Algorithms for Precomputing Constrained Widest Paths and Multicast Trees

Stavroula Siachalou and Leonidas Georgiadis, *Senior Member, IEEE*

***Abstract*—We consider the problem of precomputing constrained widest paths and multicast trees in a communication network. Precomputing and storing of the relevant information minimizes the computational overhead required to determine an optimal path when a new connection request arrives. We evaluate algorithms that precompute paths with maximal bandwidth (widest paths), which in addition satisfy given end-to-end delay constraints. We analyze and compare both the worst case and average case performance of the algorithms. We also show how the precomputed paths can be used to provide computationally efficient solutions to the constrained widest multicast tree problem. In this problem, a multicast tree with maximal bandwidth (widest multicast tree) is sought, which in addition satisfies given end-to-end delay constraints for each path on the tree from the source to a multicast destination.**

***Index Terms*—Bottleneck paths, graph theory, multicast trees, precomputation, QoS routing, widest paths, widest trees.**

I. Introduction

IN TODAY'S communication networks, transmission of multimedia traffic with varying performance requirements (e.g., bandwidth, end-to-end delay, and packet loss), 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 overall computational overhead can be achieved by computing a priori (precomputing) and storing all relevant paths in a data base. The computation of all relevant paths is also an important intermediate step in the development of algorithms for other problems related to networking [12], [14], [16]. Hence, it is important to study and evaluate algorithms that specifically address the precomputation problem.

While a large number of studies addressed the Constrained Path Routing Problem (see [2], [3], [8], [11], [13], [15], [18], and [22] and the references therein), there are relatively few works addressing the specific issues related to precomputing paths with QoS constraints. In [9], the problem of precomputing optimal paths under hop-count constraints is investigated and an algorithm is proposed that has superior performance than Bellman Ford's algorithm in terms of worst case bounds. In [20], 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. The algorithms in [5], [18], and [22] can be used for precomputation under general QoS constraints. In [5], a subset of the "relevant" (see Section II for explanation) paths are precomputed. The algorithms in [18] and [22] are similar in spirit, however, the one in [22] avoids steps that would compute irrelevant paths and hence achieves better worst case performance.

In the first part of 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. We call these paths constrained widest paths. As mentioned in [10], in order to accept as many requests as possible, it is important to select paths with maximal bandwidth. This way, after a connection is accepted, the leftover minimum bandwidth on the path links will be as large as possible. We examine three algorithms that provide all relevant paths. The first algorithm is an application in the specific context of the algorithm developed in [22] for the general Constrained Path Routing Problem. The second is based on an implementation of the basic algorithmic steps in [22] using data structures that take advantage of properties of the problem at hand. The third algorithm improves on an approach whereby iteratively relevant paths are determined and links that are not needed for further computation are eliminated [12]. We analyze the worst case performance of the algorithms in terms of running time and space requirements. In addition, we compare the average case running time and space requirements of the algorithms through simulations. The analysis shows the tradeoffs involved in the implementation of each of the algorithms.

In the second part of this paper, we consider the constrained widest multicast tree problem. In this problem, a multicast tree is sought with maximal bandwidth (tree bandwidth is the minimal of the tree link bandwidths), which in addition satisfies given end-to-end delay constraints for each path on the tree from the source to a multicast destination. We show that, using the precomputed constrained widest paths, an algorithm can be developed that computes very efficiently the required tree. Hence, in effect, by precomputing the constrained widest paths, we also precompute all of the constrained widest multicast trees.

The remainder of this paper is organized as follows. The problem is formulated in Section II. We present the algorithms in Section III, and in Section IV we examine the algorithms in

Manuscript received July 18, 2003; revised May 7, 2004, and October 1, 2004; approved by IEEE/ACM Transactions on Networking Editor R. Cohen. This paper was presented in part at the Networking Conference, May 2004, Athens, Greece.

The authors are with the School of School of Electrical and Computer Engineering Department, Telecommunications Department, Aristotle University of Thessaloniki, Thessaloniki, 54124 Greece (e-mail: stavroula@psyche.ee.auth.gr; leonid@auth.gr).

Digital Object Identifier 10.1109/TNET.2005.857117

terms of worst case running time and memory requirements. In Section V, we show how the precomputed paths can be used to provide efficient computation of the constrained widest multicast tree problem. Section VI presents numerical experiments that evaluate the performance of the proposed algorithms. Conclusions are presented in Section VII.

II. 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 $p = (u_1, u_2, \ldots, u_k)$, 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 $(u_i, u_{i+1}), i = 1, \ldots, k-1$. By $V_{\text{in}}(u)$ and $V_{\text{out}}(u)$, we denote, respectively, the set of incoming and outgoing neighbors to node u, that is

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

respectively.

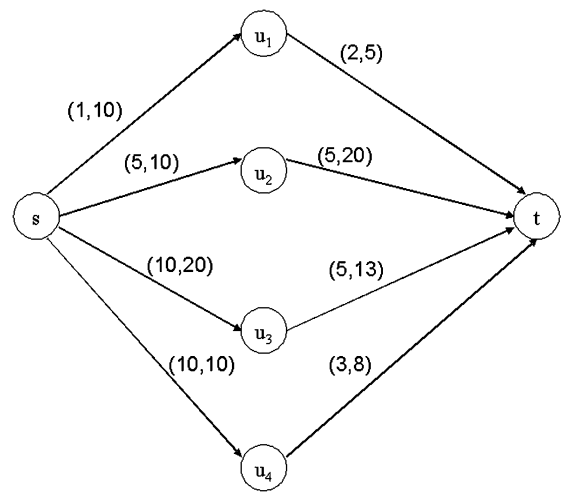
With each link $l = (u, v), u, v \in V$, there is an associated width $w_l \geq 0$ and a delay $\delta_l \geq 0$. We define the width and the delay of the path p, respectively, as

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

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 the "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 $p_u^* \in P_u(d)$ that solves the following problem.



Path	Delay	Width
$p_1 = (s, u_1, t)$	3	5
$p_2 = (s, u_2, t)$	10	10
$p_3 = (s, u_3, t)$	15	13
$p_4 = (s, u_4, t)$	13	8

Fig. 1. Example network.

Problem I: Given the 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 [9], 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 [12], [16], [22]. 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 [22] that $W_u^*(d)$ is a piecewise-constant, right continuous, nondecreasing function with a finite number of discontinuities. Consider for example the network in Fig. 1, where the pair (d, w) above each link denotes the link delay and width, respectively. As shown in this figure, in this example, there are four paths from node s to t. Hence, we have

$$P_t(d) = \begin{cases} \varnothing, & d < 3 \\ \{p_1\}, & 3 \leq d < 10 \\ \{p_1, p_2\}, & 10 \leq d < 13 \\ \{p_1, p_2, p_4\}, & 13 \leq d < 15 \\ \{p_1, p_2, p_3, p_4\}, & 15 \leq d \end{cases}.$$

Based on this, we can plot the graph of $W_t^*(d)$ in Fig. 2. The points A, B, C, D correspond to paths p_1, p_2, p_3, p_4, respectively. Notice that for $13 \leq d < 15$ the optimal width corresponds to the width of path p_2, which has smaller delay and larger width than path p_4. Hence, there is no discontinuity

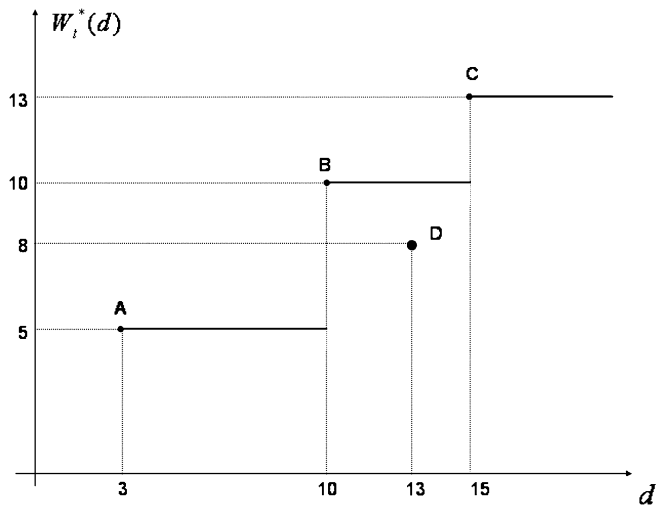


Fig. 2. Plot of $W_t^*(d)$ for the network in Fig. 1.

at point D, even though the path p_4 is added to the set $P_t(d)$ at $d = 13$.

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 III-A). A discontinuity of $W_u^*(d)$ will also be referred to as a discontinuity of node u.

