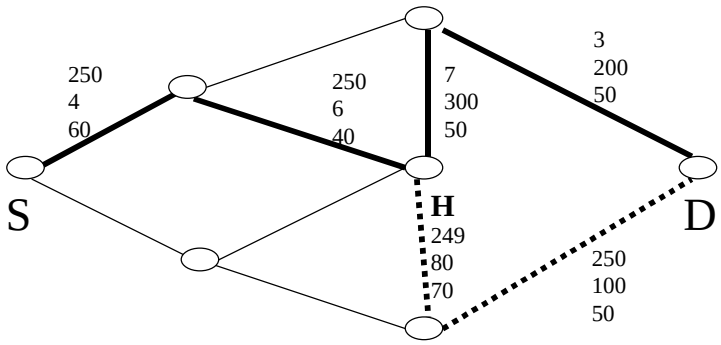


Figure 39. The shortest path is drawn in bold, whereas the hop-by-hop path is in dotted line. The constraint vector is (1000,1000,1000). Referring to the notation of the previous figure, we have that $l(P_c)=0.499$, $l(P_b)=0.5$, $l(P_a+P_c)=0.510$ and $l(P_a+P_c)=0.999$ resulting in a relative difference of 96%.

Moreover this upper bound is sharp as shown by figure 39. It is acknowledged that this upper bound is fairly large for a more realistic distribution of the link metrics, it however shows that a wrong decision at the end of the path (near the destination) can have more severe consequences as opposed to a wrong decision near the source (the first hop is always correct!), because $l(P_a)$ tends towards $l(P_a+P_b)$ when we hop towards the destination.

Property 4 gave an upper bound for the case where only one wrong decision is made. An extension towards multiple wrong decisions is easily made by viewing multiple wrong decisions as making one wrong decision multiple times for different source nodes (the node where the last wrong decision was made, becomes the new source node). The overall upper bound would then equal the sum of the independent upper bounds:

$$UB(s \to d) = \sum_{i=1}^{wd} UB(v_i \to d) \qquad (19)$$

where $UB(s->d)$ denotes the upper bound on the difference in length with the hop-by-hop path and the shortest path from s to d, P is the hop-by-hop path, wd is the number of times a wrong decision is made ($wd \geq 1$), and v_i is the node where the wrong decision is made.

This confirms that making more wrong decisions will result in longer paths, which is an intuitively obvious property.

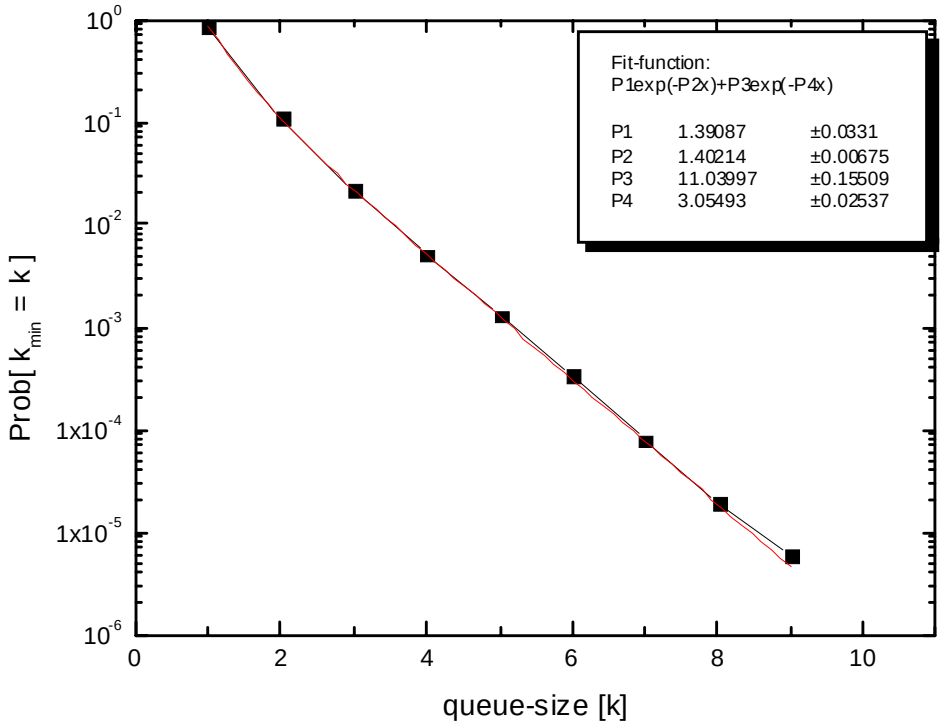


Figure 40. Probability that the minimum queue-size needed to retrieve the shortest hop-by-hop path is k. N=100, p=0.04, m=2 (100,100).

Figure 40 displays the probability that the minimum queue-size (k_{min}) needed to retrieve the shortest hop-by-hop path is k. In other words, if TAMCRA uses a queue-size equal to this minimum k_{min}, then the hop-by-hop path consists of a sequence of shortest sub-paths towards the destination. If a queue-size larger than this k_{min} is used, the same path will be found, but if a smaller queue-size is chosen, the hop-by-hop path will include non-shortest sub-paths and therefore could create loops (see 3.2). Since the use of an exact algorithm to compute the hop-by-hop path implies that the queue-size is always sufficient, the main purpose of this plot is to analyse the NP-completeness of routing in multiple dimensions.

In 86.3% of the cases we only need a queue-size of 1, which implies a Dijkstra-like complexity. With a queue-size of 1 we have a worst-case complexity of O(Nlog$N$ + mE) versus Dijkstra's O(Nlog$N$ + E). For the cases where the queue-size is larger than 1, the parameter k appears in the complexity, which increases this complexity (see (10)). The constraints in the simulation-program were denoted as doubles (real numbers), which makes the upper bound on the queue-size k_{max} (9) extremely large.

Figure 40 however shows that k_{min} is very small compared to k_{max}. The question now arises, which k_{min} refers to NP-completeness? There is no hard boundary to give here, but the plot does indicate that the function decreases exponentially (The first exponent in the fit-function is dominated by the second for $k > 1$. A similar construction was seen in figure 36.). Thus, as mentioned in 2.4.3, *the NP-completeness of multiple constraints routing is not seen in practice, not even when real numbers are used!*

Figure 41 gives the probability distribution of the difference in queue-size between a connection-oriented (shortest) path (*ksp*) and our connectionless (hop-by-hop) path (*khp*).

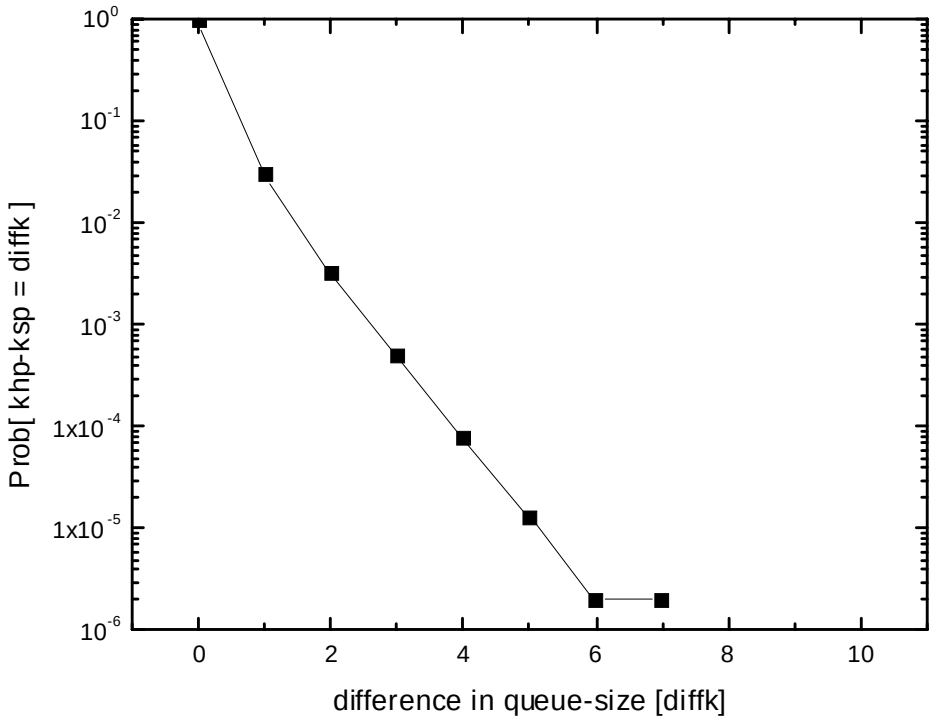


Figure 41. Difference in minimum queue-size between the hop-by-hop path and the shortest path. N=100, p=0.04, m=2 (100,100).

The minimum queue-size for the hop-by-hop path is based on the minimum queue-size needed for the shortest (sub)-paths:

$$k_{\min} = \max_{i:s->d}(k_{\min}(i)),$$

where $k_{min}(i)$ is the minimum queue-size needed to retrieve the shortest sub-path from node *i* towards destination *d*. As a consequence $k_{min} \geq k_{min}(s)$ and the difference ($k_{min} - k_{min}(s)$) can not become negative.

We can see that the largest part (99.6%) of the hop-by-hop paths does not differ in k_{min} from the shortest paths. It can thus be concluded that the probability that the hop-by-hop mode does not add extra complexity is very high. During this project, the idea surfaced to exclude the previous hops from the shortest path calculation to decrease the complexity and eliminate the possibility of creating loops when using TAMCRA with an insufficient queue-size. This approach would require that the packets keep track of the previously visited nodes, which is a considerable overhead and based on the plot above this is not necessary because hardly any complexity is added by the hop-by-hop path.

The final two plots for this type of networks (figures 42 and 43) show the probability distribution for the number of hops taken by the hop-by-hop path and the difference in hops compared with the shortest path. The two plots hardly say anything about the algorithm, but more reflect the type of graphs used.

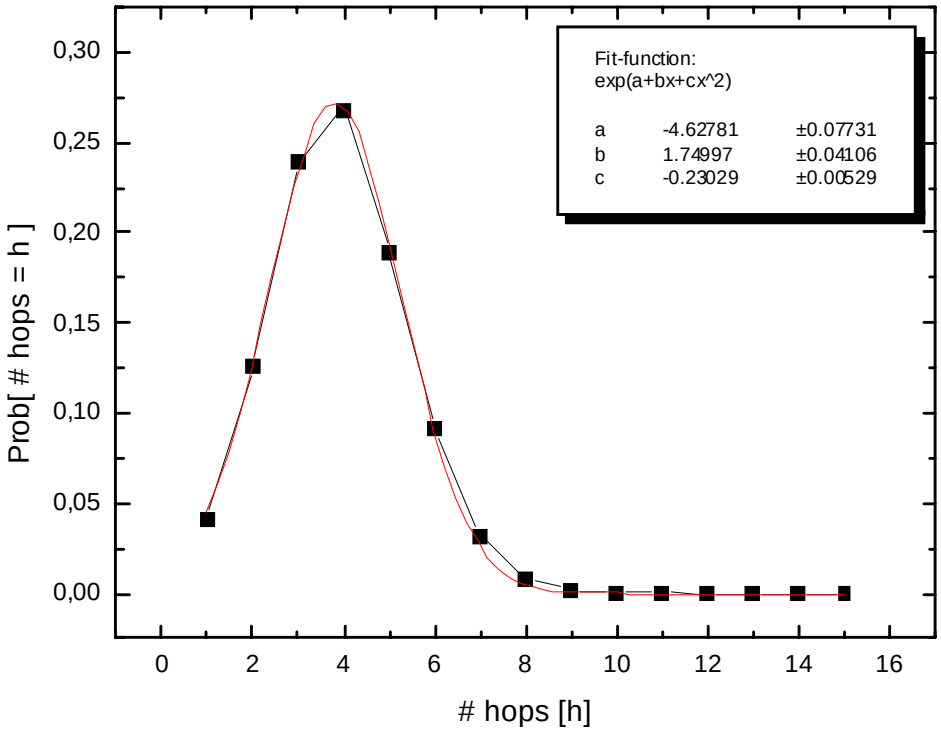


Figure 42. Probability distribution for the number of hops in the hop-by-hop path. N=100, p=0.04, m=2 (100,100).

Equations 20-22 state how we calculated the number of hops (h), its variance and the ratio of the two.

$$E[h] = \sum_{n=1}^{N} n \Pr ob[h = n] \quad (20)$$

$$\mathrm{VAR}[h] = \mathrm{E}[(h-\mathrm{E}[h])^2] = \mathrm{E}[h^2] - \mathrm{E}[h]^2 \quad (21)$$

$$\alpha = \frac{E[h]}{VAR[h]} \quad (22)$$

We found the following values:

E[h] = 3.90292

VAR[h] = 2.1529

α = 1,8129

The class of random graphs used has resulted in needing an average of roughly 4 hops to reach the destination. This gives us an indication of the maximum amount of wrong decisions that could have been made, i.e. a hop-by-hop path consisting of n hops can at most make n-1 wrong decisions.

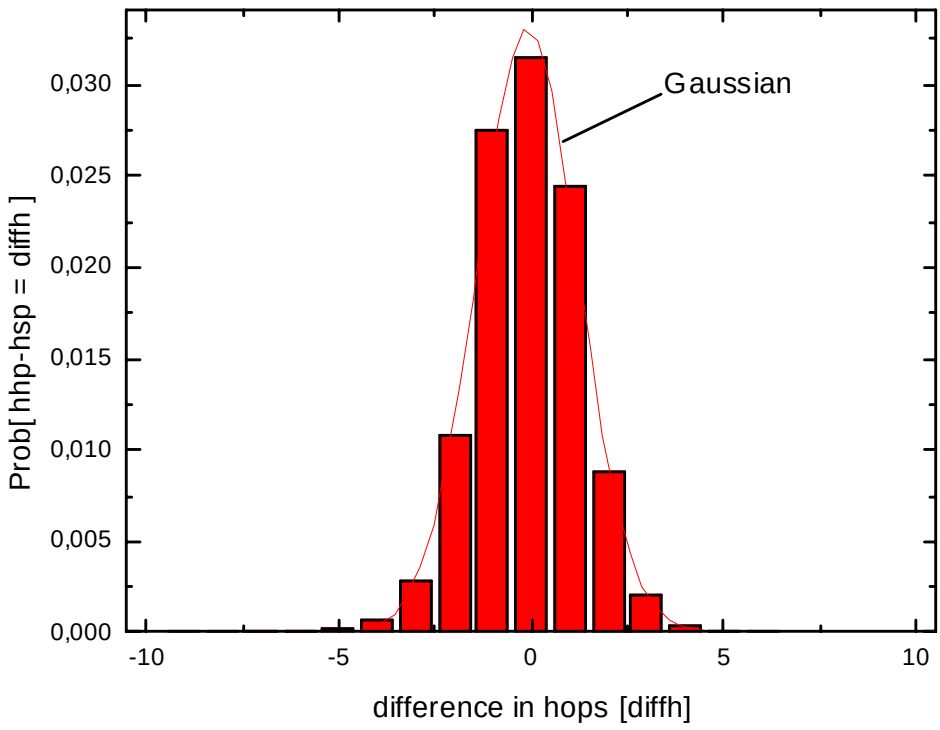


Figure 43. Probability distribution for the difference in number of hops between the hop-by-hop path and the shortest path. N=100, p=0.04, m=2 (100,100).

Figure 43 depicts the difference in hop-count between the hop-by-hop path (*hhp*) and the exact shortest path (*hsp*). Both positive (the hop-by-hop path is longer in hop-count and length) and negative (the hop-by-hop path is shorter in hop-count but longer in length) differences seem equally possible (a Gaussian with approximately zero mean) which is a manifestation of the class of random graphs and the uniformly distributed link weights.

