Precomputation of Constrained Widest Paths in Communication Networks

Stavroula Siachalou, Leonidas Georgiadis

Aristotle Univ. of Thessaloniki, Faculty of Engineering, School of Electrical and Computer Engineering, Telecommunications Dept. Thessaloniki, 54124, GREECE.
E-mails:ssiachal@auth.gr, leonid@auth.gr

Abstract. We consider the problem of precomputing constrained widest paths in a communication network. Precomputing and storing of all relevant paths minimizes the computational overhead required to determine an optimal path when a new connection request arrives. We present three algorithms that precompute paths with maximal bandwidth (widest paths), which in addition satisfy given end-to-end delay constraints. We analyze and compare the algorithms both in worst case and through simulations using a wide variety of networks.

Keywords: Precomputation, QoS Routing, Widest Paths, Bottleneck Paths, Graph Theory.

1 Introduction

In today's communication networks, transmission of multimedia traffic with varying performance requirements (bandwidth, end-to-end delay, packet loss, etc.), collectively known as Quality of Service (QoS) requirements, introduces many challenges. In such an environment, where a large number of new requests with widely varying QoS requirements arrive per unit of time, it is important to develop algorithms for the identification of paths that satisfy the QoS requirements (i.e. feasible paths) of a given connection request, with minimal computational overhead. Minimization of the computational overhead per request can be achieved by computing a priori (precomputing) and storing all relevant paths in a data base.

While a large number of studies addressed the Constrained Path Routing Problem (see [2], [4], [10], [12], [17] and the references therein) there are relatively few works dealing with the specific issues related to precomputing paths with QoS constraints [6], [8], [14]. In [8], the problem of precomputing optimal paths under hop-count constraints is investigated. They propose an algorithm that has superior performance than Bellman Ford's algorithm in terms of worst case bounds. In [14], by considering the hierarchical structure which is typical in large scale networks, an algorithm which offers substantial improvements in terms of computational complexity is presented. These studies concentrated on the hop-count path constraint.

In [9] Guerin, Orda and Williams presented the link available bandwidth metric as one of the information on which path selection may be based. They mentioned that the leftover minimum bandwidth on the path links after connection acceptance must be as large as possible in order to accept as many requests as possible. In this paper we focus on the problem of precomputing paths with maximal bandwidth (path bandwidth is the minimal of the path link bandwidths), which in addition must satisfy given end-to-end delay requirements which become known upon the arrival of a new request. We present three algorithms that provide all relevant paths. The first algorithm is an application in the specific context of the algorithm developed in [17] for the Constrained Path Routing Problem. The second is based on an implementation of the basic algorithmic steps in [17], where we introduce new data structures that take advantage of useful properties of the problem at hand. The third algorithm is based on an approach whereby iteratively relevant paths are determined and links that are not needed for further computation are eliminated. We analyze and compare the algorithms both in worst case and through simulations. The analysis considers both computation times and memory requirements and shows the trade-offs involved in the implementation of each of the algorithms.

The rest of the paper is organized as follows. The problem is formulated in Section 2. We present the three algorithms in Section 3 and in Section 4 we examine the algorithms in terms of worst case running time and memory requirements. Section 5 presents numerical experiments that evaluate the performance of the proposed algorithms. Conclusions of the work are presented in Section 6. Due to space limitation proofs are omitted. We refer the interested reader to the site [20] for a version containing proofs.

2 Model and Problem Formulation

In this section we formulate the problem related to the precomputation of constrained widest paths and define some notation that will be used in the rest of the paper.