In fact, from the routing 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—in Section V we present such a situation. Specifically, under our interpretation of path delay and width, 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 led 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 in 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 nondominated or the Pareto-optimal set [6], [18]. From a precomputation perspective, it is desirable to determine for each destination u, the set of nondominated pairs (and the associated paths). It can be shown that this set is exactly the set of discontinuities of $W_u^*(d), u \in V$.

III. Algorithm Description

In [22], the problem of determining the function discontinuities for multiple additive link costs (i.e., cost of a path is the sum of its link costs) has been addressed. 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 [22] can be adapted to the problem under consideration with minor modifications, as outlined in Section III-A. In Sections III-B and III-C, we present two additional algorithms that take into account the particular structure of the problem under consideration. The first is an implementation of the algorithm in [22] 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 and average running times and space requirements.

A. Algorithm I (ALG I)

The algorithms proposed in [22] 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 $v \in V_{\text{in}}(u)$ 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
$$(d + \delta_{vu}, \min\{W_v^*(d), w_{vu}\}), \quad u \in V_{\text{out}}(v).$$
- 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 pairs $(d_i, W_i), i = 1, 2$, as follows. We say that (d_2, W_2) is larger than (d_1, W_1) in the lexicographic order, $(d_1, W_1) \prec (d_2, W_2)$, if and only if (iff) either $d_1 < d_2$ or ($d_1 = d_2$ 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 $\hat{D}$. Let $\hat{D}(u)$ be the discontinuities in $\hat{D}$ that belong to node function $W_u^*(d)$. Hence, $\hat{D} = \cup_{u \in V} \hat{D}(u)$. The set of possible successor discontinuities of those in $\hat{D}$, that are larger than or equal to the discontinuities in $\hat{D}$, is denoted by $\hat{P}$. Let (d, W) be a smallest element of $\hat{P}$ and let u be the node to which this possible discontinuity belongs. Then (d, W) is a real discontinuity for node u, and the $k+1$-smallest among all the node discontinuities of the functions $W_u^*(d), u \in V$, iff
$$W > \max\{W_m : (d_m, W_m) \in \hat{D}(u)\}. \tag{1}$$

  The proof of correctness of the facts above is analogous to arguments used in [22, Lemma 2 and arguments in Section 3.1.1]. We outline the arguments leading to (1), since it is essential for understanding the algorithm. Let (d_l, W_l) be

TABLE I
GENERIC ALGORITHM FOR SOLVING PROBLEM I

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) $\widehat{D}(u) = \{(-\infty, -\infty, u)\}$, $u \in V - \{s\}$; $\widehat{P} = \varnothing$;
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;
9) **Else**
10) $\widehat{D}(u) \leftarrow \widehat{D}(u) \cup \{(d, W, u)\}$; go to step 5;

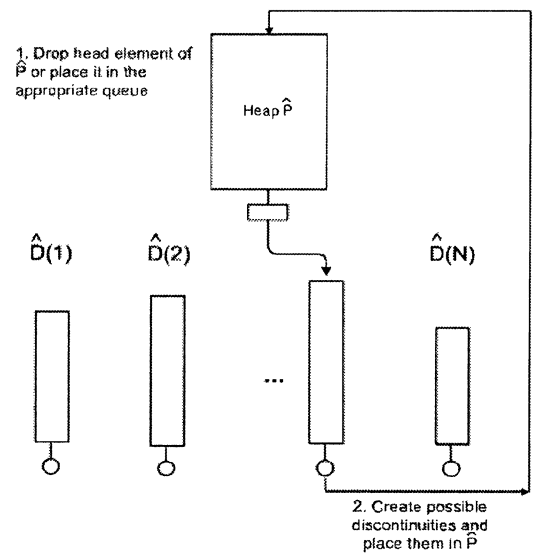


Fig. 3. Basic steps of Algorithm I.

the discontinuity in $\hat{D}(u)$ with the largest delay—hence, also with the largest width since $W_u^*(d)$ is nondecreasing. Since (d, W) is larger than (d_l, W_l) in the lexicographic order, it must hold, $d \geq d_l$. If $W \leq W_l$, then (d_l, W_l) dominates or is equal to (d, W). Hence, (d, W) cannot be a new real discontinuity. Assume next that $W > W_l$. Then necessarily $d > d_l$, since otherwise (d, W) dominates (d_l, W_l), which contradicts the fact that (d_l, W_l) is a discontinuity of $W_u^*(d)$. Using the fact that every discontinuity of a node $u \neq s$ has a predecessor discontinuity, it can be shown that, if there is a node discontinuity not in $\hat{D}$, smaller than (d, W) in the lexicographic order, then there must also be a discontinuity (d_k, W_k) in $\hat{D}$ whose successor is larger than the discontinuities in $\hat{D}$ (hence (d_k, W_k) belongs to $\hat{P}$) and smaller than (d, W). However, this contradicts the fact that (d, W) is the smallest element of $\hat{P}$. Hence, there is no discontinuity in $W_u^*(d)$ larger than (d_l, W_l) and smaller than (d, W), and since (d, W) is not dominated by any of the discontinuities in $\hat{D}(u)$, it must itself be a discontinuity.

Based on these facts, we can construct an algorithm for determining all 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 $\hat{D}(u) = \{(-\infty, -\infty, u)\}, u \in V - \{s\}$ and $\hat{D}(s) = \{(0, \infty, s)\}$. The generic algorithm is presented in Table I. Steps 1–4 of Algorithm I initialize the discontinuities pairs for each node. In step 5, having identified a real discontinuity (d, W, u), the algorithm creates the set of all possible discontinuities that have (d, W, u) as a predecessor. If there are no possible discontinuities (step 6), the algorithm stops. Else, in step 7, the smallest possible discontinuity in the lexicographic order is extracted from $\hat{P}$. In step 8, it is examined whether the latter discontinuity is a real one. If it is, it is added to the appropriate set (step 10); else the next possible discontinuity is extracted from $\hat{P}$, i.e., the algorithm moves to step 6. A schematic diagram of the operation, taken from [22], is given in Fig. 3.

In [22], two implementations of the generic algorithm were proposed, which differ mainly in the manner in which the set $\hat{P}$ 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 $\hat{D}(u)$ 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 [9]: whenever a real discontinuity (d, W, u) is found and the possible discontinuities caused by (d, W, u) are created, then links $(v, u), v \in V_{\text{in}}(u)$ 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 $(d_1 + \delta_{vu}, \min(W_1, w_{vu}), u)$. However, $\min(W_1, w_{vu}) \leq W$, 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 $(d, w, u, \text{predecessor\_disc})$, where for the source node $s, \text{predecessor\_disc} = \text{null}$, and for any other node $u, \text{predecessor\_disc}$ 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 predecessor_disc, unless it is needed for the discussion.

B. Algorithm II (ALG II)

The generic algorithm in Table I works also when lexicographic order is defined as

$$(d_1, W_1) \preccurlyeq (d_2, W_2)$$

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 $\hat{D}(u)$ 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 $g_1 < g_2 < \cdots < g_K$. 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. This is true since the steps in Algorithm I use only comparisons between link widths; hence, the same decisions are taken if $r(w_l)$ is used in place of w_l.
- Since a path width is the minimum of the widths of the path links, 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 $\hat{P}$ 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). Note that we are able to use the width of a discontinuity as an index to $A[u, k]$, since we are working with $r(w_l)$ instead of w_l. Had we used w_l directly, even if $w_l, l \in E$ were integers, we would in general need more space for $A[u, k]$, since the range of the values of w_l would be larger than K. 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 a key the delay d of a possible discontinuity. Reference [4] 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.

- $\text{create\_heap}(H)$: creates an empty heap H.
- $\text{insert}(e, H)$: inserts element e to H.
- $\text{get\_min}(e, H)$: removes and returns an element e in H with the smallest key.
- $\text{decrease\_key}(e_{\text{new}}, e, H)$: replaces in H element e with e_{new}, where element e_{new} has a smaller key than e.

Fig. 4 shows the data structures used to implement the set $\hat{P}$. With these data structures, we implement steps 5 and 7 of Generic Algorithm I in Table I as follows. For an element $e = (d, W, u)$, we denote $e.\text{delay} = d, e.\text{width} = W$.