3.4.3 Behaviour of hop-by-hop SAMCRA under different network-parameters

Figures 44-46 reflect the results for different network sizes (number of nodes is 10, 20 and 50). The linkdensity of the networks is p=0.2 and the number constraints (m) equals 2. All the constraints were chosen equal to their network size (N), which means that there always exists a path that can handle the constraints. The results for the 100-node networks with p=0.04 were also added.

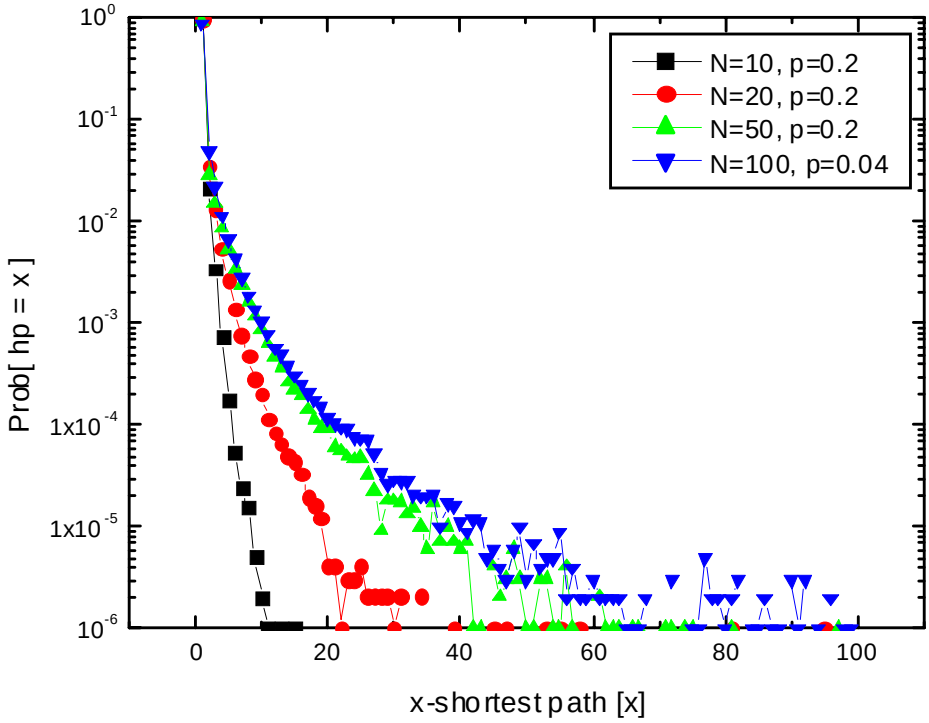


Figure 44. Probability that the hop-by-hop path is the x-shortest path.

Figure 44 illustrates that there is a difference among the different network-sizes: an increase in the network-size increases the probability of finding the *x*-

shortest path with x relatively large ($x \neq 1$). The reason is that an increase in network size increases the number of links (recall that $E \sim pN(N-1)/2$), which on its turn results in more x-shortest paths between the source and destination. The more paths there exist between source and destination, the more decisions the algorithm has to take, because each node (on average) has more paths towards the destination. The increase in network-size also increases the average number of hops on the path and therefore also the number of times the algorithm is executed. The probability of making a wrong decision, i.e. not finding the shortest path, thus increases with the size of the network.

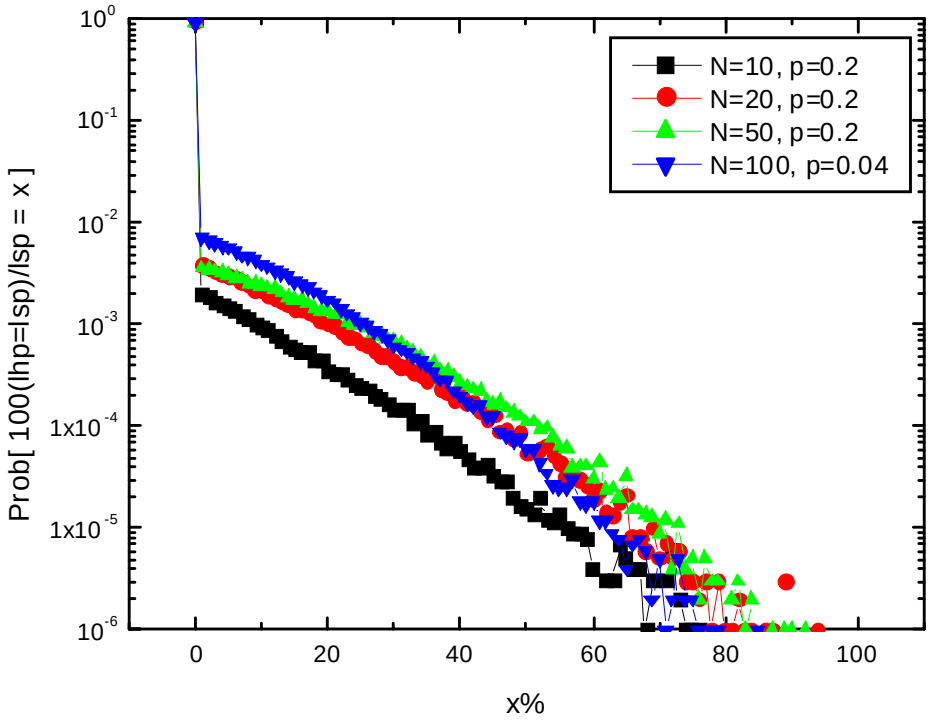


Figure 45. Probability that the hop-by-hop path is x% longer than the shortest path.

As explained for figure 44, an increase in network size decreases the probability of finding the shortest path. As explained for figure 37, this therefore increases the probability of finding a path that is x% longer than the shortest path.

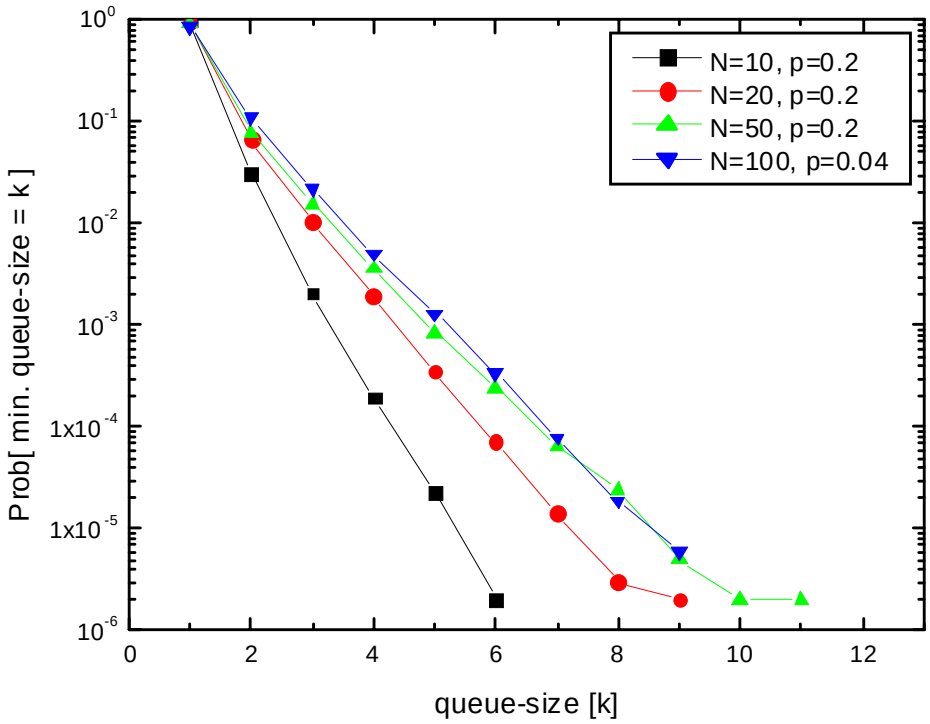


Figure 46. Probability that the minimum queue-size needed to retrieve the shortest path is k.

An increase in network-size means an increase in the number of *x*-shortest paths. Since there are more paths available, SAMCRA has to keep track of more non-dominated paths. This increases the probability of needing a larger queue-size.

Figures 47-51 show the results when using different linkdensities (*p*).

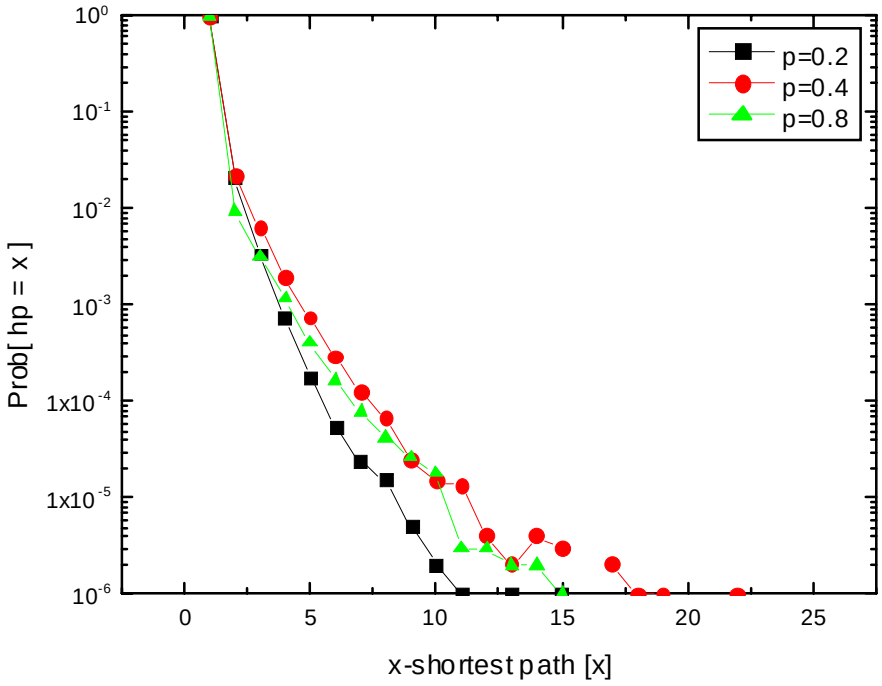


Figure 47. Probability that the hop-by-hop path equals the x-shortest path. N=10, m=2 (10,10).

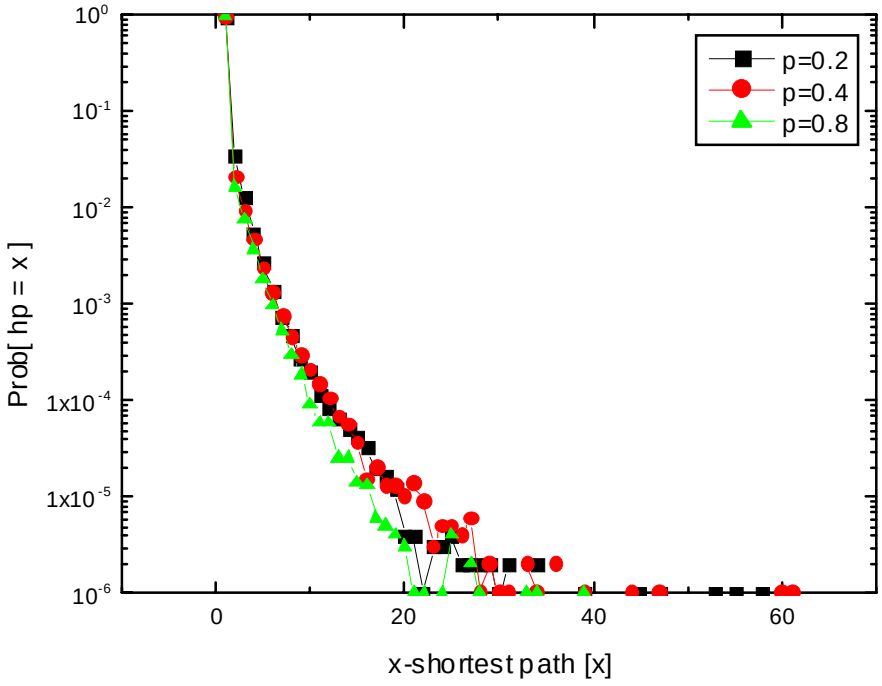


Figure 48. Probability that the hop-by-hop path equals the x-shortest path. N=20, m=2 (20,20).

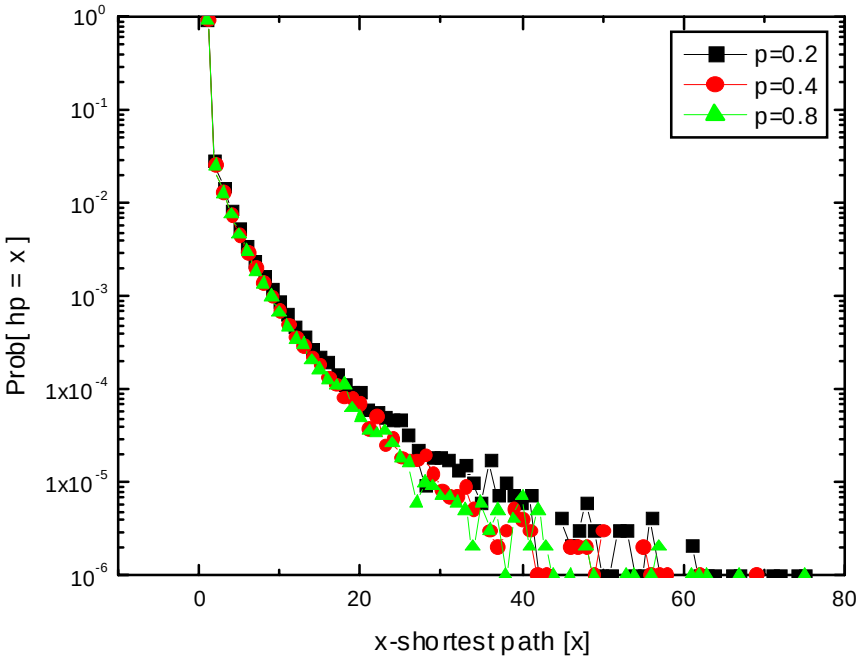


Figure 49. Probability that the hop-by-hop path equals the x-shortest path. N=50, m=2 (50,50).

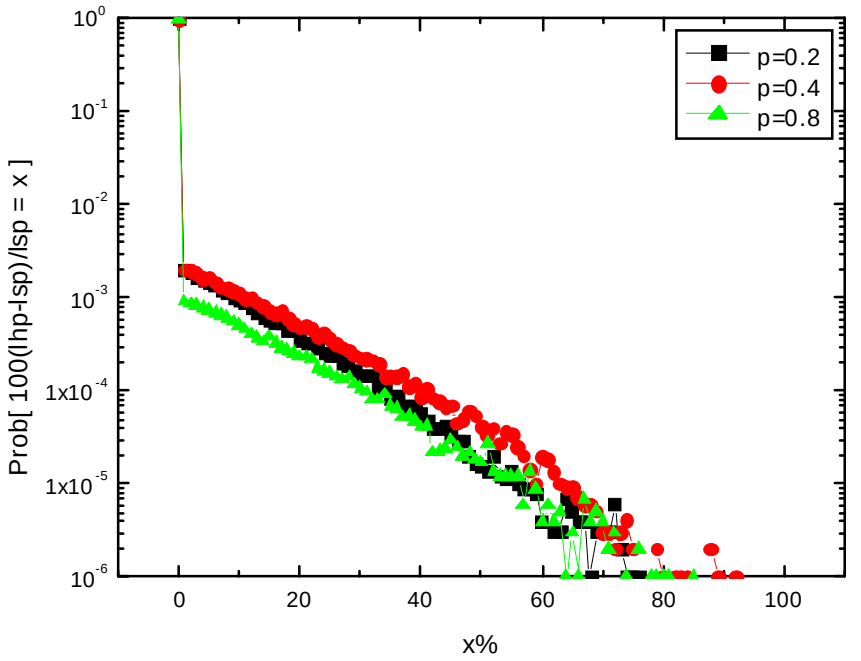


Figure 50. The probability that hop-by-hop path is x% longer than the shortest path. N=10, m=2 (10,10).