A network is represented by a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of edges (links). Let $N = |V|$ and $M = |E|$. A link l with origin node u and destination node v is denoted by (u, v). A path is a sequence of nodes $p = (u_1, u_2, ..., u_k)$, such that $u_i \neq u_j$ for all $1 \leq i, j \leq k$, $i \neq j$, and $k-1$ is the number of hops of p. By p we also denote the set of links on the path, i.e., all links of the form (u_i, u_{i+1}), $i = 1, ..., k-1$. By $V_{in}(u)$ and $V_{out}(u)$ we denote respectively the set of incoming and outgoing neighbors to node u, that is

$$V_{in}(u) = \{v \in V : (v, u) \in E\}, \; V_{out}(u) = \{v \in V : (u, v) \in E\}.$$

With each link $l = (u, v)$, u, $v \in V$ there is an associated width $w_l \geq 0$ and a delay $\delta_l \geq 0$. We define the width and the delay of the path p respectively,

$$W(p) = \min_{l \in p} \{w_l\}, \; D(p) = \sum_{l \in p} \delta_l.$$

The set of all paths with origin node s, destination node u and delay less than or equal to d is denoted by $P_u(d)$. The set of all paths from s to u is denoted by P_u.

In a computer network environment, w_l may be interpreted as the free bandwidth on link l and δ_l as the link delay. Assume that a connection request has bandwidth requirements b and end-to-end delay requirement d. Upon the arrival of a new connection request with origin node s and destination node u, a path must be selected that joins the source to the destination, such that the connection bandwidth is smaller than the free bandwidth on each link on the path, and the end-to-end delay of connection packets is smaller than the path delay. It is often desirable to route the connection through the path with the largest width in $P_u(d)$; this ensures that the bandwidth requirements of the connection will be satisfied, if at all possible, and the delay guarantees will be provided. Moreover, the leftover minimum bandwidth on the path links after connection acceptance will be as large as possible. We call such a path "constrained widest path".

According to the previous discussion, upon the arrival of a new connection request with end-to-end delay requirement d, we must select a path $p^* \in P_u(d)$ that solves the following problem.

Problem I: Given a source node s, a destination node u and a delay requirement d, find a path $p_u^* \in P_u(d)$ that satisfies

$$W(p_u^*) \geq W(p) \text{ for all } p \in P_u(d).$$

Note that when $\delta_l = 1$ for all $l \in E$, Problem I reduces to the problem addressed in [8], i.e., the problem of finding a widest path with hop count at most d. Let us assume that the source node s is fixed. In principle, in order to be able to select the appropriate path for any delay requirement one must precompute for each destination u and each delay d, an appropriate optimal path $p_u^*(d)$. At first this may seem rather formidable, both in terms of running time and in terms of space requirements. However, the situation is greatly simplified by the observation that one needs to precompute the paths $p_u^*(d)$ for only a subset of the delays. Indeed, let $W_u^*(d)$ be the value of the solution to Problem I (if no solution exists set $W_u^*(d) = -\infty$). It can be easily seen using similar arguments as in [17] that $W_u^*(d)$ is a piecewise constant, left continuous, non-decreasing function with a finite number of discontinuities. Hence, to determine the function $W_u^*(d)$, we only need to know the values of $W_u^*(d)$ at these discontinuities (we also need the paths that cause these discontinuities - see Section 3.1). A discontinuity of $W_u^*(d)$ will also be referred to as a discontinuity of node u.

In fact, from the route designer's perspective, the pairs $(d_k, W_u^*(d_k))$, where d_k is a discontinuity point of $W_u^*(d)$ are the most interesting ones, even if one takes into account routing requirements different than those considered in Problem I. Specifically, under our interpretation of path width and delay, among pairs $(D(p_i), W(p_i))$, $p_i \in P_u$, $i = 1, 2$, there is a natural "preference relation". That is, we would like to obtain paths that have as small delay as possible and as large width as possible. We are thus lead to the following natural definition of dominance

Definition I (Dominance Relation): We say that pair $(D(p_1), W(p_1))$ dominates pair $(D(p_2), W(p_2))$ (or that path p_1 dominates path p_2) if either $\{W(p_1) > W(p_2)$ and $D(p_1) \leq D(p_2)\}$, or $\{W(p_1) \geq W(p_2)$ and $D(p_1) < D(p_2)\}$.

Hence, the pairs of interest under our setup are those for which no other dominating pair can be found for the same origin-destination nodes. This set of paths is generally known as the non-dominated or the Pareto-optimal set [3], [12]. From a precomputation perspective, it is desirable to determine for each destination u, the non-dominated set of pairs (and the associated paths). It can be shown that this set is exactly the set of discontinuities of $W_u^*(d)$, $u \in V$.