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

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

1) **For** $v \in V_{\text{out}}(u)$ **do**
2) $e_{\text{new}} = (d + \delta_{uv}, \min\{k', r(w_{uv})\}, v)$;
3) $k = e_{\text{new}}.\text{width}$;
4) **If** $A[v, k]$ is null **then**
5) $A[v, k] = e_{\text{new}}$; $\text{insert}(e_{\text{new}}, H[k])$;
6) **Else**

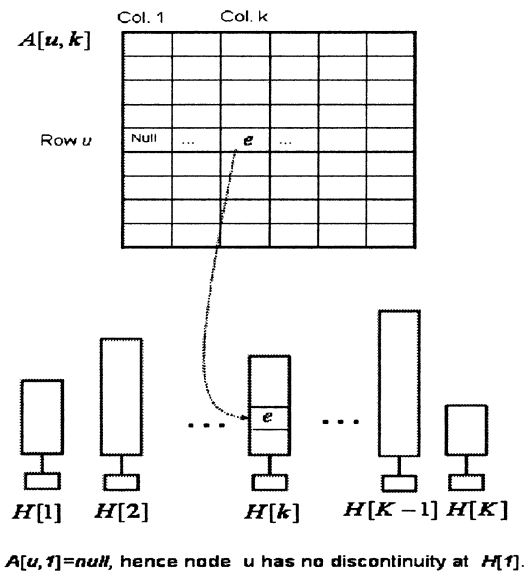


Fig. 4. Structures implementing the set of possible discontinuities $\hat{P}$.

7) $e = A[v, k]$;
8) **If** $e_{\text{new}}.\text{delay} < e.\text{delay}$ **then**
9) $A[v, k] = e_{\text{new}}$;
10) $\text{decrease\_key}(e_{\text{new}}, e, H[k])$;
11) **end do**

In step 4, 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 8, when $e_{\text{new}}.\text{delay} < e.\text{delay}$ we know that the old possible discontinuity for node v cannot be a real discontinuity since e_{new} dominates e and therefore in steps 9 and 10, we replace e with e_{new} both in $A[v, k]$ and $H[k]$. Note that, in this way, heap $H[k]$ always contains at most one possible discontinuity of any node v. Hence, with steps 9 and 10, we 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 in Table I. The tradeoff is extra memory space requirements due to array $A[v, k]$. We discuss this issue further in Sections IV and VI.

Step 7: *Among the elements* $\hat{P}$, find and extract the minimum one in the lexicographic order (with respect to $\preccurlyeq$). 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 $H[k']$ is nonempty.
2) $\text{get\_min}(e, H[k']); (d, k', u) = e$.
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 $\hat{P}$, any possible discontinuities that already exist or might be added

TABLE II
PSEUDOCODE OF ALGORITHM II

Algorithm II
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) $\widehat{D}(u) = \{(-\infty, -\infty, u)\}$, $u \in V - \{s\}$; $\widehat{P} = \emptyset$;
3) $\widehat{D}(s) = \{(0, \infty, s)\}$; $(d, k', u) = (0, \infty, s)$;
4) Determine $r(w_l)$ and K;
5) Create $A[v, k]$, $H[k]$ and initialize to null;
6) **For** $v \in V_{out}(s)$ **do**
7) $\quad e_{new} = (d + \delta_{sv}, \min\{k', r(w_{sv})\}, v)$;
8) $\quad k = e_{new}.width$;
9) $\quad A[v, k] = e_{new}$; insert($e_{new}, H[k]$);
10) **end do**
11) Determine the largest index k' such that $H[k']$ is nonempty;
12) get_min($e, H[k']$),$(d, k', u) = e$;
13) Set $A[u, k']$ to null;
14) $L = k'$;
15) /*End Initialization */
16) **For** $v \in V_{out}(u)$ **do**
17) $\quad e_{new} = (d + \delta_{uv}, \min\{k', r(w_{uv})\}, v)$;
18) $\quad k = e_{new}.width$;
19) $\quad$ **If** $A[v, k]$ is null **then**
20) $\quad\quad A[v, k] = e_{new}$; insert($e_{new}, H[k]$);
21) $\quad$ **Else**
22) $\quad\quad e = A[v, k]$;
23) $\quad\quad$ **If** $e_{new}.delay < e.delay$ **then**
24) $\quad\quad\quad A[v, k] = e_{new}$;
25) $\quad\quad\quad$ decrease_key($e_{new}, e, H[k]$);
26) **end do**
27) **If** $L = 1$ and $H[1]$ is empty **then** stop;
28) **Else**
29) $\quad$ Find the largest $k' \leq L$ such that $H[k']$ is nonempty;
30) $\quad$ get_min($e, H[k']$); $(d, k', u) = e$;
31) $\quad$ Set $A[u, k']$ to null;
32) $\quad L = k'$;
33) $\quad$ **If** $d \geq d_m$ where (d_m, k_m, u) is the last element of $\widehat{D}(u)$ **then**
34) $\quad\quad$ go to step 29;
35) $\quad$ **Else**
36) $\quad\quad$ add e to $\widehat{D}(u)$; go to step 16;

later to $\hat{P}$ are larger (with respect to $\preccurlyeq$) than or equal to (d, k, u) and thus will have width at most k; hence, all of these discontinuities are or will be located in one of the heaps $H[l], l \leq k$. Notice that this would not be true if the order $\prec$ were used. Indeed, had we used order $\prec$, we would know that any possible discontinuities that already exist or might be added later to $\hat{P}$ have a delay of at least d, but we would not be able to say whether the widths of these discontinuities are larger or smaller than k. Table II presents the pseudocode for Algorithm II. Steps 11–15 are used to initialize the scanning process. The real discontinuities $\hat{D}(u), u \in V$ are again implemented as FIFO queues.

Summarizing, Algorithm II takes advantage of the properties of path width to create a more efficient organization of the set of possible discontinuities $\hat{P}$; otherwise, it takes the same decisions as Algorithm I. Hence, the correctness of Algorithm II follows from the correctness of Algorithm I. It is worth noting that, if the widths $w_l, l \in E$ take integer values and the range of these values is, say $[a, b]$, then one can use $k = w_l - a + 1$ as an index for placing the discontinuities in the structures $A[v, k]$ and $H[k]$. In this case, step 4 of Algorithm II in Table II is not needed and computations can be saved, provided that $b - a$ is small compared to M. In any case, step 4 is not the major source of computations.

TABLE III
ITERATION STEPS OF ALGORITHM III

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** $W(p_u) = W^*$ **then** 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 **then** stop; **Else** go to step 1;

C. Algorithm III (ALG III)

The third algorithm is based on an improvement of the idea proposed in [12]. It consists of a) identifying discontinuities, b) eliminating links that are not needed to identify new discontinuities and c) repeating the process all over again. Specifically, the algorithm performs iterations of the basic steps shown in Table III. Again $\hat{D}(u), u \in V$ are implemented as FIFO queues.

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

Theorem 1: Algorithm III produces all discontinuities in $\hat{D}$.

Proof: We will show by induction that, at iteration m, all discontinuities in $\hat{D}$ with width at most W_m^* (defined in step 2 of Algorithm III in Table III) have been determined.

Assume this to be true up to iteration m (the arguments for $m = 1$ are similar). For any node $u \in V$, any other real discontinuity will necessarily have larger width than W_m^*. Hence, the removal of links with width at most W_m^* in step 2 cannot result in elimination of a path p_u that causes real discontinuities.

We claim that, at iteration $m + 1$, the pairs $(D(p_u), W(p_u))$ added to $\hat{D}(u)$ at step 2 are the real discontinuities having width W_{m+1}^*. Indeed, observe first that $(D(p_u), W(p_u))$ cannot be dominated by any $(D(q_u), W(q_u)), q_u \in P_u$, with $W(q_u) \leq W_m^*$, since $W(p_u) = W_{m+1}^* > W_m^*$. We show next that $(D(p_u), W(p_u))$ is not dominated by any $(D(q_u), W(q_u))$ with $W(q_u) > W_m^*$. Assume the contrary. Notice first that the graph G in step $m + 1$ contains path q_u because $W(q_u) > W_m^*$ and hence all links of q_u belong to G in step $m + 1$. Since p_u is a shortest path in G at step $m + 1$, it is impossible that $D(q_u) < D(p_u)$. Hence, for domination, we must have

$$W(q_u) > W(p_u)$$
$$D(q_u) = D(p_u).$$

However, this latter condition is impossible also, since, according to step 1, p_u is a widest–shortest path in a graph G at step $m + 1$. ■

The widest–shortest path problem can be solved by a modification of Dijkstra's algorithm [23], without increasing the worst case complexity. 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 [19], and we pick for our implementation the one that was shown to be the most efficient, namely the static Heap Dijkstra algorithm.