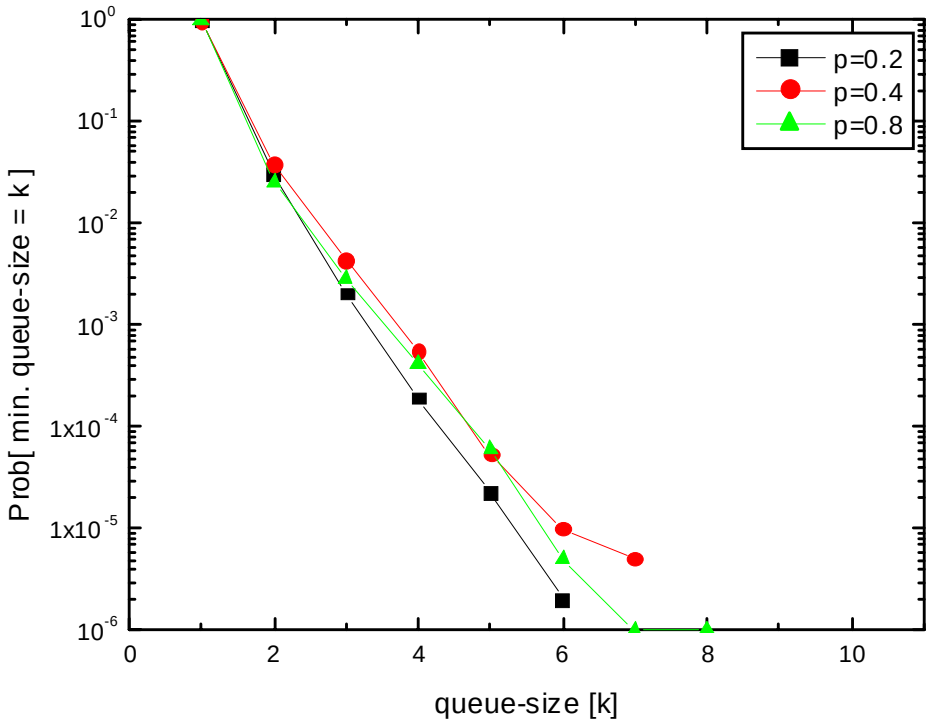


Figure 51. Probability that the minimum queue-size needed to retrieve the shortest path is k. N=10, m=2 (10,10).

The dependency on the linkdensity *p* becomes vanishingly small as *N* grows [Van Mieghem ea., 2000]. In other words, as long as $p>0$ is constant, the precise value of *p* is not important in the limit $N\to\infty$. The same phenomenon is observed here, when we look at the plots for *N*=10, 20 and 50, where the lines belonging to different *p*-values grow closer together for increasing *N*.

Figures 44-46 suggest that there does exist a small difference when using different linkdensities (*p*=0.2 for *N*=50 and *p*=0.04 for *N*=100). In this case the *N* of 100 is too small as opposed to the *p* of 0.04. The average degree d_{av} per node seems to give a more concrete threshold-value. If $2 \leq d_{av} \leq 12$ we see a small difference among the different d_{av}, which can be attributed to the following:

If we have a small d_{av}, there is only a small set of *x*-shortest paths available. The probability that hop-by-hop SAMCRA retrieves the *x*-shortest path, with *x* large, will therefore be very low. If we have a high d_{av}, there is a high probability that the destination node is linked to the source node or only a few hops away. This decreases

the probability that hop-by-hop SAMCRA finds the *x*-shortest path, with *x* large. The worst-case results are expected to lie around an average degree of 8-10.

Figures 52-54 show the results for different number of constraints/metrics. The number of nodes was chosen 20 and the linkdensity 0.2. All constraints were 20.

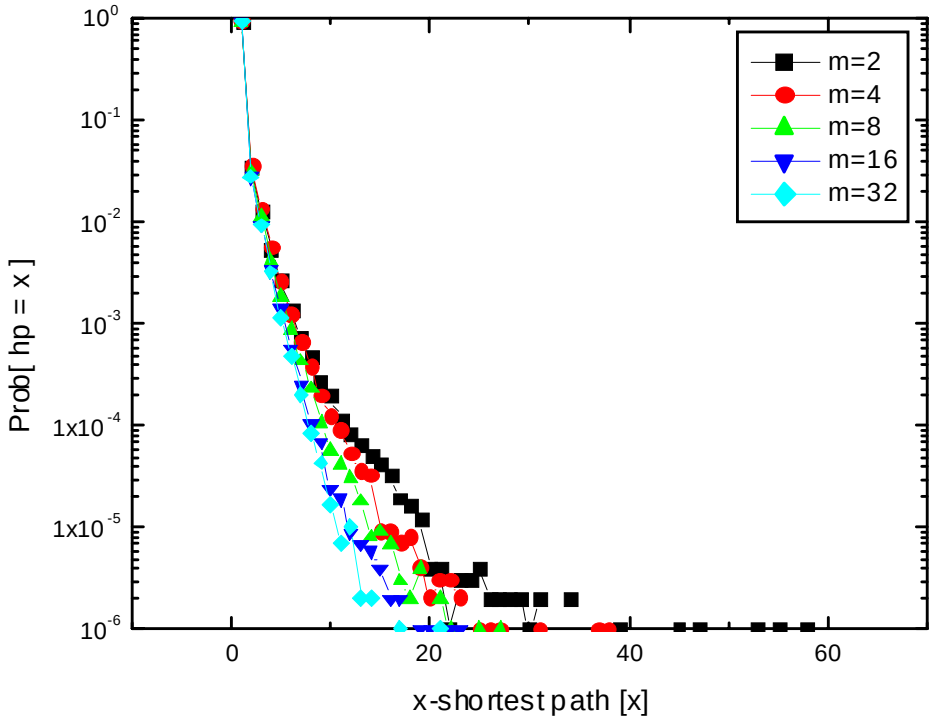


Figure 52. Probability that the hop-by-hop path equals the x-shortest path

When using only one metric, the shortest path is found with probability 1. This is due to the fact that with one metric there is no more non-linear definition of the path length. Corollary 1 does not apply for this one-dimensional case, thus sub-paths of shortest paths are now indeed also shortest paths.

Looking at the dimensions (m>1), a small difference among the different numbers of metrics can be observed. This small difference implies that an increase in the number of metrics increases the probability of finding the shorter paths. This is partly a factor of the uniform distribution of the link metrics. Due to this uniform distribution, the probability that an element of the link vector is smaller than *x*, becomes *x*:

$$\text{Prob}[\ w_m \le x\ ] = x, \qquad \text{for } 0 \le x \le 1 \tag{23}$$

where w_m is the m-th element of the link vector.

This results in a probability for the length of the link, equal to:

$$\text{Prob}[\ l(w) \le x\ ] = x^m, \qquad \text{for } 0 \le x \le 1 \tag{24}$$

where m is the number of metrics.

(24) indicates that the length of a link grows with the number of metrics:

$$\lim_{m\to\infty} l(w) = 1 \tag{25}$$

Considering that the length of independent links, and correspondingly sub-paths, increases, the probability of finding other shortest sub-paths decreases. Since there are less sub-paths for the hop-by-hop path to follow, the probability of finding a x-shortest path (x>1) decreases for increasing m, which is confirmed by figure 52.

Figure 53 shows a similar behaviour. Here the difference in length between the hop-by-hop path and the shortest path is depicted. We can see an improvement in the results for increasing m. We already saw that the probability of finding the x-shortest path (x>1) decreases with increasing m and we could therefore expect a decrease in the number of hop-by-hop paths that is y% longer (y>0) than the shortest path. However the difference in length is also affected by the length of the links. Since the length of the shortest path grows with the number of links, the length of the hop-by-hop path (which also is relatively high) will not differ much from the length of the shortest path. Hence the probability of finding a hop-by-hop path that is y% longer (y>0) than the shortest path decreases with increasing m.

It is important to note here that although an increase in m seems attractive, this result strongly depends on the values of the constraints. In our simulations these constraints were large enough to guarantee the existence of at least one path, but in reality these constraints will be much smaller and therefore the blocking-rate will increase if we increase m (because the lengths of the paths increase and surpass the constraints).

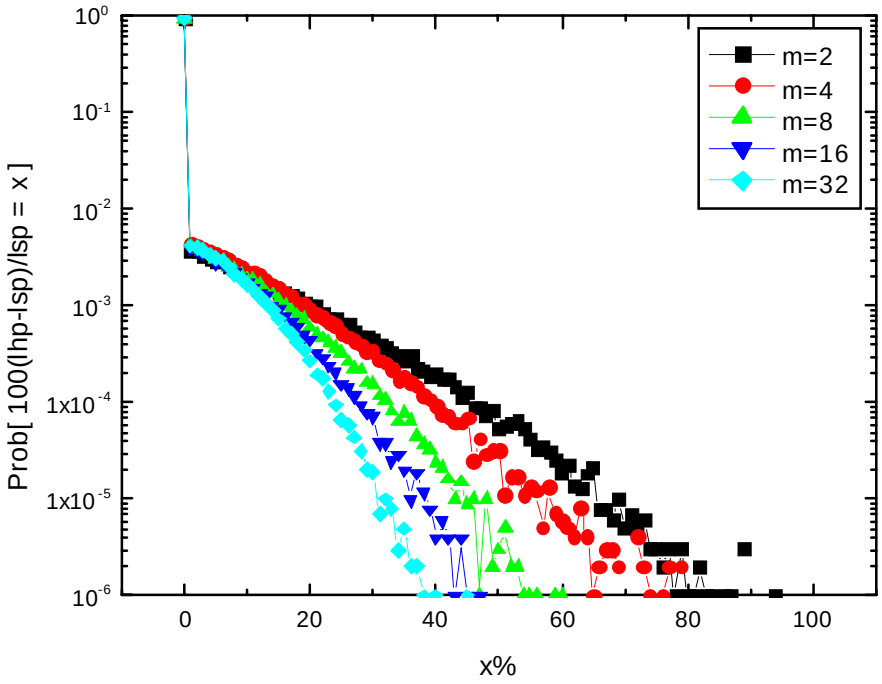


Figure 53. Probability that the hop-by-hop path is x% longer than the shortest path

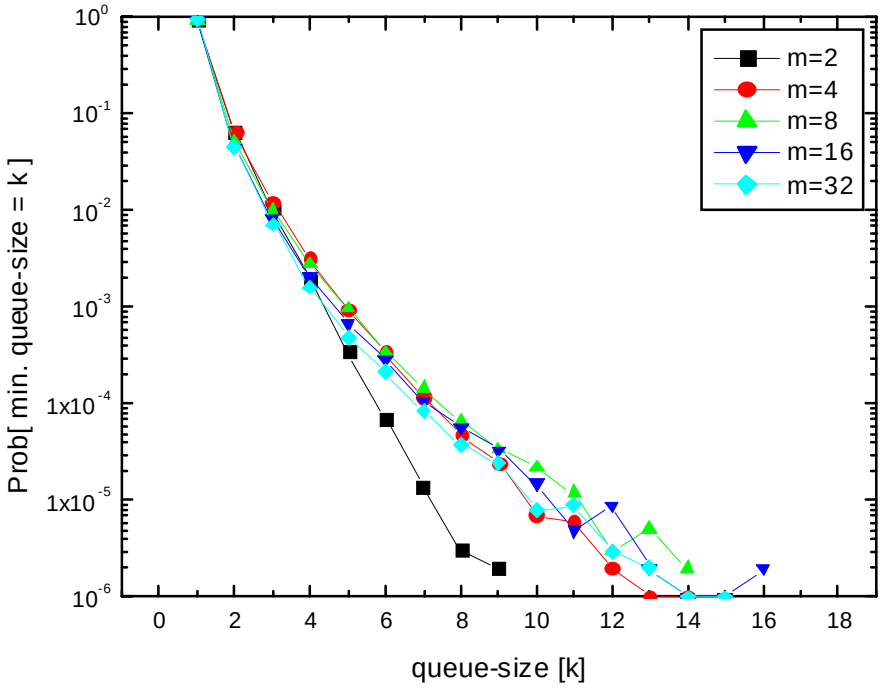


Figure 54. Probability that the minimum queue-size needed to retrieve the shortest path is k.

When the number of constraints increases, the number of non-dominated paths also increases. This means that each node has to store more paths, which is reflected in the calculation time and the needed queue-size. However, this increase only occurs up to a certain level ($m \approx 8$), after which the queue-sizes saturate or even decrease. This is due to the increase in the length of the links/paths (24), which on its turn results in more (non-dominated) paths that exceed the constraints and hence are not stored in the queue.

4. Dynamic hop-by-hop QoS routing

4.1 Framework

The previous chapter evaluated the performance of SAMCRA if used in currently existing connectionless network architectures. These networks can be typified by their inability to alter packets at routers/switches (nodes) on the path. This resulted in the necessity to use static headers/packets, which eliminated the possibility to incorporate the m costs of the path traversed so far. This on its turn resulted in the small probability that a non-shortest path could be retrieved that violates the constraints (see 3.3). To eliminate the probability of finding a hop-by-hop path that violates the constraints, we need a network that is able to alter a packet, with the costs of the path it travelled so far. Routers that are currently operable are not equipped with such a functionality, however the research community has already started to investigate the topic of multi-functional networks under the name "active networking" (see [Psounis, 1999] and [Calvert ea., 1999] for an introduction to active networks). This new approach to network architecture owes its name to the active computation on packets in the network. These networks are active in two ways: routers and switches within the network can perform computations on user data flowing through them and users can "program" the network, by supplying their own programs (see appendix B) to perform these computations [Tennenhouse ea., 1996]. Thus the internal nodes perform more and more similar to the end-nodes.

We will give two "active" scenarios in which we can situate SAMCRA:

1. The first scenario is very similar to the OSPF example given in 3.1 and will further be referred to as "active constraints". Again each node has a set of different classes of service (CoS). The only difference is that now each node first decreases the constraints of a packet with the costs of the link it came in on, before it looks which class can accommodate these constraints. This type of active routing excludes the probability of retrieving paths that violate the constraints without adding much extra complexity ($O(Nm)$ per packet, because there are not more than N hops, where N is the number of nodes in the network, and at each internal node there are m subtractions). Using classes of service however includes some inefficiency in the use of resources, because the

constraints of a packet will most likely differ from the constraints in a CoS. Scenario 2 does not have this problem.

2. In the second scenario (also called “active SAMCRA”), the packet has an extra field where it stores the *m* costs of the path traversed so far. This path vector is updated at each node it arrives at, with the link vector of the link it used to arrive at that node. The link vector is then added to the path vector and based on this new path vector SAMCRA computes the next hop towards the destination. Thus instead of assigning the start-node (= the current node) the vector 0, it is assigned the path vector and the shortest route towards the destination is computed. Obviously this way of routing is much more complex than the other ways, because it now needs $O(kNlog(kN) + k^2mE + Nm)$ per packet

Which scenario to choose depends on the amount of computational power and resources available. Since the research community is still divided on the use of active networking based on its complexity (and security), scenario one will most likely prevail over scenario two, despite the better performance of scenario two (see the following two paragraphs).

4.2 Evaluation of the hop-by-hop path with active constraints

This paragraph will present the results for routing with decreasing constraints in a 100-node network with linkdensity $p = 0.04$ and $m = 2$ constraints. Since the results look very similar to those depicted earlier (in 3.4.2) for the static case, we first subtracted the static results from the active results in order to gain a better view of this difference. (In both cases the same 10^6 random networks were used).