In the next section we present three algorithms for precomputing the discontinuities of the functions $W_u^*(d)$, $u \in V$.

3 Algorithm Description

The problem of determining the function discontinuities when link widths and delays are both additive costs (i.e., the cost of a path is the sum of its link costs) has been addressed in [17]. In the current setup, the main difference is that the path width is the minimum of its link widths (rather than the sum). However, the general algorithms in [17] can be adapted to the problem under consideration with minor modifications, as outlined in Section 3.1. In Sections 3.2 and 3.3 we present two additional algorithms that take into account the particular form of the problem under consideration. The first is an implementation of the algorithm in [17] that uses efficient data structures. The second uses a "natural" approach that eliminates successively unneeded graph edges and uses a dynamic version of Dijkstra's algorithm to determine all function discontinuities. Our intent is to compare these algorithms in terms of worst case, average running times and space requirements.

3.1 Algorithm I (ALG I)

The algorithms proposed in [17] are based on the following facts, which carry over to the situation at hand. In the discussion that follows we assume for convenience that $W_u^*(d)$ is defined for any real d, $W_u^*(d) = -\infty$, $d < 0$, and $W_s^*(d) = \infty$, $d \geq 0$. Hence by convention the source node s has a discontinuity at zero.

– For any $u \in V - \{s\}$, if $W_u^*(d)$ is discontinuous at d, then there is a $v \in V_{in}(u)$ such that $W_v^*(d)$ is discontinuous at $d - \delta_{vu}$ and $W_u^*(d) = \min\{W_v^*(d - \delta_{vu}),\ w_{vu}\}$. We call the pair $(d, W_u^*(d))$ the successor discontinuity of $(d-\delta_{vu}, W_v^*(d-\delta_{vu}))$. Also, $(d-\delta_{vu}, W_v^*(d-\delta_{vu}))$ is called the predecessor discontinuity of $(d, W_u^*(d))$. If it is known that the pair $(d, W_v^*(d))$ is a discontinuity point, then its "possible" successor discontinuities are pairs of the form
$$(d + \delta_{vu}, \min\{W_v^*(d), w_{vu}\}),\ u \in V_{out}(v).$$

– If $W_u^*(d)$ is discontinuous at d then there is a path $p^*(d) \in P_u(d)$ such that

$$W(p^*(d)) = W_u^*\ (d),\ D(p^*(d)) = d.$$

– Suppose that we impose a lexicographic order relation between discontinuity pairs $(\dot{d}_i, W_i)$, $i = 1, 2$, as follows:

$$(d_1, W_1) \prec (d_2, W_2) \text{ iff either } d_1 < d_2 \text{ or } (d_1 = d_2 \text{ and } W_1 > W_2).$$

Suppose also that among all the discontinuities of the functions $W_u^*(d)$, $u \in V$ we know the set of the k smallest ones (with respect to the lexicographic order). Call this set $\widehat{D}$. Let $\widehat{D}(u)$ be the discontinuities in $\widehat{D}$ that belong to node function $W_u^*(d)$. Hence $\widehat{D} = \cup_{u \in E} \widehat{D}(u)$. The set of possible successor discontinuities of those in $\widehat{D}$ is denoted by $\widehat{P}$. Let (d, W) be a smallest element of $\widehat{P}$ and let u be the node to which this possible discontinuity belongs. Then (d, W) is a real discontinuity for node u if and only if

$$W > \max\left\{W_m :\ (d_m, W_m) \in \widehat{D}(u)\right\}.$$

Based on these facts, we can construct an algorithm for determining all the node discontinuities as described below. In the following we will need to know the node u to which a real or possible discontinuity (d, W) belongs. For clarity we denote this discontinuity by (d, W, u). For initialization purposes we set $\widehat{D}(u) = \{(-\infty, -\infty, u)\}$, $u \in V$ and $\widehat{D}(s) = \{(0, \infty, s)\}$ .The generic algorithm is presented below.