Algorithms similar to Algorithm III have been presented in [12, p. 308] and [16, p. 20]. The main difference of the present algorithm is the use of the widest–shortest path algorithm to determine new discontinuities. As seen from the Proof of Theorem 1, at each round of the algorithm new real discontinuities are found, while the algorithms in [12] and [16] may waste rounds determining paths that do not correspond to real discontinuities. The algorithm in [12] determined the discontinuities of a specific node and can be extended to determine the discontinuities of all nodes. The algorithm in [16] adds links rather than removing them. This choice may have significant performance implications when the algorithms are used in combination with the algorithms in [19] to improve average case performance. This is due to the fact that the algorithm in [19] needs to determine a set of nodes, U, whose shortest delay will not change by the removal or addition of network links. As mentioned in the previous paragraph, when links are removed, set U is immediately determined. However, when links are added, further processing is required to determine U. In Section VI, we discuss further the performance implications of these alternatives.

IV. Worst Case Analysis

In this section, we examine the three algorithms proposed in Section III in terms of worst case running time and memory requirements. Let $R(u)$ be the number of discontinuities of $W_u^*(d)$ and denote $R_{\max} = \max_{u \in V}\{R(u)\}$. Notice that, since $W_u^*(d)$ takes only the values $w_l, l \in E$, we have $R_{\max} \leq M$. Let also $N_{\max}^{\text{in}}$ be the maximum in-degree of the nodes in V, that is,

$$N_{\max}^{\text{in}} = \max_{u \in V}\{|V_{\text{in}}(u)|\}.$$

Clearly, $N_{\max}^{\text{in}} \leq N$. In all three algorithms, we assume a Fibonacci heap implementation [4]. 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 $O(\log |H|)$ time, where $|H|$ is the number of elements in the heap.

Algorithm I

The analysis of this algorithm has been presented in [22]. According to this analysis the following hold.

Running Time: The worst case running time of the algorithm is, $O(R_{\max}(N \log N + M \log N_{\max}^{\text{in}})$ and in terms of the network size, $O(MN \log N + M^2 \log N)$.

Memory Requirements: The memory requirements of the algorithm are $O(MR_{\max})$ or $O(MN)$.

Algorithm II

Running Time: The process of determining $r(w_l), l \in E$ (step 4 of Algorithm II) amounts to sorting the elements of E. Hence, with a comparison based sorting, this process takes $O(M \log M) = O(M \log N)$ time. Each of the heaps $H[k]$ contains at most N elements and the get_min operation is applied at most once to each element. Hence, the computation time to process the get_min operations on heap $H[k]$ is at most $O(N \log N)$. The total computation time to process the possible discontinuities at the outgoing neighbors of all nodes in heap $H[k]$ (lines 16–26 of Algorithm II in Table II), is $O(M)$. Since all the rest of the operations during the processing of the elements of heap $H[k]$ take time $O(N)$, the computation time to process the kth heap is $O(N \log N + M)$. Since there are at most M heaps, the total worst case computation time is $O(MN \log N + M^2)$, including the time needed to sort $r(w_l), l \in E$.

Memory Requirements: The size of $A[v, k]$ is NM. Each of the $H[k]$ heaps contains at most N elements, and, since there are at most M such heaps, the total heap memory space needed is $O(NM)$. Since each of the queues $\hat{D}(u), u \in V$ can contain up to M discontinuities, the memory space needed to store the discontinuities is at most $O(NM)$. Therefore, the total memory space requirements are $O(NM)$.

Algorithm III

Running Time: At each iteration of steps 1 to 4 of Algorithm III in Table III, the dynamic version of the Dijkstra Algorithm [19] is used to find the widest–shortest paths. While this dynamic version reduces significantly the average running time of the algorithm, it does not reduce its worst case running time. Hence, the worst-case time bound for step 1 of the algorithm at each iteration is $O(N \log N + M)$. The rest of the operations at each iteration are of smaller order than $O(N \log N + M)$. Since there can be at most M iterations, the total worst case running time of the algorithm is $O(MN \log N + M^2)$.

Memory Requirements: This algorithm needs a single heap of size at most N. It also needs $O(NM)$ space to hold the real discontinuities. Hence, its space requirements are in the worst case, $O(NM)$.

TABLE IV
Worst Case Running Times and Memory Requirements

	Running Times	Memory
ALG I	$O(MN \log N + M^2 \log N)$	$O(MN)$
ALG II	$O(MN \log N + M^2)$	$O(MN)$
ALG III	$O(MN \log N + M^2)$	$O(MN)$

Table IV summarizes the worst case running time and space requirements of the proposed algorithms. All three algorithms have the same worst case memory requirements. 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. For example, as discussed above, the performance of ALG I depends on $R_{\max}$, which in many networks is much smaller than M. Since ALG II is based on ALG I, a detailed analysis of ALG II reveals that its performance depends on $R_{\max}$ as well. Regarding ALG III, the number of iterations of the basic algorithmic steps may be significantly smaller than M. As for memory requirements, ALG II has in general the most requirements due to the array $A[v, k]$. ALG III has the least requirements which are dominated by the necessary space to hold all node discontinuities. The simulation results in Section VI will reveal the performance difference of the algorithms in several networks of interest.

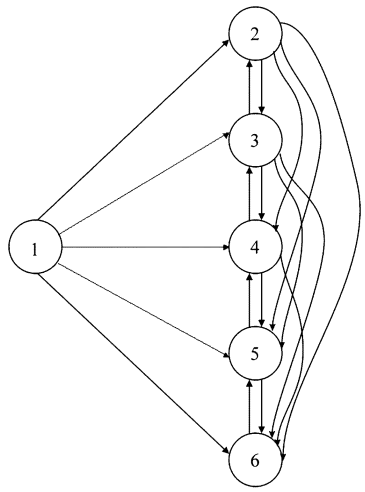


Fig. 5. Example of a network with $\Omega(N^3)$ memory requirements.

Since M can be of order $O(N^2)$, the worst case space requirements of all three algorithms is $O(N^3)$. For this to happen, most of the functions $W_u^*(d)$ should have a large number of discontinuities, and it is not immediately clear whether this can happen. Next we present an instance where indeed the memory requirements are $\Omega(N^3)$.

Consider a network consisting of n nodes and the following edges:

$$\begin{aligned}&(1,i), \qquad i=2,\dots,n\\ &(i+1,i), \qquad i=2,\dots,n-1\\ &(i,j), \qquad i=2,\dots,n-1, j=i+1,\dots,n.\end{aligned}$$

Fig. 5 shows an example of such an network with $n=6$. The number of edges in this network is

$$n-1+n-2+\sum_{i=2}^{n-1}(n-i)=\frac{1}{2}n^2+\frac{1}{2}n-2.$$

Set

$$\begin{aligned}w_{i+1,i} &= \infty\\ \delta_{i+1,i} &= 0, \qquad i=2,\dots,n-1.\end{aligned} \tag{2}$$

Pick the rest of the widths and delays so that the following inequalities hold:

$$\begin{aligned}&w_{1,i} > w_{1,i+1},\\ &\delta_{1,i} > \delta_{1,i+1}, \qquad i=2,\dots n-1\end{aligned} \tag{3}$$

$$\begin{aligned}&w_{1,2} > w_{2,j} > w_{2,j+1} > w_{1,3}\\ &\delta_{1,2} > \delta_{2,j} > \delta_{2,j+1} > \delta_{1,3}, \qquad j=3,\dots,n\end{aligned} \tag{4}$$

$$\begin{aligned}&w_{1,3} > w_{3,j} > w_{3,j+1} > w_{1,4}\\ &\delta_{1,3} > \delta_{3,j} > \delta_{2,j+1} > \delta_{1,4}, \qquad j=4,\dots,n\end{aligned} \tag{5}$$

and in general

$$\begin{aligned}&w_{1,i} > w_{i,j} > w_{i,j+1} > w_{1,i+1}\\ &\delta_{1,i} > \delta_{i,j} > \delta_{i,j+1} > \delta_{1,i+1}\\ &\qquad i=2,\dots,n-1, j=i+1,\dots,n.\end{aligned} \tag{6}$$

An example of widths and delays satisfying relations (2)–(6) is the following:

$$\begin{aligned}\delta_{1,i} &= w_{1,i} = (n+1)(n-i)\\ \delta_{i,j} &= w_{i,j} = (n+1)(n-i)-j,\\ &\qquad i=2,\dots,n-1,\ j=i+1,\dots,n.\end{aligned}$$

Under these conditions, each node has the following paths that cause discontinuities.

Node 2: $n-1$ paths of the form $(1,2),(1,3,2),\dots(1,i,i-1,\dots,3,2)$ because of relations (2) and (3).

Node 3: $2(n-2)$ paths; $n-2$ paths of the form $(1,2,3),(1,2,4,3),(1,2,i,i-1,\dots,4,3)$ because of relations (2) and (4) and $n-2$ paths of the form $(1,3),(1,4,3),\dots,(1,i,i-1,\dots,4,3)$ because of relations (2) and (3).