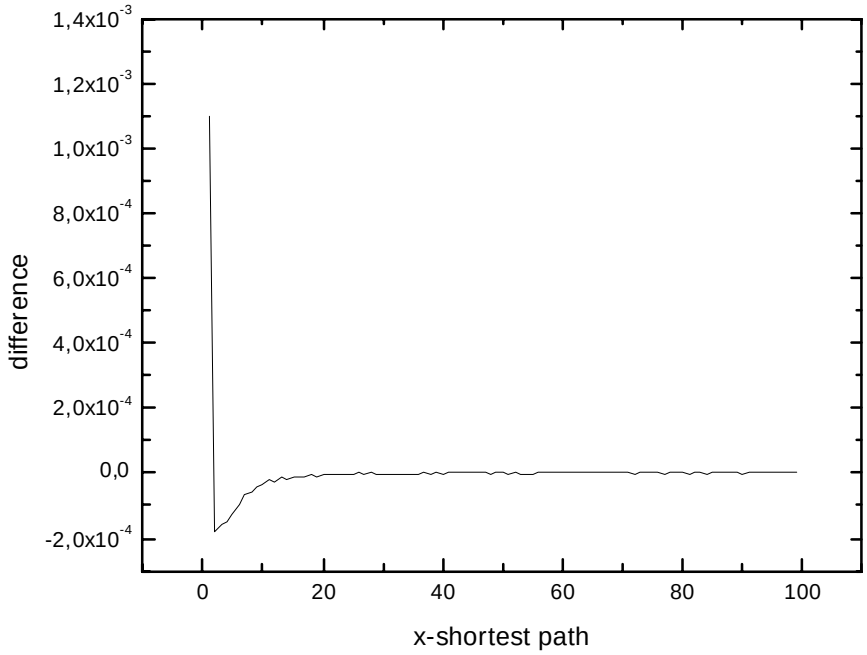


Figure 55. Difference, between active constraints and static SAMCRA, in the probability that the hop-by-hop path equals the x-shortest path. N=100, p=0.04, m=2 (100,100).

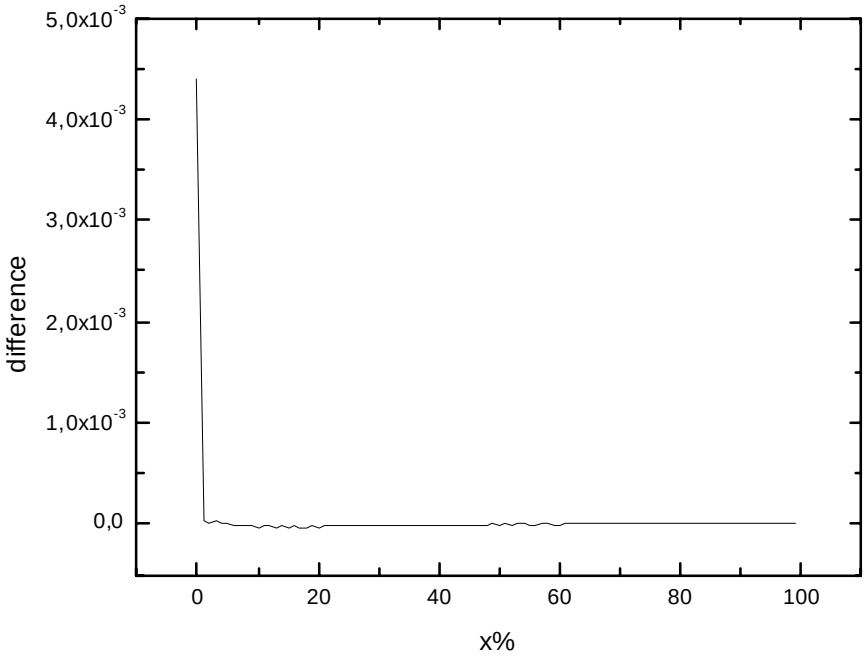


Figure 56. Difference, between active constraints and static SAMCRA, in the probability that the hop-by-hop path is x% longer than the shortest path. N=100, p=0.04, m=2 (100,100).

Figures 55 and 56 indicate a small improvement in the probability of finding the shortest path. This improvement is a factor of the decreasing constraints, since after each hop the constraints decrease, resulting in fewer paths towards the destination that are able to handle those decreased constraints. It therefore becomes less likely that a wrong decision (hop) is made and this is reflected in the probability distribution, where we can observe that relatively more shortest paths are found. The probability of finding a *x*-shortest path also depends on the values of the constraints. For the simulations, the constraints were chosen equal to *N* (the number of nodes), to guarantee a path to exist. However, these values are relatively large resulting in a lot of suitable paths towards the destination. In practice the constraint-values will be much stricter, what will increase the probability of finding the shortest path (for active constraints as well as for static SAMCRA, if such a path exists). The difference between active constraints and static SAMCRA will slightly increase if stricter constraint-values are used.

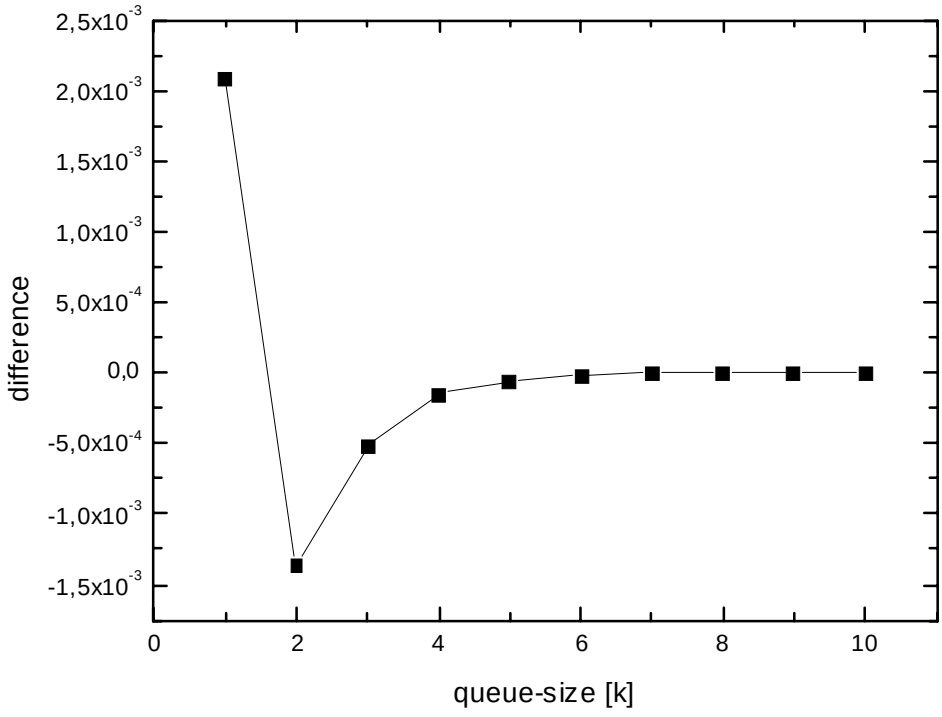


Figure 57. Difference, between active constraints and static SAMCRA, in Prob[k_{min} = k].

Besides improving the results, “active constraints” also needed on average a smaller queue-size. Again this can be attributed to the decreasing constraints, since lower

constraint-values means that a constraint is easier exceeded and those paths that exceed the constraints are not stored and therefore less queue-space is needed.

4.3 Evaluation of the hop-by-hop path with active SAMCRA

Property 5: Active SAMCRA always finds the shortest path if that path meets the constraints.

The proof of property 5 is quite trivial once one has gained a good understanding of the way SAMCRA works. The proof makes use of SAMCRA's meta-code as listed in 2.4.1.

Proof:

Important for this proof is to recall that SAMCRA keeps track of the path vectors. A path vector represents for each metric the sum of the link metrics corresponding to the links traversed by the path, i.e. $l_i(P) = \sum_{u,v \in P} l_i(u,v)$ for i = 1, …, m. The path vector is updated in line 13 of the meta-code and in line 14 its length is computed according to (6).

Previously (in 2.4.2) it was proved that SAMCRA is indeed exact. Hence to prove that active SAMCRA is exact, we need to show that it finds the same path from source s to destination d as the path computed by SAMCRA (in a connection-oriented (CO) manner).

Now consider the topology in figure 58.

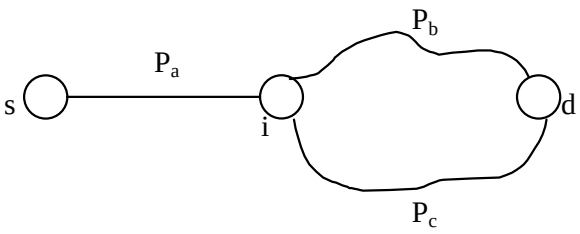


Figure 58. Example topology.

The shortest path from s to d computed by SAMCRA-CO is composed out of P_a+P_b and has a length $l(P_a+P_b)$. Suppose now that active SAMCRA has followed P_a up to node i, where it again has to compute the best path towards

d. For this computation it uses the path vector belonging to P_a. Thus instead of starting with `path(i[1]) = NULL and length(i[1]) = 0` it starts with `path(i[1]) = P`$_a$ `and length(i[1]) = l(P`$_a$`)`. At some time during the computation SAMCRA-CO extracted a minimum length $l(P_a)$ belonging to the sub-path P_a to node *i*. By starting with P_a we have confined active SAMCRA to start at the same point. Indeed if SAMCRA-CO found the shortest path by extracting P_a at node *i*, then active SAMCRA must also find this shortest path when starting with P_a at node *i*, i.e. active SAMCRA searches for the path $P_?$ for which $l(P_a + P_?)$ is shortest. It is obvious that $P_? = P_b$ because $l(P_a + P_b)$ was found to be shortest. If $P_? = P_a$ this would mean that $l(P_a + P_c) < l(P_a + P_b)$ contradicting the proof of 2.4.2 that SAMCRA is exact!

This leaves us to prove that our assumption that active SAMCRA followed the correct path up to node *i* was legitimate. Since at source node *s* the computation of active SAMCRA is identical to that of SAMCRA-CO, we know that the first hop is always correct. Thus P_a is assigned the link values of the link just traversed. As argued above, active SAMCRA also makes a correct decision at an intermediate hop, therefore the second hop is also correct. This sequence of correct decisions is continued until the destination is reached. QED.

It is proved that in static networks active SAMCRA yields the same (exact) results as connection-oriented SAMCRA. However in practice networks show a dynamic behaviour, i.e. the link metrics change over time or a link failure may occur. The global view of the network provided to SAMCRA may therefore become inaccurate. This inaccuracy has its largest impact on connection-oriented protocols, since they only compute a shortest path at the source and then set up a connection based on this shortest path. If the network conditions change during a connection, this connection may become inefficient (e.g. it may violate the constraints) and may even fail. Since active SAMCRA works in a hop-by-hop mode, at each hop the best path towards the destination is computed based on up to date network information. Active SAMCRA is therefore able to adapt to changing networks and hence is considered to perform even better than a connection-oriented approach.

5. Conclusions

Guaranteed quality of service (QoS) is a missing element in today's Internet, where all packets are treated as being equally important. Guaranteed quality of service is needed to enable multimedia applications to function correctly and will therefore become an essential part of future routing protocols. The main problem surrounding QoS routing is to compute a path that obeys a set of QoS-constraints. Moreover, this (algorithmic) problem has been proved to be NP-complete.

The main motivation for this thesis was to assess the feasibility of "hop-by-hop destination based only" guaranteed QoS routing that ignores the source and previous path history (conform current IP routing), with the emphasis on the algorithmic problem of routing with multiple constraints. Thus, guaranteed QoS refers to guaranteeing to find a path within the constraints, if such a path exists. An overview of QoS-algorithms was given, which indicated that SAMCRA has the best properties to be used in hop-by-hop QoS routing. SAMCRA guarantees to always find the shortest path from source to destination. This exactness is essential when routing is performed in a hop-by-hop manner otherwise loops may form. In practice, i.e. when the number of constraints (m) is fixed and the constraints themselves (L_i for $i = 1, \ldots, m$) are bounded integers, the running-time of SAMCRA is polynomial in its input length. Moreover even if m and L_i are not bounded, simulations show that the NP-completeness will be hardly ever seen.

To evaluate the feasibility of "hop-by-hop destination based only" guaranteed QoS routing, some extensive simulations were run. For 10^6 random networks hop-by-hop SAMCRA computed a path from source to destination. Unfortunately the simulations revealed that hop-by-hop SAMCRA cannot guarantee quality of service. Since SAMCRA is exact this means that there is no other algorithm that can guarantee quality of service under the same circumstances (an IP environment). However the simulations reveal that, although exactness is missing, the performance may still be considered good: in 90 % of the cases the (exact) shortest path is found and the added length of the remaining paths (10 %) is relatively small and can be upper bounded.

Finally it was shown that if the current Internet mode of "hop-by-hop destination based only" routing (ignoring the past path history) is relaxed, hop-by-hop

routing with active packets can be exact provided that the length vector of the path traversed so far and the constraint vector are stored in each packet.

Future work:

Future work may be done on two subjects.

1. Find the subset of network topologies that lead to NP-completeness. This includes a thorough evaluation of the (worst-case) number of non-dominated paths that exist in a network.
2. Apply SAMCRA in practice, i.e. determine which QoS-parameters are needed and find a suitable protocol in which SAMCRA may be implemented. Also evaluate the performance of the entire protocol, for instance what happens is the traffic on the links is high/low? Finally an extensions towards multicast QoS may be made.

Appendix A. On NP-completeness

Let us say that a function *f(n)* is *O(g(n))* whenever there exists a constant *c* such that $|f(n)| \leq c|g(n)|$ for all values of $n \geq 0$. A polynomial time algorithm is defined to be one whose time complexity function is *O(p(n))* for some polynomial *p(n)*, where *n* is used to denote the input length. Expressions such as log(*n*) are also considered polynomial, because they can be bounded by a "regular" polynomial.

A problem is referred to as being intractable if it is so hard that no polynomial time algorithm can possibly solve it [Garey ea., 1979]. Whether all NP-complete problems are intractable has not been proved, but it is commonly acknowledged that any problem belonging to the class of NP-complete problems is hard to solve.

Proving a problem to be NP-complete is mostly done by reducing it to a known NP-complete problem. The first known NP-problem was provided by Cook [Cook, 1971]. All other NP-complete problems can therefore be reduced to this (or in theory any other) NP-complete problem. Therefore if one NP-complete problem is proved to be intractable or solvable in polynomial time, this also holds for the entire class of NP-complete problems. In [Garey ea., 1979] six other basic NP-complete problems are listed and proved. Of these six problems, the PARTITION problem is best suited to prove that the MCP-problem is NP-complete.

PARTITION problem:

Instance: Given a finite set *A* and a "size" $s(a) \in \mathbb{Z}^+$ for each $a \in A$.

Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a) \quad ?$$

The PARTITION problem is particularly useful for proving NP-completeness results for problems involving numerical parameters such as lengths, weights, costs, capacities, etc.

A.1 Proving the MCP-problem to be NP-complete

A lot of articles (see references and references therein) assume the MCP-problem to be NP-complete based on [Garey ea., 1979]. Here it is claimed that the "shortest weight-constrained path" problem (MCP with $m = 2$) is NP-complete, but no

proof is given. The claim is based on private communication with N. Megiddo and hence not verifiable. Wang and Crowcroft [Wang ea., 1996] also noticed this and therefore provided a proof that the MCP-problem is indeed NP-complete. This proof is displayed below in a slightly modified form. The modifications were made because some small errors were encountered and for the sake of clarity.