Generic Algorithm I

Input: Graph G with link widths w_{uv} and delays δ_{uv}. **Output:** The queues $\widehat{D}(u)$, $\forall u \epsilon V$.

```
1. /* Begin Initialization
2. $\widehat{D}(u) = \{(-\infty, -\infty, u)\}$ ; $u \in V$, $\widehat{P} = \varnothing$;
3. $\widehat{D}(s) = \{(0, \infty, s)\}$ ; $(d, W, u) = (0, \infty, s)$;
4. /*End Initialization*/
5. Create all possible successor discontinuities of $(d, W, u)$
   (i.e., the set $\{(d + \delta_{uv}, \min\{W, w_{uv}\}, v),\ v \in V_{out}(u)\}$ and add them to $\widehat{P}$);
6. If $\widehat{P}$ is empty, then stop;
7. Among the elements $\widehat{P}$ (possible successor discontinuities), find and extract
   (i.e., delete from $\widehat{P}$) the minimum one in the lexicographic order. Denote
   this element $(d, W, u)$;
8. If $W \leq \max\left\{w_m : (d_m, W_m, u) \in \widehat{D}(u)\right\}$, then go to step 6. Else,
9. $\widehat{D}(u) \leftarrow \widehat{D}(u) \cup \{(d, W, u)\}$ ;
10. go to step 5;
```

In [17] two implementations of the generic algorithm were proposed, which differ mainly in the manner in which the set $\widehat{P}$ is organized. In the current work we pick the implementation that was shown to be more efficient both in worst case and average case analysis. For our purposes, it is important to note that the sets $\widehat{D}(u)$ are implemented as FIFO queues, and that the elements (d, W, u) in these queues are generated and stored in increasing order of both d and W as the algorithm proceeds. Furthermore, in our implementation of Algorithm I, we introduce an additional optimization that is based on the following observation in [8]: whenever a real discontinuity (d, W, u) is found and the possible discontinuities caused by (d, W, u) are created, then links (v, u), $v \in V_{in}(u)$ with $w_{vu} \leq W$ can be removed from further consideration. This is so, since these links cannot contribute to the creation of new discontinuities for node u. Indeed, any newfound discontinuity (d_1, W_1, v) at node v, will create a possible discontinuity $(d_1 + \delta_{vu},\ \min(W_1, w_{vu}),\ u)$. But $\min(W_1, w_{vu}) \leq W$ and hence this possible discontinuity cannot be a real one for node u.

As usual, in order to be able to find by the end of the algorithm not only the discontinuities, but paths that correspond to these discontinuities, one must keep track of predecessor discontinuities as well. That is, in the implementation we keep track of $(d, w, u, \mathit{predecessor\_disc})$, where for the source node s, $\mathit{predecessor\_disc} = \mathit{null}$, and for any other node u, $\mathit{predecessor\_disc}$ is a pointer to the predecessor discontinuity of (d, w, u). To simplify the notation, in the description of all algorithms we do not explicitly denote $\mathit{predecessor\_disc}$, unless it is needed for the discussion.

3.2 Algorithm II (ALG II)

The Generic Algorithm in Section 3.1 works also when lexicographic order is defined as

$$(d_1, W_1) \preccurlyeq (d_2, W_2) \text{ if either } W_1 > W_2 \text{ or } (W_1 = W_2 \text{ and } d_1 < d_2).$$

In this case, the elements (d, W, u) in the FIFO queues $\widehat{D}(u)$ are generated and stored in decreasing order of both d and W as the algorithm proceeds.

Algorithm II uses the lexicographic order $\preccurlyeq$, and is based on an extension of ideas presented in [7] to speedup computations. The basic observations are the following.

- Suppose that link widths take $K \leq M$ different values $g_1 < g_2 < \ldots < g_K$. Let $r(w_l)$ be the ranking order of w_l, i.e. if for link l it holds $w_l = g_i$, set $r(w_l) = i$. If one uses $r(w_l)$ instead of the link's actual width in the calculations, the resulting discontinuities occur at the same delays and for the same paths as if the actual widths were used.
- Path widths always take one of the values in the set $\{w_{vu}, (v, u) \in E\}$ i.e., they take at most K different values. Hence the same holds for the values of $W_u^*(d)$ and the widths of all possible discontinuities.