Node 4: $3(n-3)$ paths; $n-3$ paths of the form $(1,2,4),(1,2,5,4),(1,2,i,i-1,\dots,5,4)$ because of relations (2) and (4), $n-3$ paths of the form $(1,3,4),(1,3,5,4)\dots(1,3,i,i-1,\dots 4)$ because of relations (2) and (5), and $n-3$ paths of the form $(1,4),(1,5,4),\dots,(1,i,i-1,\dots 4)$ because of relations (2) and (3).

In general, it can be seen that node $k+1, k=1,\dots,n-1$ has $k(n-k)$ discontinuities, and hence the total number of discontinuities is

$$\sum_{k=1}^{n-1}k(n-k)=\frac{1}{6}n^3-\frac{1}{6}n=\Omega(n^3).$$

Example 2: Consider the example network in Fig. 5. The nodes have the following paths that cause discontinuities.

Node 2:

1) $p_i=(1,i,i-1,\dots,2), D(p_i)=\delta_{1,i}, W(p_i)=w_{1,i}, i=2,3,4,5,6.$

Node 3:

1) $p_i=(1,2,i,i-1,\dots,3), D(p_i)=\delta_{1,2}+\delta_{2,i}, W(p_i)=w_{2,i}, i=3,4,5,6.$
2) $p_i=(1,i,i-1,\dots,3), D(p_i)=\delta_{1,i}, W(p_i)=w_{1,i}, i=3,4,5,6.$

Node 4:

1) $p_i=(1,2,i,i-1,\dots,4), D(p_i)=\delta_{1,2}+\delta_{2,i}, W(p_i)=w_{2,i}, i=4,5,6.$
2) $p_i=(1,3,i,i-1,\dots,4), D(p_i)=\delta_{1,3}+\delta_{3,i}, W(p_i)=w_{3,i}, i=4,5,6.$
3) $p_i=(1,i,i-1,\dots,4), D(p_i)=\delta_{1,i}, W(p_i)=w_{1,i}, i=4,5,6.$

Node 5:

1) $p_i=(1,2,i,i-1,\dots,5), D(p_i)=\delta_{1,2}+\delta_{2,i}, W(p_i)=w_{2,i}, i=5,6.$
2) $p_i=(1,3,i,i-1,\dots,5), D(p_i)=\delta_{1,3}+\delta_{3,i}, W(p_i)=w_{3,i}, i=5,6.$
3) $p_i=(1,4,i,i-1,\dots,5), D(p_i)=\delta_{1,4}+\delta_{4,i}, W(p_i)=w_{4,i}, i=5,6.$

4) $p_i = (1, i, i-1, \ldots, 5), D(p_i) = \delta_{1,i}, W(p_i) = w_{1,i}, i = 5, 6.$

Node 6:

1) $p = (1, 2, 6), D(p) = \delta_{1,2} + \delta_{2,6}, W(p_i) = w_{2,6}.$
2) $p = (1, 3, 6), D(p) = \delta_{1,3} + \delta_{3,6}, W(p_i) = w_{3,6}.$
3) $p = (1, 4, 6), D(p) = \delta_{1,4} + \delta_{4,6}, W(p_i) = w_{4,6}.$
4) $p = (1, 5, 6), D(p) = \delta_{1,5} + \delta_{5,6}, W(p_i) = w_{5,6}.$
5) $p = (1, 6, 6), D(p) = \delta_{1,6}, W(p_i) = w_{1,6}.$

■

Note that, since any precomputation algorithm has to create all of the discontinuities, the previous instance shows also that the worst case running time of any algorithm is $\Omega(n^3)$. All three algorithms have worst case running times of at least $O(n^4)$. Whether this gap can be closed is an open problem.

In all three algorithms, the node discontinuities $\hat{D}(u)$ were implemented as FIFO queues. Once the algorithms complete, a discontinuity at node u with a specific property can be found by searching through the FIFO queues. The search time will be of the order of the number of elements in $\hat{D}(u)$, which is at most M. In several applications, one is interested in discontinuities of the following form: the discontinuity whose delay is the largest among those whose delay is at most d. For example, this is the case when one is interested in providing a solution to Problem I. Since elements of $\hat{D}(u)$ are stored either in decreasing or in increasing order of discontinuity delay and width, it is helpful to store $\hat{D}(u)$ as an array. With this implementation, we can perform binary search for discontinuities of the previously described form and the search time becomes logarithmic in the number of elements in $\hat{D}(u)$, i.e., $O(\log M) = O(\log N)$. Of course, we also have to determine the associated path using the pointers in $\mathrm{predecessor\_disc}$. Since a path contains at most N nodes, the overall process takes $O(N)$ time in the worst case.

V. Computation of Constrained Bottleneck Multicast Trees

In this section, we show how the node discontinuities obtained in the previous section can be used to provide a quick solution to a problem related to multicast communication.

Assume that the source node s needs to establish a multicast tree session T with a subset U of the nodes. The multicast transmission bandwidth is b and node $u \in U$ has end-to-end delay QoS bound $\bar{d}_u$. Let $\mathbf{d}_U = \{\bar{d}_u : u \in U\}$ and define the width of a tree

$$W(T) = \min_{l \in T} \{w_l\}.$$

Let also $\mathbf{T}_U(\mathbf{d}_U)$ be the set of directed trees with source s, spanning set U (i.e., there is a path on the tree from s to any node in U) and having the following property: the delay of the path on the tree from s to $u \in U$ is at most $\bar{d}_u$. The generalization of Problem I in such a situation is,

Problem II: Given a source node s, a destination node set U and delay requirements $\mathbf{d}_U$, find a tree $T_U^* \in \mathbf{T}_U(\mathbf{d}_U)$ that satisfies

$$W(T_U^*) \geq W(T) \text{ for all } T \in \mathbf{T}_U(\mathbf{d}_U).$$

There is a large body of literature addressing the multicast tree design problem, where emphasis is placed on conserving resources. In particular, maximization of total consumed bandwidth is sought. While it is true that multicasting is used to conserve resources, this is not the only issue that must be addressed. The associated optimization problem for conserving resources is known to be NP-complete and one has to rely on heuristics, which again lead away from the optimal. Trying to develop algorithms that are close to the optimal in terms of resource consumption is time consuming, especially when one has to take into account QoS constraints. This is a reason, among others, that several existing protocols do not rely on multicast trees that solely optimize resources. In the environment we are considering, the situation is aggravated by the demand that QoS objectives are satisfied for each user on the multicast tree. The solution we present here ensures that the QoS objectives of all users are satisfied, provides a form of resource conservation since a tree with maximal width is sought and, as will be seen, is computable very efficiently based on precomputed paths. Hence, it may provide an alternative way of selecting a multicast tree, although further work is needed to evaluate the benefits and disadvantages of each approach.

We assume that we already obtained the discontinuities $\hat{D}(u), u \in V$, by one of the algorithms in the previous section, and that $\hat{D}(u)$ is implemented as an array. Elements of $\hat{D}(u)$ are stored in the array either in decreasing or in increasing order of delay and width. In a practical implementation, the elements $\hat{D}(u)$ will be updated whenever new topology information arrives at the source node s, while Problem II must be solved online, in more frequent intervals, each time a new multicast tree request arrives. We describe next an algorithm for determining quickly an optimal tree for Problem II.

For any node $u \in U$, we can obtain a path $p_u^* \in P_u(\bar{d}_u)$ that solves Problem I by finding the discontinuity in $\hat{D}(u)$ (if one with finite delay exists) whose delay is the largest among the delays of the discontinuities in $\hat{D}(u)$ not exceeding $\bar{d}_u$. Assume that the paths p_u^* exist for all $u \in U$—otherwise, Problem II does not have a solution. Let

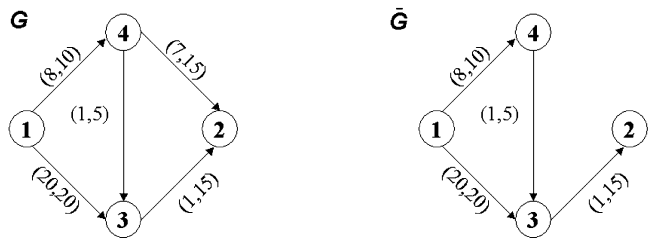
$$W_{\min} = \min_{u \in U} \{W(p_u^*)\}.$$

Let $\bar{G}$ be the graph obtained by including the links and nodes of all the paths $p_u^*, u \in U$. The widths and delays of the links in $\bar{G}$ are the same as in G. Graph $\bar{G}$ is not a tree in general, as the following example shows.