Theorem A1: Given a graph $G(V,E)$ with for each link $e \in E$ m additive metrics $l_i(e)$ for $i = 1, \ldots, m$, a source node s and a destination node d and m positive integers L_i for $i = 1, \ldots, m$, where $m \geq 2$, $l_i(e) \geq 0$, $L_i \geq 0$. The problem of deciding if there is a simple path P from source to destination that satisfies the following constraints $l_i(e) \leq L_i$ for $i = 1, \ldots, m$ (the MCP-problem) is NP-complete.

Proof:

A proof by induction. First we show that the MCP-problem for $m = 2$ is NP-complete. Since PARTITION is a well-known NP-complete problem, we shall show that PARTITION $\propto$ 2CP-problem to prove its NP-completeness.

Given a network with n+1 nodes and 2n links (between each pair of nodes i and i+1, two links) and a set of numbers $a_i \in A$ for $i = 1, \ldots, n$. Let $\sum_{i=1}^{n} a_i = S$, let metric $l_1(i,i+1)$ for the two links from node i to node i+1 be respectively S and S-a_i and let $l_2(i,i+1)$ be respectively 0 and a_i ($0 \leq a_i \leq S$ and $\sum_{i=1}^{n} a_i = S$).

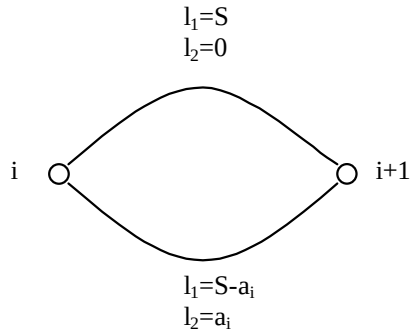


Figure A1. Assignment to two links between node i and i+1

The constraints are chosen as follows:

$L_1 = nS - S/2$ and $L_2 = S/2$

Thus to solve the 2CP-problem we need to find a path from s=1 to d=n+1 for which:

$l_1(P) \le nS - S/2$ (A1)

$l_2(P) \le S/2$ (A2)

Note that for both the upper and lower link between i and i+1 we have

$$l_1(P) + l_2(P) = \sum_{i=1}^{n} (l_1(i, i+1) + l_2(i, i+1)) = nS \quad \text{(A3)}$$

From (A1) we know $l_1(P) \le nS - S/2$, hence according to (A3) $l_2(P) \ge S/2$. However (A2) restricts $l_2(P)$ to $l_2(P) \le S/2$. A solution satisfying the constraints is therefore only found if $l_2(P) = S/2$ and $l_1(P) = nS - S/2$.

The problem has now transformed into a PARTITION problem, i.e. the only way to solve this 2CP-problem is to find a set $A' \subseteq A$ for which

$$\sum_{a_i \in A'} a_i = \frac{S}{2} \qquad (= \sum_{a_i \in A - A'} a_i, because \sum_{a_i \in A} a_i = S)$$

Once the set A' is known, the solution to the 2CP-problem is found by choosing the lower link if $a_i \in A$' and choosing the upper link if $a_i \notin A$'. For the resulting path, we get $l_2(P) = S/2$ and since $l_1(P)+l_2(P) = nS$ we also get $l_1(P) = nS - S/2$. Of course if no set A' could be retrieved, then no feasible path exists.

This solves the 2CP-problem.

We will now show that the MCP-problem with m=n $\propto$ MCP-problem with m=n+1.

m=n+1: $l_i(P) \le L_i$ for i = 1, …, n+1 (A4)

Let L_{n+1} be a large number, say $L_{n+1} = \sum_{e \in E} l_{n+1}(e)$. Thus we have $l_{n+1} \le L_{n+1}$ for any path P. So $l_i(P) \le L_i$ for i = 1, …, n if and only if (A4) holds. QED.

The topology used in the proof (figure A1) might seem unrealistic, since it allows two links between a pair of nodes. These links may however be seen as two different paths from node i to node i+1. The topology in figure A2 illustrates this for two simple paths between node i to node i+1. The example clearly shows the explosive growth that may occur.

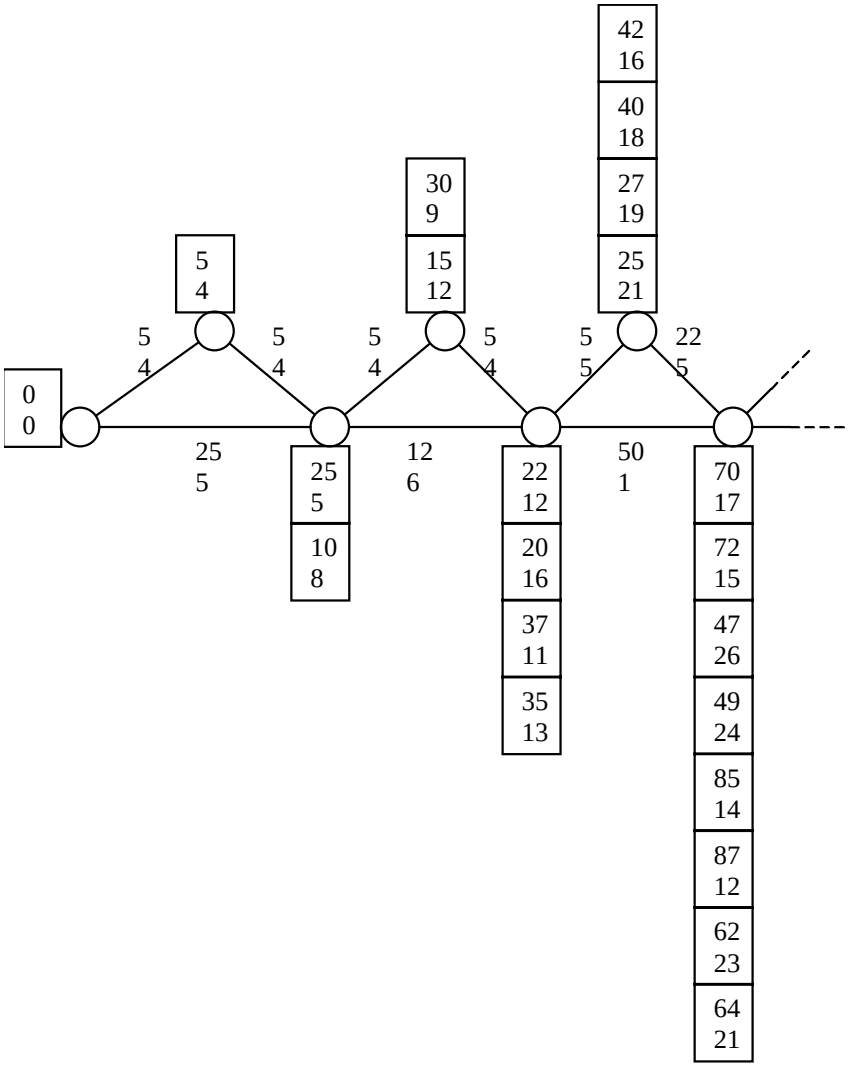


Figure A2. The growth in queue-size when SAMCRA is used to calculate the shortest path.

Figure A2 illustrates that when using a similar topology as the one of figure A1, the number of sub-paths towards some node *i* grows exponential. In this case the number of paths grows with $O(2^{\lfloor (N-1)/2 \rfloor})$, provided that all paths are non-dominated [Henig, 1985]. If there are more paths between node *i* and node *i*+1, the increase in sub-paths will grow even faster.

A.2 Polynomial-time solvable instances of the MCP-problem

Theorem A2: If all metrics/constraints except one take bounded integer values, the MCP-problem for *m* fixed is solvable in polynomial time.

Proof:

To prove theorem A2 to be correct, it suffices to give an algorithm that works in polynomial time once all except one metrics/constraints take bounded integer values. Such an algorithm is described in [Chen ea., 1998a]. It is essentially an extended Bellman-Ford algorithm. It finds a path P such that $l_1(P) \leq L_1$, where $l_1(P), L_1 \in R^+$ and $l_i(P) \leq x_i$ for $i = 2, \ldots, m$, where $l_i(P), x_i \in I$ (= a finite set of positive integers):

```
Initialise(G,s)
for each vertex v ∈ V, each k_i ∈ [0,x_i], 2≤i≤m do
      d[v, k_2, …, k_m] = ∞
      π[v, k_2, …, k_m] = NULL
for each k_i ∈ [0,x_i], 2≤i≤m do
      d[s, k_2, …, k_m] = 0

Relax(u, k_2, …, k_m, v)
for each i ∈ [2,m] do
      k'_i = k_i + l'_i(u,v)
if k'_i ≤ x_i, for each i ∈ [2,m] then
      if d[v, k'_2, …, k'_m] > d[u, k_2, …, k_m]+l_1(u,v) then
            d[v, k'_2, …, k'_m] = d[u, k_2, …, k_m]+l_1(u,v)
            π[v, k'_2, …, k'_m] = u

Main(G,s)
Initialise(G,s)
for i = 1 to |V|-1 do
      for each k_i ∈ [0,x_i], 2≤i≤m do
            for each edge (u,v) do
                  Relax(u, k_2, …, k_m, v)
```

The path is stored in π. The worst-case time complexity of this algorithm is $O(x_1 * \ldots * x_m VE)$, where x_i is a bounded integer and therefore this complexity is polynomial for a fixed m. QED.

Theorem A3: If all metrics are dependent on a common metric and growing in the same direction, the MCP-problem is solvable in polynomial time.

Proof:

If all metrics are related to each other, each metric may be transformed to a common metric. If this transformation-function is composed out of positive exponents (negative exponents like x^{-2} are not allowed), all metrics are growing in the same direction, i.e. if the common metric grows so do all the metrics. Thus if we minimise the common metric automatically all metrics are minimised. Since finding a shortest path based on one common metric is easily done in polynomial time, the "related" MCP-problem is also solvable in polynomial time. QED.

Both theorem A2 and A3 were mentioned in [Chen ea., 1998b], but remained incomplete. For theorem A2 they did not mention the need for a fixed (or bounded) number of metrics and for theorem A3 they mentioned that a relationship suffices for polynomiality to apply, which is incorrect because if one metric x is related to metric y according to $x = 1/y$ then minimising y maximises x (and minimising $y + 1/y$ equals the "linear" heuristic of Jaffe [Jaffe, 1984] which is not exact).

Appendix B. A multi-functional SAMCRA algorithm

SAMCRA with a non-linear definition of the path length as:

$$l(P) = \max\left(\frac{l_i(P)}{L_i}\right), \quad \text{for } i = 1, \ldots, m$$

where *l(P)* obeys the fundamental properties of the length of a vector (see [Golub ea., 1983] and corollary 1), treats all metrics in the same way. By using another non-linear function for the path length it is possible to differentiate among the different metrics, i.e. some metrics will be given a higher priority than others and will therefore be optimised sooner. As long as the paths lie within the constraints, each function for the path length may be used.

A general formulation for the non-linear path length:

$$l(P) = F(P), \quad \max\left(\frac{l_i(P)}{L_i}\right) \le 1 \text{ for } i = 1, \ldots, m$$

$$= \infty, \quad \text{else}$$

F may take on any form, e.g. Jaffe's "linear" function: $F(P) = \sum_{i=1}^{m} d_i l_i$, $0 \le d_i \le 1$

The larger d_i is, the sooner that parameter *i* will be optimised! d_i is therefore our priority-parameter in this function.

One may even incorporate min/max QoS-parameters into this function.

Example:

L_1 = 10 Mb/s (bandwidth is a min parameter -> the link on the path with the smallest bandwidth determines the total available bandwidth)

L_2 = 10 ms (delay is an additive parameter)

L_3 = 10 (cost is an additive parameter)

Path travelled so far: $l_1(P)$ = 15, $l_2(P)$ = 5, $l_3(P)$ = 5

New link: l_1 = 12, l_2 = 1, l_3 = 1 -> new path: $l_1(P)$ = 12, $l_2(P)$ = 6, $l_3(P)$ = 6

$$F(P) = d_1 \frac{L_1}{l_1(P)} + d_2 \frac{l_2(P)}{L_2} + d_3 \frac{l_3(P)}{L_3}, \quad l_1(P) \ge L_1$$

$$= \infty, \quad \text{else}$$

Note that for min/max QoS-parameters a path P_2 is dominated by P_1 when $l_i(P_1) \geq l_i(P_2)$ for each i belonging to the set of min/max metrics.

The idea of having different functions F, is very suited for use in an active network: the user himself can program the network to use a certain F.

The SSR+DCCR algorithm [Liang ea., 1998] already makes use of a function F that was specifically created to target the delay-constrained least-cost path problem.

Appendix C. C-code of the static simulations-program

The modules below were used to run the simulations for the “static” hop-by-hop algorithm. Only the main modules are listed, the other modules can be found elsewhere:

Fibonacci heaps: see [Cormen ea., 1997]

Prim’s algorithm was used to check if the graph is connected. Other algorithms can also be used for this purpose.