We use these observations to speed up the computations of Generic Algorithm I as follows. First, we use $r(w_l)$ in place of the link widths. Next we organize the set of possible discontinuities $\widehat{P}$ as follows. We create an array $A[u,k]$, $1 \leq u \leq N$, $1 \leq k \leq K$, where $A[u,k]$, if nonnull, denotes a possible discontinuity of the form (d,k,u). We also create K heaps $H[k]$, $1 \leq k \leq K$. Heap $H[k]$ contains the nonnull elements of $A[u,k]$, $1 \leq u \leq N$ and uses as key the delay d of a possible discontinuity. Reference [5] contains various descriptions of heap structures. For our purposes we need to know that the following operations can be performed on the elements of a heap structure.

- create_heap(H): creates an empty heap H.
- insert(e, H): inserts element e to H.
- get_min(e, H): removes and returns an element e in H with the smallest key.
- decrease_key(e_{new}, e, H): replaces in H element e with e_{new}, where element e_{new} has smaller key than e.

With these data structures, we implement steps 5 and 7 of Generic Algorithm I as follows. For an element $e = (d, W, u)$ we denote $e.delay = d$, $e.width = W$.

- **Step 5:** *Create all possible successor discontinuities of* (d, W, u) *and add them to* $\widehat{P}$.

  /* let $k' = r(W)$, hence we have available the discontinuity (d, k', u) */

  ```
  1. For $v \in V_{out}(u)$ do
     (a) $e_{new} = (d + \delta_{uv}, \min\{k', r(w_{uv})\}, v)$; $k = e_{new}.width$;
     (b) If $A[v,k]$ is null $\{A[v,k] = e_{new}$; insert($e_{new}, H[k]$)$\}$. Else {
     (c) If $e_{new}.delay < e.delay$ then{
         i. $e = A[v,k]$; $A[v,k] = e_{new}$;
         ii. decrease_key($e_{new}, e, H[k]$)}};
  2. end do
  ```

  In step 1b, if $A[v,k]$ is null, there is no possible discontinuity for node v with width k. Hence a new possible discontinuity for node v with width k is created and placed both in $A[v,k]$ and $H[k]$. In step 1c, when $e_{new}.delay < e.delay$ we know that the old possible discontinuity for node v cannot be a real discontinuity since e_{new} dominates e and therefore in step 1(c)i we replace the e with e_{new} both in $A[v,k]$ and $H[k]$. These last two steps avoid inserting unnecessary elements in the heap $H[k]$, thus decreasing the time that the get_min operation takes in step 7 of Generic Algorithm I. The trade-off is extra memory space requirements due to array $A[v,k]$. We discuss this issue further in Sections 4 and 5.
- **Step 7:** *Among the elements* $\widehat{P}$*, find and extract the minimum one in the lexicographic order. Denote this element* (d, W, u).

  /* let $k' = r(W)$, hence we have available the discontinuities (d, k', u) */

  The heaps $H[k]$ are scanned starting from the largest index and moving to the smallest. The index of the heap currently scanned is stored in the variable L which is initialized to K.

1. Find the largest $k' \leq L$ such that the heap $H[k']$ is nonempty;
2. get_min$(e, H[k'])$; $(d, k', u) = e$;
3. Set $A[u, k']$ to null;
4. $L = k'$;

The scanning process (largest to smallest) works since whenever a possible discontinuity (d, k, u) is removed from $\widehat{P}$, any possible discontinuities that already exist or might be added later to $\widehat{P}$ are larger (with respect to $\preccurlyeq$) than (d, k, u) and thus will have width at most k. Notice that this would not be true if the order $\prec$ were used. The real discontinuities $\widehat{D}(u)$, $u \in V$ are again implemented as FIFO queues.

3.3 Algorithm III (ALG III)

The third algorithm we consider is based on the idea of identifying discontinuities, eliminating links that are not needed to identify new discontinuities and repeating the process all over again. Specifically, the algorithm performs iterations of the basic steps shown below. Again $\widehat{D}(u)$, $u \in V$ are implemented as FIFO queues.

Algorithm III

Input: Graph G with link widths w_{uv} and delays δ_{uv}. **Output:** The queues $\widehat{D}(u)$, $\forall u \epsilon V$.