Example 3: Fig. 6 shows a network G with the discontinuities $\hat{D}(u) = \{(\mathrm{delay}, \mathrm{width})\}$ of each node. Next to each discontinuity is the associated path $\{\mathrm{path}\}$. We assume that $s = 1$, and $U = \{2, 3, 4\}$. With $\bar{d}_i, i = 2, 3, 4$ we represent the delay bound for each node i. The assumed values of $\bar{d}_i$ are shown in the figure. Graph $\bar{G}$ is constructed by finding for every $u \in U$, the discontinuity $\hat{D}(u)$ whose delay is the largest among the delays of the discontinuities $\hat{D}(u)$ not exceeding $\bar{d}_u$. For node 2, the delay constraint is $\bar{d}_2 = 13$, thus we choose the first discontinuity with $\mathrm{delay} = 10 < \bar{d}_2$ which corresponds to the path $\{1 \to 4 \to 3 \to 2\}$. Similarly, we choose the following paths to construct graph $\bar{G}$:

$$\{1 \to 3\}, \{1 \to 4\}$$

which is definitely not a tree.



$$\hat{D}(2)=\begin{Bmatrix}(10,5)\\(15,10)\\(21,15)\end{Bmatrix}\begin{matrix}\{1\to4\to3\to2\}\\\{1\to4\to2\}\\\{1\to3\to2\}\end{matrix},\ \bar{d}_2=13$$

$$\hat{D}(3)=\begin{Bmatrix}(9,5)\\(20,20)\end{Bmatrix}\begin{matrix}\{1\to4\to3\}\\\{1\to3\}\end{matrix},\ \bar{d}_3=25$$

$$\hat{D}(4)=\{(8,10)\}\{1\to4\},\ \bar{d}_4=10$$

Fig. 6. Example of a graph G that shows that $\bar{G}$ is not a tree.

Clearly, for the width of $\bar{G}$, it holds that

$$W(\bar{G})=\min_{l\in\bar{G}}\{w_l\}=\min_{u\in U}\{W(p_u^*)\}=W_{\min}. \tag{7}$$

Let $\bar{T}_U$ be the tree in $\bar{G}$ that consists of the shortest delay paths from s to all nodes in U. The next lemma shows that the tree $\bar{T}_U$ solves Problem II and that its width is $W_{\min}$.

Lemma 4: It holds that $W(T_U^*)=W(\bar{T}_U)=W_{\min}$. Hence, $\bar{T}_U$ solves Problem II.

Proof: We first show that

$$W_{\min}\geq W(T_U^*). \tag{8}$$

Let $u\in U$. There is a path p_u in T_U^* that joins s to node $u\in U$. By definition of $\mathbf{T}_U(\mathbf{d}_U), p_u\in P_u(\bar{d}_u)$. Since p_u^* solves Problem I with end-to-end delay $\bar{d}_u$, it holds that

$$W(p_u^*)\geq W(p_u).$$

Hence, taking into account (7), we have

$$W_{\min}=\min_{u\in U}\{W(p_u^*)\}\geq\min_{u\in U}\{W(p_u)\}=W(T_U^*).$$

Next, we show that

$$W_{\min}\leq W(\bar{T}_U)\leq W(T_U^*) \tag{9}$$

which together with (8) implies the lemma.

Since there is a path p_u^* from s to $u\in U$ in $\bar{G}$ with delay $\bar{d}_u$ and $\bar{T}_U$ is a shortest delay path tree in $\bar{G}$, we are ensured that the delay of the path in $\bar{T}_U$ from s to u will be at most $\bar{d}_u$. This implies that $\bar{T}_U\in\mathbf{T}_U(\mathbf{d}_U)$ and therefore

$$W(\bar{T}_U)\leq W(T_U^*).$$

However, since $\bar{T}_U$ is a subgraph of $\bar{G}$, we have

$$W_{\min}=W(\bar{G})\leq W(\bar{T}_U)$$

and hence (9) is true. ■

According to Lemma 4, we can obtain a solution to Problem II by finding the shortest path tree $\bar{T}_U$ in $\bar{G}$. This process involves constructing first the graph $\bar{G}$, which in the worst case takes time $O(n|U|)$. In addition, one has to still determine the tree $\bar{T}_U$. A direct application of a shortest path algorithm requires time $O(N\log N+M)$. It is possible to take advantage of the structure of $\bar{G}$ and reduce the computation time for determining $\bar{T}_U$ to $O(M)$. However, we will not examine this approach further, since, as we will show next, based on the knowledge of the value of the solution to Problem II which according to Lemma 4 is $W_{\min}$, and the discontinuities $\hat{D}(u), u\in V$, we can obtain the required tree in time $O(\max\{|U|\log N,N\})$.

Once we know the value $W_{\min}$ of the solution to Problem II, we can determine a corresponding tree T_U^* as follows. Let (d_u,W_u), be a discontinuity of $\hat{D}(u)$ with the following property.

Property I: The delay d_u of discontinuity (d_u,W_u) is the smallest one among the delays of the discontinuities of $\hat{D}(u)$ whose width is larger than or equal to $W_{\min}$. Let $\hat{p}_u$ be the path from s to $u\in U$ that causes discontinuity (d_u,W_u).

Let $\hat{p}_u=(i_1=s,i_2,\ldots,i_{k-1},i_k=u)$. As discussed in Section III-A with each node $i_n, 2\leq n\leq k$, there is an associated discontinuity (D_{i_n},W_{i_n}), and $(D_{i_{n-1}},W_{i_{n-1}})$ is the predecessor discontinuity of (D_{i_n},W_{i_n}). The following Lemma will be useful in the sequel.

Lemma 5: If (d_u,W_u) satisfies Property I and $\hat{p}_u=(i_1=s,i_2,\ldots,i_{k-1},i_k=u)$, then the associated discontinuities at all nodes $i_n, 2\leq n\leq k$, satisfy Property I.

Proof: The proof is by induction from $n=k$ to $n=1$. The statement is true for $n=k$. Assume that the associated discontinuity (D_{i_n},W_{i_n}) of node $i_n, n\leq k$, satisfies Property I. As $(D_{i_{n-1}},W_{i_{n-1}})$ is the predecessor discontinuity of (D_{i_n},W_{i_n}), it holds that $(D_{i_n},W_{i_n})=(D_{i_{n-1}}+\delta_{i_{n-1},i_n},\min\{W_{i_{n-1}},w_{i_{n-1},i_n}\})$. Since $W_{i_n}\geq W_{\min}$, it also holds that $\min\{W_{i_{n-1}},w_{i_{n-1},i_n}\}\geq W_{\min}$, hence $W_{i_{n-1}}\geq W_{\min}$ and $w_{i_{n-1},i_n}\geq W_{\min}$. This implies that $(D_{i_{n-1}},W_{i_{n-1}})$ satisfies Property I. To see this assume the contrary, i.e., that there is a discontinuity $(D'_{i_{n-1}},W'_{i_{n-1}})$ at node i_{n-1} with $W'_{i_{n-1}}\geq W_{\min}$ and $D'_{i_{n-1}}<D_{i_{n-1}}$. Then the possible discontinuity $(D'_{i_{n-1}}+\delta_{i_{n-1},i_n},\min\{W'_{i_{n-1}},w_{i_{n-1},i_n}\})$ at node i_n must be dominated or be equal to a real discontinuity (D'_{i_n},W'_{i_n}), i.e.,

$$\begin{aligned}D'_{i_n}&\leq D'_{i_{n-1}}+\delta_{i_{n-1},i_n}<D_{i_n}\\W'_{i_n}&\geq\min\{W'_{i_{n-1}},w_{i_{n-1},i_n}\}\geq W_{\min}.\end{aligned}$$

The latter conditions show that (D_{i_n},W_{i_n}) does not satisfy Property I, which is a contradiction. ■

Let $\hat{G}$ be the graph obtained by including the links and nodes of all of the paths $\hat{p}_u, u\in U$. We then have the following theorem on which we can base the algorithm for finding a solution to Problem II.

Theorem 6: The graph $\hat{G}$ is a tree that solves Problem II.

Proof: Assume that a node $v\neq s$ in graph $\hat{G}$ has m outgoing neighbors, and hence this node belongs to at least m of the paths $\hat{p}_u, u\in U$. According to Lemma 5, for each of these paths, the associated discontinuity (d_v,W_v) on node v is the unique one having Property I. Hence, v has a unique incoming neighbor, namely the predecessor node of the discontinuity (d_v,W_v). Since each node in $\hat{G}$ other than s has a unique

TABLE V
ALGORITHM FOR COMPUTING CONSTRAINED BOTTLENECK MULTICAST TREES

Algorithm IV
Input: The array $\hat{D}(u)$ and the delay requirements $\bar{\mathbf{d}}_u$.
Output: The tree $\hat{G}$.