The random graph generator was a simple routine. Different types of graphs may be used, just make sure that they are represented by numadj, adj and datadj.

```
/*
samcra-cl.c:
This file is customised to give input to the hop-by-hop shortest path
calculation with the SAMCRA algorithm (CL).
This hop-by-hop shortest path is compared with the shortest path
found with source-based SAMCRA (CO).
The results are written in out1 through out7.

INPUT:
- number of topologies
- number of nodes in a topology
- link density
- number of additive metrics on a link
- the constraints

OUTPUT: - which source-based path is found with hop-by-hop SAMCRA and
          what is the difference in length with the shortest path
        - which queue-sizes (complexity) were needed
        - how many hops does the hop-by-hop path have and what is the
          difference in hops with the source based path

Hans De Neve   24/07/98 (creator of the tamcra version)
F.A. Kuipers 25/01/2000 (converted tamcra-cl2.c to samcra-cl.c)
*/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include "/home/mieghemp/nr/include/nrutil.h"
#include "linked_list.h"
#define MAXPATH 1000000
#define LENGTH 100
#define PERCENT 100

typedef struct el{
        int k;
        int first;
        struct el *next;
        } element;

struct dynamic_array{
        int data;
        int index;
```

```
        struct dynamic_array *next;
        };

main(argc,argv)
int argc;
char *argv[];
{
  clock_t clock(void);
  int           i,N,m,s,l,connected,out2;
  int           k,j,iter,*primlist,hopcount_sp,complexity;
  int           buffer;
  long int      seed,num;
  double        a;
  int           *klist1, *klist2, *klist3, *klist4, *klist5, *klist6, *mink,*hopcount,*hopcountdiff;
  int           **adj, **OUT_total;
  int           *numadj;
  double        ***datadj;
  double        *A,p;
  int           time3, time4;
  double        time5, time6;
  double        treevalue;
  double        *maximumvalue;
  element       ***ordertensor, *nextpointer;
  struct dynamic_array *queue, *complex;
  FILE          *resultaat,*inputfile;

  /*Read input from inputfile */

  inputfile=fopen(argv[1],"r");
  fscanf(inputfile,"%d",&iter);     /*input for the number of topologies*/
  fscanf(inputfile,"%d",&N);        /*input for the number of nodes*/
  fscanf(inputfile,"%lf",&p);       /*input for the linkdensity*/
  fscanf(inputfile, "%d",&m);       /*input for the number of constraints
                                    (=metrics per  link)*/
  A = dvector(1,m);
  for(l=1; l<=m; l++)
    {
     fscanf(inputfile,"%lf",&A[l]);  /*input for the constraint-values*/
    }
  fclose(inputfile);

  /*Initialisation */

  /*buffer represents the maximum number of paths samcra works with.
    So you can have the shortest path, second shortest path up to
    the 100th shortest path*/
  buffer = 100;
  /*numadj (amount of neighbouring nodes), adj (neighbouring nodes)
    and datadj (metrics of links), represent the topology. They will
    be assigned by randomtopolMqos*/
  adj = imatrix(1,N,1,50);
  numadj = ivector(1,N);
  datadj = d3tensor(1,m,1,N,1,50);
  primlist = ivector(1,N);
  OUT_total = imatrix(1,N,1,N);
  seed = -1;
  /*klist1-3 are used to calculate which hop-by-hop path is found*/
  klist1  = ivector(1,buffer);
  klist2  = ivector(1,buffer);
  klist3  = ivector(1,buffer);
```

```
/*klist4-6 are used to calculate the difference in length between
  the shortest path and the hop-by-hop path*/
klist4  = ivector(1,PERCENT);
klist5  = ivector(1,PERCENT);
klist6  = ivector(1,PERCENT);
/*mink is used to show how many times a certain k (queue-size) is
  needed to get the shortest path*/
mink = ivector(1,buffer);
maximumvalue = dvector(1,N);
hopcount = ivector(1,N);
hopcountdiff = ivector(-N,N);
queue=NULL;
complex=NULL;

out2 = 0;
hopcount_sp = 0;
complexity = 0;
time5=0;
time6=0;

for(j=1; j<=buffer; j++){
  klist3[j]=0;
}

for(j=1; j<=PERCENT;j++)
  klist6[j]=0;

for(j=1;j<=buffer;j++){
  mink[j]=0;
}

for(j=1;j<=N;j++)
  hopcount[j]=0;

/*Start main algorithm*/

for(i=1; i<=iter; i++)
  {
    if(i%1000==0)
          fprintf(stdout,"passed topology %d of %d\n",i,iter);
    /*
          Allocate pointer space for ordertensor. Ordertensor will later
          on store the k needed for the k-shortest paths.
    */
    ordertensor = (element***) malloc((size_t) (N+1)*sizeof(element**));
    for(s=0; s<=N;s++){
          ordertensor[s]=(element**) malloc((size_t) (N+1)*sizeof(element*));
          for(j=0; j<=N; j++){
            ordertensor[s][j]=(element*) malloc(sizeof(element));
            ordertensor[s][j]->k=0;
            ordertensor[s][j]->next=NULL;
          }
    }

    for(s=1;s<=N;s++){
          for(j=1;j<=N;j++){
            OUT_total[s][j]=0;
          }
    }
```

```
for(s=1; s<=N; s++)
      maximumvalue[s]=0;

/*
  Generate a random topology with N nodes, m link metrics and a
  linkdensity p. This function is called in randomtopol_mod.c and
  returns the topology via numadj, adj and datadj
*/
randomtopolMqos(&seed,N,m,adj,numadj,datadj,p);

connected=1;
/*
  Check whether the random topology is connected. This is done via
  the Prim algorithm: If Prim's algorithm can calculate a minimum
  spanning tree then the topology is connected. This function is
  called in prim_adj_mod.c.
*/
Prim_adj(adj,numadj,datadj,N,primlist,&connected,&treevalue);

/*
  If the topology is connected, proceed with program, otherwise
  create a new topology
*/
if(connected==1)
      {
       for(j=1; j<=buffer; j++)
         klist2[j]=0;

       for(j=1; j<=PERCENT; j++)
         klist5[j]=0;

       time3=clock();
       /*
        Start samcratensor in samcratensor_mod.c. This function will
        fill up ordertensor
       */
      for(s=1; s<=N; s++)

samcratensor(s,adj,numadj,datadj,m,A,N,klist1,buffer,ordertensor,OUT_total,maximumvalue,&hopcou
nt_sp);

       /*
        Start allpaths_sort_max_fib in samcra-cl_mod.c. This function
        actually calculates the hop-by-hop path and compares it with
        the shortest path
        (or the list of x-shortest paths, that is also created here).
       */
       for(s=N;s<=N;s++)
        {
        allpaths_sort_max_fib(s,adj,numadj,datadj,m,A,N,klist1,klist4,buffer,ordertensor,OUT_tota
        l,maximumvalue,&out2,mink,hopcount,hopcountdiff,&hopcount_sp,&complexity);

          for(j=1;j<=buffer;j++)
                klist2[j]+=klist1[j];

          for(j=1;j<=PERCENT;j++)
                klist5[j]+=klist4[j];

        }/*for*/

       time4=clock();
       time5+=(time4-time3)*0.000001;
```

```
            for(j=1; j<=buffer; j++)
              klist3[j]+=klist2[j];

            for(j=1; j<=PERCENT; j++)
              klist6[j]+=klist5[j];

            insertdata(&queue,out2-1,(getdata(queue,out2-1)+1));
            insertdata(&complex,complexity-1,(getdata(complex,complexity-1)+1));
          } /*if*/
    else
          i-=1;

    /*deallocate ordertensor,nextpointer*/
    for(s=0; s<=N; s++){
          for(j=0; j<=N; j++){
            while(ordertensor[s][j]->next!=NULL){
              nextpointer=ordertensor[s][j];
              ordertensor[s][j]=nextpointer->next;
              free(nextpointer);
            }
            free(ordertensor[s][j]);
          }
          free(ordertensor[s]);
    }
    free(ordertensor);

  }/*for(i=1;i<=iter;i++)*/

/*write results*/
resultaat=fopen("config","w");
fprintf(resultaat,"Number of topologies = %d\n",iter);
fprintf(resultaat,"Number of nodes = %d\n",N);
fprintf(resultaat,"The link density = %lf\n",p);
fprintf(resultaat,"The constraints:");
for(l=1;l<=m;l++){
  fprintf(resultaat," %lf",A[l]);
}
fprintf(resultaat,"\n");
fclose(resultaat);

resultaat=fopen("out1","w");
//fprintf(resultaat,"Number of k-shortest paths:\n");
fprintf(stdout,"\n\nNumber of k-shortest paths:");
for(j=1;j<=buffer;j++){
  fprintf(resultaat,"%d\t%d\n",j,klist3[j]);
  fprintf(stdout,"\n[%d]=%d",j,klist3[j]);
}
fclose(resultaat);

resultaat=fopen("out2","w");
//fprintf(resultaat,"Number of paths that are [x-percent] longer than the shortest path:\n");
fprintf(stdout,"\n\nNumber of paths that are [x-percent] longer than the shortest path:");
for(j=1; j<=PERCENT; j++){
  fprintf(resultaat,"%d\t%d\n",j-1,klist6[j]);
  fprintf(stdout,"\n[%d]=%d",j-1,klist6[j]);
}
fclose(resultaat);

resultaat=fopen("out3","w");
```

```
 //fprintf(resultaat,"Counter (=number of paths found * entries in queue) per topology\n");
 for(s=1;s<=length_array(queue);s++){
  if(getdata(queue,s-1)!=0)
    fprintf(resultaat,"%d\t%d\n",s,getdata(queue,s-1));
 }
 fclose(resultaat);

 resultaat=fopen("out4","w");
 //fprintf(resultaat,"Minimum k (=out) needed per topology to get shortest path\n");
 for(s=1;s<=buffer;s++)
  fprintf(resultaat,"%d\t%d\n",s,mink[s]);
 fclose(resultaat);

 resultaat=fopen("out5","w");
 //fprintf(resultaat,"Number of times the hop-by-hop samcra solution consists of x hops\n");
 for(s=1;s<=N;s++)
  fprintf(resultaat,"%d\t%d\n",s,hopcount[s]);
 fclose(resultaat);

 resultaat=fopen("out6","w");
 //fprintf(resultaat,"difference in hopcount between shortest and hop-by-hop path\n");
 for(s=-N;s<=N;s++)
  fprintf(resultaat,"%d\t%d\n",s,hopcountdiff[s]);
 fclose(resultaat);

 resultaat=fopen("out7","w");
 //fprintf(resultaat,"The complexity per topology\n");
 for(s=1;s<=length_array(complex);s++){
  if(getdata(complex,s-1)!=0)
    fprintf(resultaat,"%d\t%d\n",s,getdata(complex,s-1));
 }
 fclose(resultaat);

 fprintf(stdout,"\nTIME = %lf\n",time5);

 /*free memory*/

 delete_array(&queue);

 free_imatrix(adj,1,N,1,50);
 free_ivector(numadj,1,N);
 free_ivector(primlist,1,N);
 free_d3tensor(datadj,1,m,1,N,1,50);
 free_imatrix(OUT_total,1,N,1,N);
 free_ivector(klist1,1,buffer);
 free_ivector(klist2,1,buffer);
 free_ivector(klist3,1,buffer);
 free_ivector(klist4,1,PERCENT);
 free_ivector(klist5,1,PERCENT);
 free_ivector(klist6,1,PERCENT);
 free_dvector(A,1,m);
 free_dvector(maximumvalue,1,N);
 free_ivector(mink,1,buffer);
 free_ivector(hopcount,1,N);
 free_ivector(hopcountdiff,-N,N);
}
```

```
/*
samcratensor_mod.c:
This module calculates the necessary information needed to calculate
the hop-by-hop path in the module samcra-cl_mod.c.
This module calculates the shortest path between each node and the
destination node and stores this information in ordertensor.
Ordertensor keeps track of the k (queue-size) that is needed for the
shortest path, second shortest path, etc. (until k=1 or maximum
number of paths has been reached).
The module is referenced from samcra-cl.c for each node. This is done
so that we have a path between each pair of nodes.

Hans De Neve, 24/07/98 (created tamcratensor_mod2.c)
F.A. Kuipers 25/01/2000 (this module has been derived from
tamcratensor_mod2.c. Some changes that were made are marked in this
file. Because SAMCRA is a self-adaptive (= dynamic) algorithm, I had
to adjust previ, prevj, poscounter and sum so that they would
dynamically allocate their memory)
*/

#include <string.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "/home/mieghemp/nr/include/nrutil.h"
/*
  Because samcra needs a dynamic array I created a module
  linked_list_mod.c. The functions in this module are linked via
  linked_list.h
*/
#include "linked_list.h"
#define NULL 0
#define POWER 10

/*global definition of the type fiblist, see [Cormen ea., 1997]:
  Fibonacci lists*/
struct fiblist {
        double key;
        int    id;
        int    el;
        int    degree;
        int    mark;
        struct fiblist *parent;
        struct fiblist *child;
        struct fiblist *left;
        struct fiblist *right;
        };

/*dynamic array of integers as used in linked_lsit_mod.c*/
typedef struct el{
        int k;
        int first;
        struct el *next;
        } element;

/*dynamic array of doubles as used in linked_list_mod.c*/
struct dynamic_array{
  int data;
  int index;
  struct dynamic_array *next;
};

struct dynamic_vector{
```

```
  double data;
  int index;
  struct dynamic_vector* next;
};

void
samcratensor(END,adj,max,datadj,numm,A,aantalnodes,klist,buffer,ordertensor,OUT_total,maximumv
alue,hopcount_sp)

int END;
int **adj;
int *max;
double ***datadj;
int numm;
int aantalnodes;
int *klist;
double *A;
int buffer;
element ***ordertensor;
int **OUT_total;
double *maximumvalue;
int *hopcount_sp;

{
  int aantalpaden,out,countdown;
  int i,j,l,m,n,k,a,u,v,q,r,s,p,*counter,*num;
  double ref, newvalue, *intermediate;
  struct dynamic_array **previ, **prevj, **poscounter;
  struct dynamic_vector ***sum;
  struct fiblist *helppo, **heap;

/*declaration of the functions maintaining the Fibonacci heap,see
  [Cormen ea., 1997] for implementing and maintaining Fibonacci
  heaps*/
  void create_heap();
  struct fiblist *extract_min_from_list();
  struct fiblist *insert();
  void consolidate();
  void decrease_key();
  void cut();
  void cascading_cut();

  element *nextpointer, *newpointer;

  /*memory allocation*/

  aantalpaden=buffer;
  /*counter keeps track of the number of times a path is stored in the
     queue*/
  counter = ivector(1,aantalnodes);
  num = ivector(1,1);
  intermediate = dvector(1,numm);
  /*previ: previous node on the path*/
  previ = ivector(1,aantalnodes);
  /*prevj: index of previous node*/
  prevj = ivector(1,aantalnodes);
  poscounter = ivector(1,aantalnodes);
  /*sum: keeps track of the sum of the linkmetrics on the path*/
  sum = dmatrix(1,numm,1,aantalnodes);
  heap = ivector(1,1);
```

```
for(i=1;i<=aantalnodes;i++){
  previ[i]=NULL;
  prevj[i]=NULL;
  poscounter[i]=NULL;
}

for(i=1;i<=numm;i++){
  for(j=1;j<=aantalnodes;j++){
    sum[i][j]=NULL;
  }
}

for(i=1;i<=aantalnodes;i++)
  counter[i]=0;

create_heap(heap,num);
helppo=insert(heap,num,0.0,END,1);
counter[END]+=1;
for(l=1; l<=numm; l++){
  insertdatad(&sum[l][END],0,0.0);
}

k=0;
/*countdown = aantalnodes: because only N-1 connections are measured,
  but in this case also END is taken into account*/
countdown=aantalnodes;
(*hopcount_sp) = 1;

/*main algorithm*/

while(countdown > 0)
  {
    /*
        Select the minimum element from the Fibonacci list
    */
    helppo=extract_min_from_list(heap,num);

    /*If fiblist is empty before we have a k=1 path for each
      destination*/
    if(helppo==NULL)
          {
           countdown=0;
          }
    else
          {
           i=helppo->id;    /*i represents the extracted node from the fib
                              heap*/
           j=helppo->el;

           if(helppo->key > maximumvalue[END])
             maximumvalue[END]=helppo->key;

           /*keep track of the number of entries that is truly used*/
           if(i==END)
             q=1;
           else{
             q=getdata(poscounter[getdata(previ[i],j-1)],getdata(prevj[i],j-1)-1);
           }
           OUT_total[END][i]++;
           for(s=1;s<=length_array(poscounter[i]);s++){
```

```
 if(getdata(poscounter[i],s-1)==q){
  q++;
  s=0;
 }/*if*/
}/*for*/
insertdata(&poscounter[i],j-1,q);

nextpointer=ordertensor[END][i];
/*run through the list until the last element which is the one
  with NULL pointer*/
while((nextpointer->next)!=NULL)
 nextpointer=nextpointer->next;
if(nextpointer->k==0){
 nextpointer->k=q;
 if(i!=END){            /*exception for first entry*/
  u=i;
  v=j;
  while((getdata(previ[u],v-1))!=END){
        s=u;
        u=getdata(previ[u],v-1);
        v=getdata(prevj[s],v-1);
  }
  nextpointer->first=u;  /*this is the first entry of that path*/
 }
 else
  nextpointer->first=0;
 if(q!=1){
  /*create a new element for the next entry, which will surely
    come!*/
  newpointer=(element*) malloc(sizeof(element));
  nextpointer->next=newpointer;
  newpointer->k=0;
  newpointer->next=NULL;
 }
 else
  countdown-=1;
}

ref=helppo->key;
for(m=1; m<=max[i]; m++)
 {
  /*Go to the mth neighbour of node i*/
  a=adj[i][m];

  newvalue=0;
  /*
       intermediate[l] = sum of all the lth constraints up to
       node a. newvalue = maximum of intermediate[l] divided by
       the lth constraint
  */
  for(l=1; l<=numm; l++)
       {
        intermediate[l]=(getdatad(sum[l][i],j-1)+datadj[l][i][m]);
        if((intermediate[l]/A[l]) > newvalue)
          newvalue=intermediate[l]/A[l];
       } /*for*/

  p=1;
  /*
       Check if a is not dominated. This is checked against all
       entries that have entered the queue of node a. If a is
```

```
            not dominated then p=counter[a]+1. Counter[a] represents
            the number of times an entry entered the queue of a.
        */
        while(p<=counter[a]){
            l=1;
            while(l<=numm){
              if((intermediate[l]-getdatad(sum[l][a],p-1)) >= 0.0)
                l++;
              else
                l=numm+2;
            }
            if(l==(numm+1))
              p=(counter[a]+1);
            p++;
        }

        /*
            newvalue shouldn't exceed 1, because then the constraints
            aren't met. The new entry may also not be dominated
            (p=counter[a]+1). Then we can insert the node. First we
            keep track of the path by inserting a's previous node i
            and its place in the queue j. We also keep track of the
            sum of the metrics up to node a. Then we insert the node
            into the fibonacci heap and increase the counter.
        */
        /*the fundamental changes made to create SAMCRA were done at
          this point*/
        if((newvalue <=1.0)&&(p==(counter[a]+1)))
            {
              p=counter[a]+1;
              insertdata(&previ[a],p-1,i);
              insertdata(&prevj[a],p-1,j);
              for(l=1; l<=numm; l++){
                insertdatad(&sum[l][a],p-1,intermediate[l]);
              }
              helppo=insert(heap,num,newvalue,a,p);
              counter[a]+=1;
            }/*newvalue*/
        }/*for*/
      }/*else*/
  }/*while*/

/*free memory*/

for(a=1;a<=aantalnodes;a++){
  delete_array(&previ[a]);
  delete_array(&prevj[a]);
  delete_array(&poscounter[a]);
}

for(a=1;a<=numm;a++){
  for(i=1;i<=aantalnodes;i++){
      delete_vector(&sum[a][i]);
  }
}

free_ivector(counter,1,aantalnodes);
free_ivector(num,1,1);
free_dvector(intermediate,1,numm);
free_ivector(previ,1,aantalnodes);
free_ivector(prevj,1,aantalnodes);
```

```
  free_ivector(poscounter,1,aantalnodes);
  free_dmatrix(sum,1,numm,1,aantalnodes);
  free_ivector(heap,1,1);

}

/*
samcra-cl_mod.c:
This module calculates the position of hop-by-hop paths in the list
of all loop-free paths.
Input is given from the samcra-cl.c module and samcratensor_mod.c.

Hans De Neve, 24/07/98
F.A. Kuipers, 25/01/2000 (this module was originally used for tamcra-
cl_mod2.c, but has now been altered for SAMCRA in an analogous way as
done with samcratensor_mod.c. The first while loop is similar as
samcratensor and therefore some comments were omitted.)
*/

#include <string.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include "/home/mieghemp/nr/include/nrutil.h"
#include "linked_list.h"
#define NULL 0
#define POWER 10
#define ADJACENT 50
#define PERCENT 100

struct fiblist {      /*global definition of the type fiblist*/
          double key;
          int    id;
          int    el;
          int    degree;
          int    mark;
          struct fiblist *parent;
          struct fiblist *child;
          struct fiblist *left;
          struct fiblist *right;
          };