1. Find the widest-shortest paths from s to all nodes in G. That is, for any node $u \in V$, among the shortest-delay paths find one, say p_u, that has the largest width.
2. Let W^* be the minimum among the widths of the paths p_u, $u \in V - \{s\}$. For any $u \in V$, if $W(p_u) = W^*$, add $(D(p_u), W(p_u))$ at the end of queue $\widehat{D}(u)$.
3. Remove from G all links with width at most W^*.
4. If s has no outgoing links, stop. Else go to step 1.

This algorithm produces all discontinuities in $\widehat{D}$ as the next theorem shows.

Theorem 1. *Algorithm III produces all discontinuities in $\widehat{D}$.*

Proof. Proof can be found in [20].

The widest-shortest path problem can be solved by a modification of Dijkstra's algorithm [15]. In fact, after the removal of the links of G in step 3, paths whose width is larger than W^* will still remain the widest-shortest paths when the algorithm returns to step 1. Hence the computations in the latter step can be reduced by taking advantage of this observation. Algorithms that address this issue have been presented in [13] and we pick for our implementation the one that was shown to be the most efficient.

4 Worst Case Analysis

In this section we examine the three algorithms proposed in Section 3 in terms of worst case running time and memory requirements, the analysis of the algorithms is presented in [20]. In all three algorithms we assume a Fibonacci heap implementation [5]. In such implementation of a heap H, all operations except $\text{get\_min}(e, H)$ take $O(1)$ time. Operation $\text{get\_min}(e, H)$ takes $O(\log L)$ time, where L is the number of elements in the heap.

The worst case running times of the algorithms are,$O(MN \log N + M^2 \log N)$ for the first algorithm and $O(MN \log N + M^2)$ for ALG II and III. All three algorithms have the same worst case memory requirements equal to $O(NM)$. ALG II and ALG III have the same worst case running time, which is slightly better than the worst case running time of ALG I. Hence based on these metrics, all three algorithms have similar performance. However, worst case analysis alone is not a sufficient indicator of algorithm performance. The simulation results in Section 5 reveal the performance difference of the algorithms in several networks of interest.

5 Simulation Results

We run the following set of experiments. We generate:

Power Law Networks: This is one of the methods that attempt to generate network topologies that are "Internet like". We choose a number of N nodes and a number of M links ($M = \alpha N, \alpha > 1$). The links are used to connect nodes randomly with each other in such a manner that the node degrees follow a power law [18].

Real Internet Networks: These networks were taken from [19] and are based on network topologies observed on the dates 20/09/1998, 01/01/2000 and 01/02/2000.

For each experiment the delay of a link is picked randomly with uniform distribution among the integers $[1, 100]$. For the generation of the link widths we use the following method.

Widths are generated in such a manner that they are correlated to their delays. Thus, for each link l a parameter β_l is generated randomly among the integers $[1, 10]$. The width of link l will then be $w_l = \beta_l(101 - d_l)$.

We also run experiments using link widths uncorrelated to their delays, thus w_l is picked randomly with uniform distribution among the integers $[1, 100]$. For a given algorithm and for fixed number of nodes and edges we notice that the running time is much smaller when the width values are uncorrelated to delays and therefore are not presented here. This is due to the fact that when widths are correlated to delays, the number of discontinuities is increased.

We generate Power Law Networks with 400, 800 and 1200 nodes and with ratios $\alpha = M/N$ equal to 4, 8, 16. For each N and α we generate 10 different networks and for each network we generate the link widths according to the method previously described (correlated to delays).