1) **For** each node $u \in U$ determine the value $W(p_u^*)$;
2) Determine $W_{\min} = \min_{u \in U} \{W(p_u^*)\}$;
3) **For** each $u \in U$ determine the discontinuity (d_u, W_u) having Property I;
4) Construct $\hat{G}$ using the predecessor node information that is included in $\hat{D}(u)$, $u \in U$;

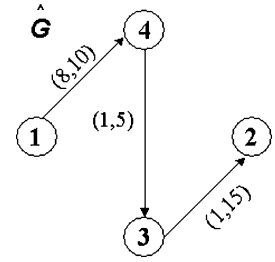


Fig. 7. Tree $\hat{G}$ obtained by the application of Algorithm IV to the graph G shown in Fig. 6.

incoming neighbor and by construction there is a path from s to any node in $\hat{G}$, it follows from [1] that $\hat{G}$ is a tree.

By construction, the width of $\hat{G}$ is at least $W_{\min}$. Also, since the path $\hat{p}_u$ has a delay smaller than or equal to that of p_u^* (both paths have a width larger than or equal to $W_{\min}$ and $\hat{p}_u$ satisfies Property I), we conclude that $\hat{G} \in \mathbf{T}_U(\mathbf{d}_U)$. This together with Lemma 4 imply that $W(\hat{G}) \leq W(T_U^*) = W_{\min}$, hence $\hat{G}$ solves Problem II and its width is $W_{\min}$. ■

According to Theorem 6, to determine the tree $\hat{G}$ we can employ the algorithm shown in Table V. In step 1, for all $u \in U$, a discontinuity in $\hat{D}(u)$ is found whose delay is the largest among the delays of the discontinuities in $\hat{D}(u)$ not exceeding $\bar{d}_u$. In step 2, the minimum width among the widths of the discontinuities found in step 1 is determined. Of course, step 2 can be completed while executing step 1; we state the steps separately for clarity in the description. In step 3, for all $u \in U$, we determine a discontinuity (d_u, W_u) satisfying Property I. Finally, in step 4, we construct tree $\hat{G}$ using the predecessors of the discontinuities found in step 3. Notice that the predecessor information contained in $\hat{D}(u), u \in V$ already defines the tree $\hat{G}$. Specifically, the parent v of node $u \in U$ in the tree $\hat{G}$ is the node associated with the predecessor discontinuity (d_v, W_v) of (d_u, W_u), the parent of node v is the node associated with the predecessor discontinuity of (d_v, W_v), and so on. Step 4 is needed to simply mark the nodes that participate in the multicast tree.

Example 7: Consider again the example in Fig. 6. Following the steps of Algorithm IV, we have the following.

Step 1)
a) Node 2: $p_2^* = \{1 \to 4 \to 3 \to 2\}, W(p_2^*) = 5$.
b) Node 3: $p_3^* = \{1 \to 3\}, W(p_3^*) = 20$.
c) Node 4: $p_4^* = \{1 \to 4\}, W(p_4^*) = 10$.

Step 2) $W_{\min} = \min\{5, 20, 10\} = 5$

Step 3)
a) Node 2: $(d_2, W_2) = (10, 5)$, parent node 3.
b) Node 3: $(d_3, W_3) = (9, 5)$, parent node 4.
c) Node 4: $(d_4, W_4) = (8, 10)$, parent node 1.

Step 4: the nodes 2, 3, 4, and 1 are marked as belonging to the tree.

The tree $\hat{G}$ obtained by applying Algorithm IV is shown in Fig. 7.

Worst Case Analysis of Algorithm IV

Recall that the discontinuities in $\hat{D}(u)$ are stored either in decreasing or increasing order of both width and delay. Hence, using binary search, the determination of each $W(p_u^*), u \in U$ takes $O(\log N)$ time and steps 1 and 3 take time $O(|U| \log N)$. Step 2 takes $O(|U|)$ time. Finally, step 4 takes $O(N)$ time since $\hat{G}$, being a tree, contains at most $N-1$ links. Hence, the overall worst-case running time is $O(\max\{|U| \log N, N\})$.

Notice that there are no complicated structures involved in Algorithm IV, and hence the constants involved in the previous bounds are small.

VI. SIMULATION RESULTS

We run two sets of experiments. Each set employs different methods of network generation. Thus, we generate the following.

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.[1]

Real Internet Networks: These networks were taken from The Real Networks[2] and are based on network topologies observed on the dates 20/09/1998, 01/01/2000, and 01/02/2000.

We also run experiments using "uniform" random networks which are formed by picking uniformly a subset of M links among the link set of a complete N-node graph. For this type of networks, for the same N and M, the running times of the algorithms are smaller than those obtained for Power Law and Real Internet Networks. However, the comparative performance of the algorithms were similar with the performance results of Power Law Networks and therefore are not presented here.

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 two different methods.

- **Width 1**. Each link width w_l is picked randomly with uniform distribution among the integers [1, 100].
- **Width 2**. In this case, link 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 generate Power Law Networks with 400, 800, and 1200 nodes and with ratios $\alpha = M/N$ equal to 4, 8, and 16. For each N and α, we generate ten different networks, and for each network we generate the link widths according to the two methods previously described (Width 1 and Width 2).

[1]The Power Law Simulator. [Online]. Available: http://www.cs.bu.edu/brite

[2]The Real Networks. [Online]. Available: http://moat.nlanr.net/Routing/rawdata

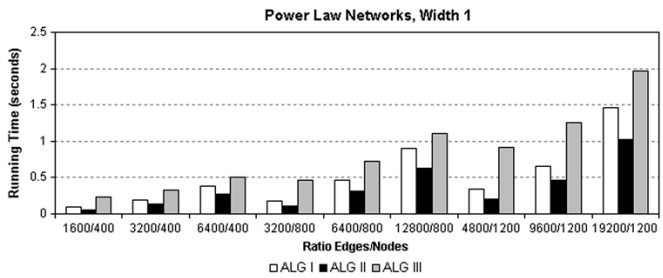


Fig. 8. Running time for Power Law Networks with width independent of delays.

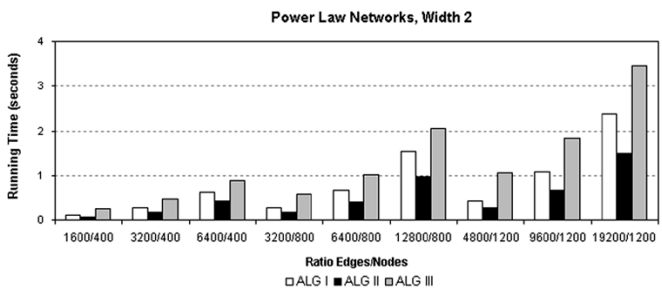


Fig. 9. Running time for Power Law Networks with width correlated to delays.

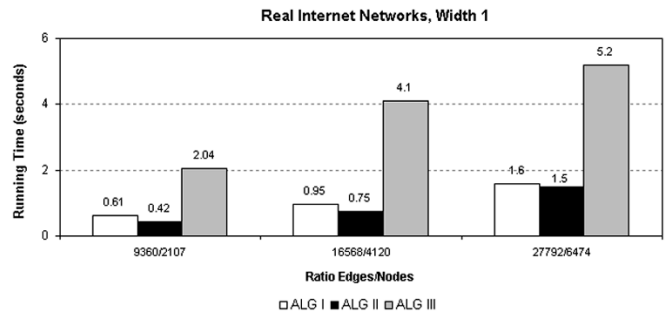


Fig. 10. Running time for Real Internet Networks with width independent of delays.

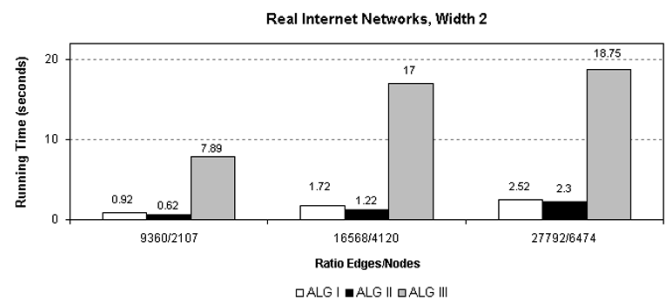


Fig. 11. Running time for Real Internet Networks with width correlated to delays.

The experiments were run on a Pentium PC III, with 1.2 GHz and 256 MB RAM.

As will be discussed in the end of this section, compared to the algorithms in [12], [16], that are based on a similar idea, Algorithm III has better average case performance, while all three algorithms have the same average case memory requirements, and the same worst case running time. Hence, in the following, we use Algorithm III as a representative of this class of algorithms.