typedef struct el{
          int k;
          int first;
          struct el *next;
          } element;

struct dynamic_array{
  int data;
  int index;
  struct dynamic_array *next;
};

struct dynamic_vector{
  double data;
  int index;
  struct dynamic_vector* next;
};
```

```
void
allpaths_sort_max_fib(END,adj,max,datadj,numm,A,aantalnodes,klist1,klist4,buffer,ordertensor,OUT_
total,maximumvalue,out2,mink,hopcount,hopcountdiff,hopcount_sp,complexity)

int END;
int **adj;
int *max;
double ***datadj;
int numm;
int aantalnodes;
int *klist1;
int *klist4;
double *A;
int buffer;
element ***ordertensor;
int **OUT_total;
double *maximumvalue;
int *out2;
int *mink, *hopcount, *hopcountdiff, *hopcount_sp, *complexity;

{
  int aantalpaden,out,countdown, *maxpath, pathcounter, *path, kshortest,total_dominated, false_route;
  int i,ii,j,l,m,n,k,kk,E,a,u,q,r,s,ss,p,w,z,check,*counter;
  int *num, *OUT, *OUT_overall, non_dominated, hopcount_shortest, hopdiff;
  double ref, newvalue, *intermediate, complex;
  struct dynamic_array **previ, **prevj, ***nexti, ***nextj, **poscounter, **poscounter_overall,
**non_dom;
  struct dynamic_vector ***sum, **key_overall;
  double *key_shortest, *diff;
  element *nextpointer;
  struct fiblist *helppo, **heap;

  /*declaration of the functions maintaining the Fibonacci heap*/
  void create_heap();
  struct fiblist *extract_min_from_list();
  struct fiblist *insert();
  void consolidate();
  void decrease_key();
  void cut();
  void cascading_cut();

  FILE    *intern;

  /*memory allocation*/

  /*aantalpaden: maximum number of paths samcra will consider*/
  aantalpaden=buffer;
  /*maxpath is used for determing which hop-by-hop path is found*/
  maxpath = ivector(1,aantalnodes);
  /*path keeps track of the hop-by-hop path*/
  path    = ivector(1,aantalnodes);
  /*counter keeps track of the amount of elements that entered the
    queue*/
  counter = ivector(1,aantalnodes);
  num = ivector(1,1);
  intermediate = dvector(1,numm);
  previ = ivector(1,aantalnodes);
  prevj = ivector(1,aantalnodes);
  non_dom = ivector(1,aantalnodes);
  nexti = imatrix(1,aantalnodes,1,ADJACENT);
```

```
nextj = imatrix(1,aantalnodes,1,ADJACENT);
poscounter = ivector(1,aantalnodes);
poscounter_overall = ivector(1,aantalnodes);
sum = dmatrix(1,numm,1,aantalnodes);
heap = ivector(1,1);
OUT = ivector(1,aantalnodes);
OUT_overall = ivector(1,aantalnodes);
key_overall = dvector(1,aantalnodes);
key_shortest = dvector(1,aantalnodes);
diff = dvector(1,aantalnodes);

for(i=1;i<=aantalnodes;i++){
  previ[i]=NULL;
  prevj[i]=NULL;
  non_dom[i]=NULL;
  poscounter[i]=NULL;
  poscounter_overall[i]=NULL;
  for(a=1;a<=ADJACENT;a++){
    nexti[i][a]=NULL;
    nextj[i][a]=NULL;
  }
}

for(i=1;i<=aantalnodes;i++){
  key_overall[i]=NULL;
}

for(i=1;i<=numm;i++){
  for(j=1;j<=aantalnodes;j++){
    sum[i][j]=NULL;
  }
}

for(i=1;i<=aantalnodes;i++)
  counter[i]=0;

create_heap(heap,num);
helppo=insert(heap,num,0.0,END,1);
insertdata(&non_dom[END],0,1);
counter[END]+=1;
for(l=1; l<=numm; l++){
  insertdatad(&sum[l][END],0,0.0);
}

k=0;
countdown=aantalpaden*aantalnodes;
ii=0;

for(i=1;i<=aantalnodes;i++){
  maxpath[i]=1;
  diff[i]=0;
}

for(i=1; i<=aantalnodes;i++){
  OUT[i]=0;
  OUT_overall[i]=0;
}

/*main algorithm*/
```

```
/*
 The while loop calculates all loop-free paths.
 Also the dominated paths are taken into account,
 because the hop-by-hop path may be a dominated path.
(samcratensor only calculated the non-dominated paths)
*/
while(countdown > 0)
 {
  helppo=extract_min_from_list(heap,num);
  if(helppo==NULL)
        {
         countdown=0;
        }
  else
        {
         countdown-=1;
         i=helppo->id;
         j=helppo->el;

         /*
                Count the number of hops for the shortest path.
                This is used for comparison with the hop-by-hop path.
         */
         if((i==1)&&(ii==0)){
           ii++;
           q=i;
           r=j;
           while(q!=END){
                 (*hopcount_sp)++;
                 s=q;
                 q=getdata(previ[q],r-1);
                 r=getdata(prevj[s],r-1);
            }
          }

         if(getdata(non_dom[i],j-1)==1)
           insertdata(&poscounter[i],j-1,(++OUT[i]));
         insertdata(&poscounter_overall[i],j-1,(++OUT_overall[i]));
         ref=helppo->key;
         insertdatad(&key_overall[i],j-1,ref);
         if(OUT_overall[i]==1)
           key_shortest[i]=ref;
         for(m=1; m<=max[i]; m++)
           {
            /*
                Check if no loops are made
            */
            check=1;
            q=i;
            r=j;
            while(q!=END)
                 {
                  if(adj[i][m]==q){
                    check=0;
                  }
                  s=q;
                  q=getdata(previ[q],r-1);
                  r=getdata(prevj[s],r-1);
                 } /*while*/
            if(adj[i][m]==q)
                 check=0;
```

```
if(check==1){
        a=adj[i][m];
        newvalue=0;
        for(l=1; l<=numm; l++)
          {
            intermediate[l]=(getdatad(sum[l][i],j-1)+datadj[l][i][m]);
            if((intermediate[l]/A[l]) > newvalue)
              newvalue=intermediate[l]/A[l];
          } /*for*/
        p=1;
        while(p<=counter[a]){
          l=1;
          while(l<=numm){
            if((intermediate[l]-getdatad(sum[l][a],p-1)) >= 0.0)
              l++;
            else
              l=numm+2;
          }
          if(l==(numm+1))
            p=(counter[a]+1);
          p++;
        }
        if(p==(counter[a]+1)&&(getdata(non_dom[i],j-1)==1))
          non_dominated=1;
        else
          non_dominated=0;

        total_dominated=0;
        if(OUT[a] >= OUT_total[END][a]){
          p=1;
          while(p<=counter[a]){
            if(getdata(non_dom[a],p-1)==0)
              p++;
            else{
              l=1;
              while(l<=numm){
                    if((intermediate[l]-getdatad(sum[l][a],p-1)) >= 0.0)
                      l++;
                    else{
                      l=numm+2;
                      p=counter[a]+2;
                    }
              }
              if(l==(numm+1))
                    p++;
            }
          }
          if(p==(counter[a]+1))
            total_dominated=1;
        }
        /*Dominated paths are also examined. Now we get the list
          of all loop-free paths. It is necessary to consider
          dominated paths, because hop-by-hop samcra can create a
          dominated path!*/
        if(newvalue <=maximumvalue[END])
          {
            p=counter[a]+1;
            insertdata(&previ[a],p-1,i);
            insertdata(&prevj[a],p-1,j);
            insertdata(&non_dom[a],p-1,non_dominated);
```

```
                    w=1;
                    while(getdata(nexti[i][w],j-1)!=0)
                      w++;
                    insertdata(&nexti[i][w],j-1,a);
                    insertdata(&nextj[i][w],j-1,p);
                    for(l=1; l<=numm; l++){
                      insertdatad(&sum[l][a],p-1,intermediate[l]);
                    }
                    helppo=insert(heap,num,newvalue,a,p);
                    counter[a]+=1;
                   }  /*newvalue*/
              }  /*if*/
            }  /*for*/
          }/*else*/
  }/*while*/

/*
 Calculate the hop-by-hop path and compare it to the source-
 based loop-free paths calculated above.
*/
for(i=1; i<=1; i++){
  false_route=0;
  if(i!=END){
    pathcounter=0;
    kshortest=1;
    nextpointer=ordertensor[i][END];
    /* removed for samcra, because samcra has no limited queue-size
       while((nextpointer->k) > kmax){
         nextpointer=nextpointer->next;
         kshortest++;
    }*/
    if((nextpointer->k)==0){
          /*
           In some cases the hop-by-hop path can't be retrieved from the
           list of loop-free paths. They are then assigned
           "maxpath[i]=aantalpaden". The results of these paths are
           however distributed in the same way as the results for the
           other paths.
          */
          maxpath[i]=aantalpaden;     /*no k=1 path*/
    }
    else if(kshortest>1){
          j=1;
          while((getdata(previ[i],j-1)!=(nextpointer->first))&&(j<=aantalpaden))
            j++;
          if(j==(aantalpaden+1)){
            maxpath[i]=aantalpaden;   /*no path with the proper first hop*/
          }
          else{
            kshortest=getdata(poscounter_overall[i],j-1);
            maxpath[i]=kshortest;
          }
    }

    if(maxpath[i]!=aantalpaden){
          /*look for the shortest entry*/
          /*starting from the destination point i*/
          for(j=1; j<=aantalpaden; j++){
            if(getdata(poscounter_overall[i],j-1)==kshortest){
              q=i;
              r=j;
```

```
while(q!=END){
 /*scanning back as long as this is the shortest entry*/
 while((getdata(poscounter_overall[q],r-1)==kshortest)&&(q!=END)){
      /*
       Check if hop-by-hop tamcra creates a loop.
       This is not necessary for samcra.

      for(ss=1;ss<=pathcounter;ss++){
        if(path[ss]==q){
          (*loop)++;
          q=END;
        }
      }*/
      path[++pathcounter]=q;
      s=q;
      q=getdata(previ[q],r-1);
      r=getdata(prevj[s],r-1);
      kshortest=1;
      nextpointer=ordertensor[q][END];
      /*while((nextpointer->k) > kmax){  removed for SAMCRA
        nextpointer=nextpointer->next;
        kshortest++;
      }*/
      if((nextpointer->k)==0){
       maxpath[i]=aantalpaden;   /*no k=1 path*/
       q=END;
      }
      else if(kshortest>1){
       u=1;
       while((getdata(previ[q],u-1)!=(nextpointer->first))&&(u<=aantalpaden))
        u++;
       if(u==(aantalpaden+1)){
        maxpath[i]=aantalpaden; /*no path with the proper next hop*/
        q=END;
       }
       else
        kshortest=getdata(poscounter_overall[q],u-1);
      } /*else if*/
 } /*while*/

 if(q!=END){
      /*look for a new entry with 1*/
      for(s=1; s<=aantalpaden; s++){
       if(getdata(poscounter_overall[q],s-1)==0){
        /*which happens when a path with the proper history was
          not found*/
        for(s=1; s<=aantalpaden; s++){
         if(getdata(poscounter_overall[q],s-1)==kshortest){
               r=s;
               s=aantalpaden+1;
               false_route=1;
         }
         if(s==aantalpaden){
               maxpath[i]=aantalpaden;
               q=END;
         }
        }
       }
       else if(getdata(poscounter_overall[q],s-1)==kshortest){
       /*checking the history*/
        m=q;
```

```
                    n=s;
                    z=pathcounter;
                    w=1;
                    while((w<=ADJACENT)&&(z>=1)){
                     if(getdata(nexti[m][w],n-1)==path[z]){
                          z--;
                          p=m;
                          m=getdata(nexti[m][w],n-1);
                          n=getdata(nextj[p][w],n-1);
                          w=1;
                     }
                     else
                          w++;
                    }
                    if(z==0){
                     maxpath[i]=getdata(poscounter_overall[m],n-1);
                     /*the value we're looking for*/
                     /*the difference in length with the shortest path*/
                     diff[i]=100.0*(getdatad(key_overall[m],n-1)-key_shortest[m])/key_shortest[m];
                     r=s;          /*this is the new entry to start with above*/
                     s=aantalpaden+1;
                     false_route=0;
                    } /*if*/
                   } /*else if poscounter*/
                  } /*s=aantalpaden*/
                  if(s==(aantalpaden+1)){
                   maxpath[i]=aantalpaden;
                   q=END;
                  }
                } /*q!=END*/
                else
                     j=aantalpaden;     /*leaving loop as q=END*/
               }/*while*/
              } /*if*/
             } /*for*/
   } /*if*/
  } /*if*/
  if(false_route==1){
    maxpath[i]=aantalpaden;
  }
} /*for*/

for(j=1; j<=aantalpaden; j++)
  klist1[j]=0;

for(j=1; j<=PERCENT;j++)
  klist4[j]=0;

for(i=1; i<=1; i++){
  if(i!=END){
    klist1[maxpath[i]]+=1;
  }
}

for(i=1; i<=1; i++){
  if((i!=END)&&(maxpath[i]!=aantalpaden)){
    klist4[(int)(diff[i]+1)]+=1;
  }
}
```

```
/*Which k (queue-size) is needed to get the shortest path*/
out=0;
for(a=1;a<=pathcounter;a++){
  if((ordertensor[path[a]][END]->k)>out)
    out = ordertensor[path[a]][END]->k;
}

/*What's the maximum counter used*/
(*out2)=0;
for(a=1;a<=aantalnodes;a++){
  if(counter[a]>(*out2))
    (*out2) = counter[a];
}

/*calculate the complexity (amount of operations) used to find the
  hop-by-hop path*/
kk=0;
complex=0;
E=0;
for(a=1;a<=aantalnodes;a++){
  E +=max[a];
}
E = E/2;
for(a=1;a<=pathcounter;a++){
  kk=ordertensor[path[a]][END]->k;
  complex += kk*aantalnodes*log(kk*aantalnodes) + kk*kk*numm*E;
}
(*complexity) = (int) complex;

if((out>0)&&(out<=aantalpaden))
  mink[out]++;
hopcount_shortest = (*hopcount_sp);
if(hopcount_shortest>aantalnodes)
  hopcount_shortest = pathcounter;
hopdiff = (pathcounter - hopcount_shortest);

intern=fopen("topol","a");
if(mink[out]==1){
  fprintf(intern,"\nMinimum k for shortest path = %d\n",out);
  for(j=1;j<=aantalnodes;j++){
    fprintf(intern,"\nnumadj[%d]:",j);
    for(s=1;s<=max[j];s++){
          fprintf(intern," %d(",adj[j][s]);
          for(a=1;a<=numm;a++){
            fprintf(intern," %.2lf", datadj[a][j][s]);
          }
          fprintf(intern,")");
    }
  }
  fprintf(intern,"\nHop-by-hop path:");
  for(ii=1;ii<=pathcounter;ii++){
    fprintf(intern," %d",path[ii]);
  }
  fprintf(intern," %d\n",END);
}
fclose(intern);

hopcount[pathcounter]++;
hopcountdiff[hopdiff]++;
```

```
 /*free memory*/
 for(a=1;a<=aantalnodes;a++){
  delete_array(&previ[a]);
  delete_array(&prevj[a]);
  delete_array(&poscounter[a]);
  delete_array(&non_dom[a]);
  delete_array(&poscounter_overall[a]);
  for(i=1;i<=ADJACENT;i++){
   delete_array(&nexti[a][i]);
   delete_array(&nextj[a][i]);
  }
  delete_vector(&key_overall[a]);
 }

 for(a=1;a<=numm;a++){
  for(i=1;i<=aantalnodes;i++){
   delete_vector(&sum[a][i]);
  }
 }

 free_ivector(counter,1,aantalnodes);
 free_ivector(maxpath,1,aantalnodes);
 free_ivector(path,1,aantalnodes);
 free_ivector(num,1,1);
 free_dvector(intermediate,1,numm);
 free_ivector(previ,1,aantalnodes);
 free_ivector(prevj,1,aantalnodes);
 free_ivector(non_dom,1,aantalnodes);
 free_imatrix(nexti,1,aantalnodes,1,ADJACENT);
 free_imatrix(nextj,1,aantalnodes,1,ADJACENT);
 free_ivector(poscounter,1,aantalnodes);
 free_ivector(poscounter_overall,1,aantalnodes);
 free_ivector(OUT,1,aantalnodes);
 free_ivector(OUT_overall,1,aantalnodes);
 free_dvector(key_overall,1,aantalnodes);
 free_dvector(key_shortest,1,aantalnodes);
 free_dvector(diff,1,aantalnodes);
 free_dmatrix(sum,1,numm,1,aantalnodes);
 free_ivector(heap,1,1);
}
/*
linked_lis_mod.c:
This module is created to simulate a dynamic array and to make/alter
linked lists
F.A. Kuipers 25/01/2000*/

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <assert.h>

/*dynamic_array represents an array of integers*/
struct dynamic_array{
 int data;                    /*data gives the data in the array*/
 int index;                   /*index gives the place in the array*/
 struct dynamic_array* next;  /*point to the next element in the array*/
};

/*dynamic_vector represents an array of doubles*/
struct dynamic_vector{
```

```
 double data;
 int index;
 struct dynamic_vector* next;
};

/*This function calculates the length of a dynamic_array, by
  retrieving the maximum index number*/
int length_array(struct dynamic_array* head)
{
 struct dynamic_array* current = head;
 int count = 0;

 if(current!=NULL){
   while(current!=NULL){
     if((current->index)>count)
           count = current->index;
     current = current->next;
   }
   return count+1;
 }
 else
   return 0;
}

/*This function puts an element at the head of the array*/
void putdata(struct dynamic_array** headref, int index, int data)
{
 struct dynamic_array* newnode = malloc(sizeof(struct dynamic_array));

 newnode->data = data;
 newnode->index = index;
 newnode->next = (*headref);
 (*headref) = newnode;
}

/*This function retrieves the data placed at a certain index.
  If there is no data present, a 0 will be returned*/
int getdata(struct dynamic_array* head, int index)
{
 struct dynamic_array* current = head;
 int count = 0;          /*the index of the node we're currently looking at*/
 int result;

 while(current != NULL){
   if(current->index==index)
     return(current->data);
   count++;
   current = current->next;
 }
 return(0);
}

/*This function inserts data at a certain place in the array (index).
  This function calls the function putdata*/
void insertdata(struct dynamic_array** headref, int index, int data)
{
 if(*headref==NULL){
   putdata(headref,index,data);
 }
 else{
```

```
   struct dynamic_array* current = *headref;
   struct dynamic_array* prevcur;

   while((current!=NULL)&&(current->index!=index)){
     prevcur = current;
     current = current->next;
   }

   if(current!=NULL){
          current->data = data;
   }
   else{
     putdata(&(prevcur->next),index,data);
   }
  }
}

/*This function deallocates the memory allocated for the
  dynamic array. When a dynamic array isn't used any more,
  this function should be called*/
void delete_array(struct dynamic_array** headref)
{
  struct dynamic_array* current = *headref;
  struct dynamic_array* next;

  while(current!=NULL){
    next = current->next;
    free(current);
    current = next;
  }

  *headref = NULL;
}

/*The functions below are the same as above, but apply to
  the dynamic array of doubles*/

int length_vector(struct dynamic_vector* head)
{
  struct dynamic_vector* temp = head;
  int count = 0;

  if(temp!=NULL){
    while(temp!=NULL){
      if((temp->index)>count)
          count = temp->index;
      temp = temp->next;
    }
    return count+1;
  }
  else
    return 0;
}

void putdatad(headref, index, data)
struct dynamic_vector** headref;
int index;
double data;
{
  struct dynamic_vector* newnode = malloc(sizeof(struct dynamic_vector));
```

```
  newnode->data = data;
  newnode->index = index;
  newnode->next = (*headref);
  (*headref) = newnode;
}

double getdatad(head, index)
struct dynamic_vector* head;
int index;
{
  struct dynamic_vector* new = head;
  int count = 0;
  double result;

  while(new != NULL){
    if(new->index==index){
      result = (new->data);
      return result;
    }
    count++;
    new = new->next;
  }
  return 0.0;
}

void insertdatad(headref, index, data)
struct dynamic_vector** headref;
int index;
double data;
{
  if(*headref==NULL){
    putdatad(headref, index, data);
  }
  else{
    struct dynamic_vector* current = *headref;
    struct dynamic_vector* prevcur;

    while((current!=NULL)&&(current->index!=index)){
      prevcur = current;
      current = current->next;
    }
    if(current!=NULL){
            current->data = data;
    }
    else{
      putdatad(&(prevcur->next), index, data);
    }
  }
}

void delete_vector(struct dynamic_vector** headref)
{
  struct dynamic_vector* current = *headref;
  struct dynamic_vector* next;

  while(current!=NULL){
    next = current->next;
    free(current);
    current = next;
```

```
  }

  *headref = NULL;
}
```