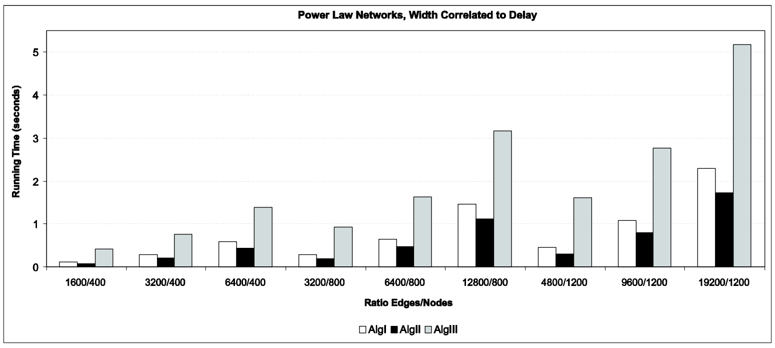

Fig. 1. Running Time for Power Law Networks with width correlated to delays

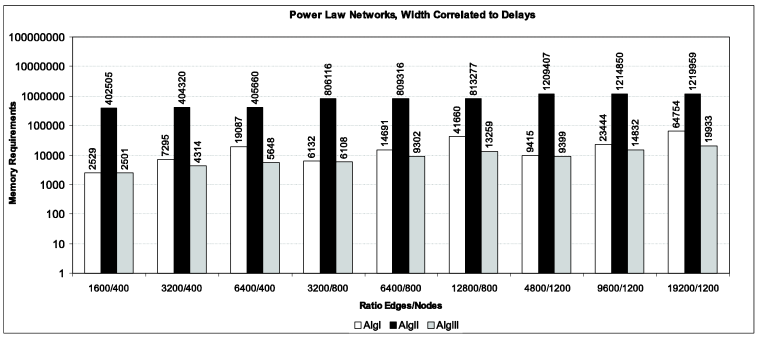

Fig. 2. Memory Requirements for Power Law Networks with width correlated to delays.

The experiments were run on a Pentium PC IV, 1.7GHz, 256MB RAM. In Figure 1 we present the average running times (in seconds) of the three algorithms for Power Law Networks. We make the following observations.

– Algorithm II has the best running time performance, and Algorithm III the worst.
– Compared to Algorithm II, the running times of Algorithm I and Algorithm III are found to be up to 1.5 times and 6 times larger, respectively.
– Algorithm II performs better than Algorithm I and III for all experiments and especially for large networks.

The additional optimization (removal of unneeded links) in Algorithm I improves its running time but not by much.

The Real Internet Networks have $N = 2107$, 4120, 6474 nodes and $M = 9360$, 16568, 27792 links respectively. In these networks we also performed 10 experiments, where in each experiment we picked randomly a source node. Figure 3(a) shows the average running time of the three algorithms. We notice again

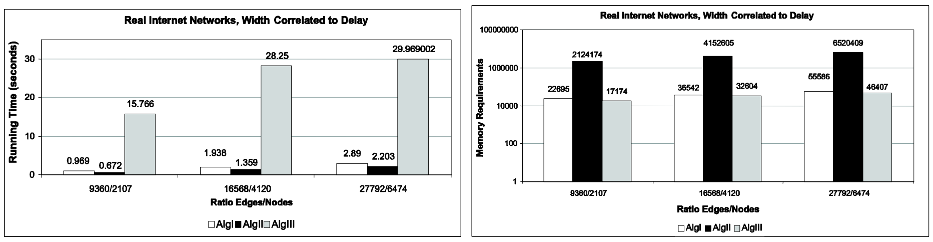

Fig. 3. (a)Running Time and (b)Memory Requirements for Real Internet Networks with width correlated to delays.

that Algorithm II has the best running time performance and Algorithm III the worst. The running time of Algorithm III has been found to be 20 times larger than that of Algorithm II in some experiments. The performance of Algorithm I is worse, but comparable to that of Algorithm II.

Next we look at the memory requirements of the algorithms. The memory space needed to store the network topology is common to all algorithms. The additional memory requirements of the three algorithms at any time during their execution, are determined mainly by the total number of elements in the queues $\widehat{D}(u)$, $u\epsilon V$ as well as: a) the heap size $\widehat{P}$ of possible discontinuities for Algorithm I, b) the heaps $H[k]$, $k \in K$ and the array $A[u,k]$, $1 \leq u \leq N$, $1 \leq k \leq K$ for Algorithm II and c) the heap size to run the dynamic version of Dijkstra's algorithm for Algorithm III. For each experiment we determined the maximum of memory space needed to store the previously mentioned quantities. This space depends on the particular network topology for Algorithm I and III, while for Algorithm II it is already of order $O(KN)$ due to the array $A[u,k]$. As a result, the memory requirements of Algorithm II are significantly larger than those of the other two algorithms. This is indicated in Figure 2-3(b) where we present the memory requirements of the three algorithms for Power Law and Real Internet Networks. Algorithm III has the smallest memory, followed by Algorithm I whose memory requirements are comparable to those of Algorithm III. Due to the need of array $A[u,k]$, Algorithm II has significantly larger memory requirements.

Summarizing our observations, Algorithm II has the best running time, however its memory requirements are significantly worse than those of the other two algorithms. At the other end, Algorithm III has the best memory space requirements, however its running time is significantly worse than that of the other two. Algorithm I represents a compromise between running time and space requirements, as its performance with respect to these measures, while not the best, is comparable to the best.

6 Conclusions

We presented three algorithms for precomputing constrained widest paths in a communication network. We analyzed the algorithms in terms of worst case running time and memory requirements. We also presented simulation results indicating the performance of the algorithms in networks of interest. The worst case analysis showed that all three algorithms have similar performance, with Algorithm I being slightly worse in case of worst case running time. However, the simulations revealed significant performance differences and indicated the conditions under which each algorithm is appropriate to be used.

References

1. Claude Berge, *Graphs*, North-Holland Mathematical Library, 1991.
2. D. Blokh, G. Gutin, "An Approximation Algorithm for Combinatorial Optimization Problems with Two Parameters", *IMADA* preprint PP-1995-14, May 1995.
3. K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*, Wiley, 2001.
4. S. Chen, K. Nahrstedt, "On Finding Multi-Constrained Paths", *in Proc. of IEEE International Conference on Communications (ICC'98),* pp. 874-879, Atlanta, GA, June 1998.
5. T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms,* Mc Graw Hill, 1990.
6. Yong Cui, Ke Xu, Jianping Wu, "Precomputation for Multi-Constrained QoS Routing in High Speed Networks", *IEEE INFOCOM 2003.*
7. L. Georgiadis, "Bottleneck Multicast Trees in Linear Time", to be published in *IEEE Communications Letters.*
8. R. Guerin, A. Orda, "Computing Shortest Paths for Any Number of Hops", *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, October 2002.
9. R. Guerin, A. Orda and Williams D., "QoS Routing Mechanisms and OSPF Extensions", 2nd IEEE Global Internet Mini-Conference, Phoenix, AZ, November 1997.
10. T. Korkmaz, M. Krunz and S. Tragoudas, "An Efficient Algorithm for Finding a Path Subject to Two Additive Constraints", *Computer Communications Journal*, vol. 25, no. 3, pp. 225-238, Feb. 2002.
11. K. Mehlhorn, S. Naher, *Leda:A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 2000.
12. P. Van Mieghem, H. De Neve and F.A. Kuipers, "Hop-by-hop Quality of Service Routing", *Computer Networks*, vol. 37/3-4, pp. 407-423, November 2001.
13. P. Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng, "New Dynamic Algorithms for Shortest Path Tree Computation ", *IEEE/ACM Transactions on Networking*, vol. 8, no. 6, December 2000.
14. A. Orda and A. Sprintson, "QoS Routing: The Precomputation Perspective", *IEEE INFOCOM 2000*, vol. 1, pp. 128-136, 2000.
15. J. L. Sobrino, "Algebra and Algorithms for QoS Path Computation and Hop-by-Hop Routing in the Internet", *IEEE INFOCOM 2001*, Anchorage, Alaska, April 22-26, 2001.
16. A. Orda and A. Sprintson, "A Scalable Approach to the Partition of QoS Requirements in Unicast and Multicast", *IEEE INFOCOM 2002.*
17. S. Siachalou, L. Georgiadis, "Efficient QoS Routing", *Computer Networks Journal,*vol.43/3, pp351-367, October 2003.
18. The Power Law Simulator, *http://*www.cs.bu.edu/brite.
19. The Real Networks, http://moat.nlanr.net/Routing/raw-data.
20. http://genesis.ee.auth.gr/georgiadis/english/public/networking04full.pdf