References

Andrew, L.H. and A.A.N. Kusuma, 1998, "*Generalized Analysis of a QoS-aware routing algorithm*", IEEE GLOBECOM 1998, Piscataway, NJ, USA, vol. 1, pp. 1-6.

Apostolopoulos, G., D. Williams, S. Kamat, R. Guérin, A. Orda and T. Przygienda, 1999, "*QoS Routing Mechanisms and OSPF extensions*", RFC 2676, August.

Bennet, J. and H. Zhang, 1996, "*Hierarchical Packet Fair Queueing Algorithms*", ACM SIGCOMM '96, August.

Blake, S., D. Black, M. Carlson, E. Davies, Z. Wang, W. Weiss, 1998, "*An Architecture for Differentiated Services*", RFC 2475, December.

Braden, R., L. Zhang, S. Berson, S. Herzog and S. Jamin, 1997, "*Resource ReSerVation Protocol (RSVP) – Version 1 Functional Spec*", RFC 2205, September.

Calvert K., M. Doar, E. Zegura, 1997, "*Modelling Internet Topology*", IEEE Comm. Magazine, pp. 160-163.

Calvert, K.L., ed., 1999, "*Architectural Framework for Active Networks, version 1*", RFC draft, July 27.

Chen, S. and K. Nahrstedt, 1998a, "*On Finding Multi-constrained Paths*", Proc. ICC '98, Atlanta, Georgia.

Chen, S. and K. Nahrstedt, 1998b, "*An Overview of Quality of Service Routing for Next-Generation High-Speed Networks: Problems and Solutions*", IEEE Network, November/December.

Cook, S.A., 1971, "*The complexity of theorem-proving procedures*", Proc. 3rd Ann. ACM Symp. On Theory of Computing, Association for Computing Machinery, New York, pp. 151-158.

Cormen, T.H., C.E. Leiserson, R.L. Rivest, 1997, "*Introduction to algorithms*", The MIT Press, Cambridge, MA, ISBN 0-262-03141-8.

Crawley, E., R. Nair, B. Rajagopalan and H. Sandick, 1998, "*A Framework for QoS-based Routing in the Internet*", RFC 2386, August.

Demers, A., S. Keshav and S. Shenker, 1989, "*Analysis and Simulation of a fair Queuing Algorithm*", ACM SIGCOMM '89, pp. 3-12.

De Neve, H. and P. Van Mieghem, 1998, "*A multiple quality of service routing algorithm for PNNI*", IEEE ATM workshop, Fairfax, May 26-29, pp. 324-328.

De Neve, H. and P. Van Mieghem, 2000, "*TAMCRA: A Tunable Accuracy Multiple Constraints Routing Algorithm*", Computer Communications 23, pp. 667-679.

Dijkstra, E.W., 1959, "*A note on two problems in connection with graphs*", Numerische Mathematik 1:269-271.

Ergün, F., R. Sinha and L. Zhang, 2000, "*QoS routing with performance dependent costs*", INFOCOM 2000, Tel Aviv, Israel, March 26-30.

Garey, M.R. and D.S. Johnson, 1979, "*Computers and Intractability: A Guide to the Theory of NP-completeness*", W.H. Freeman and company, San Francisco, ISBN 0-7167-1044-7.

Golestani, S., 1994, "*A Self-Clocked Fair Queuing Scheme for Broadband Applications*", IEEE INFOCOMM '94, pp. 636-646, June.

Golub, G.H. and C.F. van Loan, 1983, "*Matrix Computations*", North Oxford Academic, first edition, Oxford.

Guérin, R.A. and A. Orda, 1999, "*QoS Routing in Networks with Inaccurate Information: Theory and Algorithms*", IEEE/ACM Transactions on Networking, vol. 7, no. 3, pp. 350-364, June.

Guérin, R.A. and A. Orda, 2000, "*Networks with advance reservations: The routing perspective*", INFOCOM 2000, Israel, March 26-30.

Hassin, R., 1992, "*Approximation schemes for the restricted shortest path problem*", Mathematics of Operations research, 17(1): 36-42, February.

Henig, M.I., 1985, "*The shortest path problem with two objective functions*", European J. of Operational Research 25, pp. 281-291.

Iwata, A., R. Izmailov, D.-S. Lee, B. Sengupta, G. Ramamurthy and H. Suzuki, 1996, "*ATM Routing Algorithms with Multiple QoS Requirements for Multimedia Internetworking*", IEICE Trans. Commun., vol. E79-B, no. 8, August.

Jaffe, J.M., 1984, "*Algorithms for Finding Paths with Multiple Constraints*", Networks, vol.14, pp.95-116.

Lee, W.C., M.G. Hluchyj and P.A. Humblet, 1995, "*Routing subject to quality of service constraints in integrated networks*", IEEE Network, vol. 9, no. 4, pp. 46-55, July/August.

Liang, G. and I. Matta, 1998, "*Search Space Reduction in QoS Routing*", College of Computer Science, Northeastern University, http://www.ccs.neu.edu/home/matta/publications.html

Ma, Q. and P. Steenkiste, 1997, "*Quality of Service Routing with Performance Guarantees*", Proc. 4th International IFIP Workshop on QoS, May.

Malkin, G., 1994, "*RIP Version 2 – Carrying Additional Information*", RFC 1723, November.

Moy, J., 1998, "*OSPF Version 2*", STD 54, RFC 2328, April.

Nelakuditi, S., Z.-L. Zhang and R.P. Tsang, 2000, "*Adaptive Proportional Routing: A localized QoS Routing Approach*", INFOCOM 2000, Tel Aviv, Israel, March 26-30.

Orda, A. and A. Sprintson, 2000, "*QoS routing: The Precomputation Perspective*", INFOCOM 2000, Tel Aviv, Israel, March 26-30.

Psounis, K., 1999, "*Active Networks: Applications, Security, Safety, and Architectures*", IEEE Communications Surveys, First quarter.

Royden, H.L., 1988, "*Real Analysis*", Macmillan Publishing Company, 3rd edition, New York.

Salama, H.F., D.S. Reeves and Y. Viniotis, 1997, "*A Distributed Algorithm for Delay-Constrained Unicast Routing*", Proc. IEEE INFOCOMM '97, Japan, April.

Shenker, S., C. Patridge and R. Guérin, 1997, "*Specification of Guaranteed Quality of Service*", RFC 2212, September.

Sun, Q. and H. Langendorfer, 1997, "*A New distributed Routing Algorithm with End-to-End delay Guarantee*", Proc. IFIP Fifth International Workshop on Quality of Service, Columbia University, New York, May.

Tennenhouse, D.L., S.J. Garland, L. Shrira and M.F. Kaashoek, 1996, "*From Internet to ActiveNet*", RFC January, http://www.tns.lcs.mit.edu/publications/rfc96/.

Van Mieghem, P., 1998, "*A lower bound for the end-to-end delay in networks: Application to voice over IP*", Globecom'98, Nov. 8-12, Sydney (Australia), pp. 2508-2513.

Van Mieghem, P. and H. De Neve, 1999, "*Hop-by-hop Quality of Service Routing*", submitted to IEEE JSAC, October.

Van Mieghem, P., G. Hooghiemstra and R. van der Hofstad, 2000, "*A Scaling Law for the Hopcount*", Delft University of Technology, Information Technology and Systems, submitted to ACM SIGCOMM 2000, January 28.

Wang, Z. and J. Crowcroft, 1996, "*QoS Routing for supporting Multimedia Applications*", IEEE J. Select. Areas Commun., 14(7):1188-1234, September.

Widyono, R., 1994, "*The Design and Evaluation of Routing Algorithms for Real-Time Channels*", Technical Report ICSI Tr-94-024, International Computer Science Institute, U.C. Berkley, June.

Zhang, L., 1990, "*Virtual Clock: A new Traffic Control Algorithm for Packet Switching Networks*", ACM SIGCOMM '90, pp. 19-29, September.

Zhang, Z., C. Sanchez, B. Salkewicsz and E. Crawley, 1997, "*Quality of Service Extensions to OSPF or Quality of Service Path First Routing (QOSPF)*", draft-zhang-qos-ospf-01.