In Figs. 8 and 9, we present the average running times (in seconds) of the three algorithms, Algorithms I, II, and III, for Power Law Networks. We make the following observations.

- For a given algorithm and for a fixed number of nodes and edges, we notice that the running time increases when the width values are generated according to the second method (Width 2). This is due to the fact that, when widths are correlated to delays, the number of discontinuities is increased.
- Algorithm II has the best running time performance and Algorithm III the worst.
- Compared to Algorithm II, the running times of Algorithms I and III are found to be up to 1.6 times and 2.3 times larger, respectively.
- Algorithm II performs better than Algorithms I and III for all experiments and especially for large networks.

The Real Internet Networks have $M = 9360, 16568, 27\,792$ links and $N = 2107, 4120, 6474$ nodes, respectively. We refer to these networks as N2107, N4120, and N6474, respectively. The link delays are picked randomly with uniform distribution among the integers [1, 100] and the link widths are generated according to the two methods. In these networks, we also performed ten experiments, where in each experiment we picked randomly a source node. Figs. 10 and 11 show the average running time of the three algorithms. We notice again 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 14 times larger than that of Algorithm II in some experiments. The performance of Algorithm I is worse, but is comparable to that of Algorithm II.

The additional optimization (removal of unneeded links) in Algorithm I improves its running time but not by much. Specifically for the Real Network with $N = 6474$ nodes and $M = 27\,792$ edges, and Width 2 the running time with and without the optimization is, respectively, 2.52 and 2.78 s.

Next we look at the memory requirements of the algorithms. The memory space needed to store the network topology is common to all algorithms and is not presented in the figures below.

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 $\hat{D}(u), u\epsilon V$ as well as: 1) the heap size $\hat{P}$ of possible discontinuities for Algorithm I; 2) 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 3) 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 Figs. 12–15, where we present the memory requirements of the three algorithms for Power Law and Real Internet Networks. Algorithm III has the smallest memory requirements, followed by Algorithm I, whose memory requirements are comparable to those of Algorithm III.

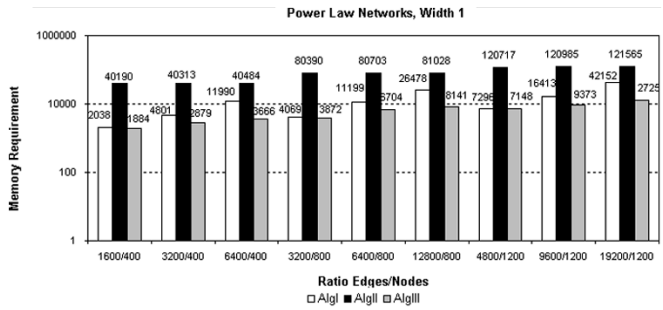


Fig. 12. Memory requirements for Power Law Networks with width independent of delays.

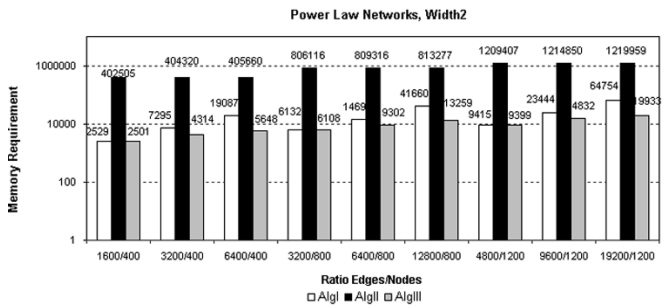


Fig. 13. Memory requirements for Power Law Networks with width correlated to delays.

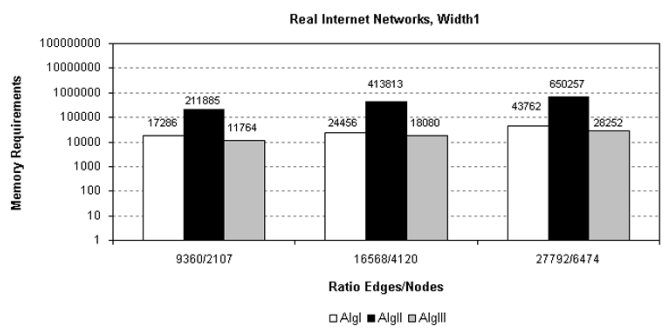


Fig. 14. Memory requirements for Real Internet Networks with width independent of delays.

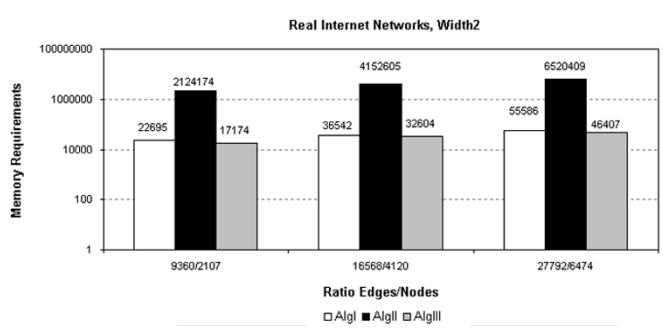


Fig. 15. Memory requirements for Real Internet Networks with width correlated to delays.

Due to the need of array $A[u, k]$, Algorithm II has significantly larger memory requirements.

To compare Algorithm III with the similar algorithms that were proposed in [12] (Algorithm IV—ALG IV) and [16] (Algorithm V—ALG V), we run the algorithms for the three Real Internet Network topologies and Width 2. In the implementation of Algorithms IV and V, we also incorporated the mechanisms in [19] to improve average running time. The results are presented in Table VI. We see that Algorithm III achieves the best running time with Algorithm IV being very close, while Algorithm V has the worst performance. The difference in performance of Algorithm V was explained in the end of Section III-C. For this set of experiments, the use of widest–shortest path algorithm in Algorithm III in place of Dijkstra's algorithm used in Algorithm IV, has a small effect on running time. This is explained by the fact that for the values of link delays and widths used, the number of shortest paths between the source and a given destination is generally small. Hence, using Dijkstra's algorithm often results in also determining the widest–shortest path and not many rounds are wasted determining paths that do not correspond to real discontinuities. To demonstrate this, we run experiments for Real Internet topologies N4120 and N6474, assuming that all link delays are equal to 1 (hence we are interested in paths with smallest hop-count). This choice increases the number of shortest delay paths between a source and a destination. The link widths were picked randomly in the range 1 to u, where $u = 1000, 2000, 5000, 10\,000$. The larger u is, the larger the likelihood that there are several shortest delay paths with different width. The results appear in Table VII. We see that the difference is now more pronounced. Another point of interest in Table VII is that the performance of all three algorithms deteriorates rapidly as u increases. This is due to the fact that as the range of possible width values increases, the number of discontinuities from the source to any of the nodes in the network having the same width, decreases. Hence, while for smaller ranges it is probable that many discontinuities are found at each round, this probability decreases as the range increases. As a result, all three algorithms need more rounds to find all possible discontinuities as the range of link widths increases. In contrast to this, the performance of Algorithms I and II does not depend on the range of width values. For this set of experiments, the running time of Algorithm I and Algorithm II was less than 0.8 s.

TABLE VI
RUNNING TIMES OF ALG III, ALG IV, ALG V FOR THE THREE REAL INTERNET TOPOLOGIES

	N2107	N4120	N6474
ALG III	7.8	17	18
ALG IV	8	17	18
ALG V	14.5	29	33

TABLE VII
RUNNING TIMES FOR REAL INTERNET NETWORKS WITH DELAYS EQUAL TO 1 AND WIDTHS BETWEEN 1 AND U

	u	1000	2000	5000	10000
N4120	ALG III	3.5	52.3	85	101
	ALG IV	3.6	54.5	90.8	109
	ALG V	6.2	113.7	272	438
N6474	ALG III	3.9	58.6	109.8	149
	ALG IV	4	59.8	115.9	162.7
	ALG V	7	125	309	571

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,

Algorithms III, IV, and V have the best memory space requirements; however, their running time is significantly worse than that of Algorithms I and II and depends heavily on the range of width values. Algorithm I represents a good compromise between running time and space requirements, as its performance with respect to these measures, while not the best, is comparable to the best.

VII. Conclusion

We presented three algorithms for precomputing constrained widest paths and multicast trees 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 the 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. Finally, we considered the constrained widest multicast tree problem. We provided an efficient algorithm for constructing a constrained widest multicast tree using the precomputed constrained widest paths.

Acknowledgment

The authors would like to thank the reviewers for their comments and suggestions that helped improve the presentation of the paper.

References

[1] C. Berge, *Graphs*. Amsterdam, The Netherlands: North-Holland Mathematical Library, 1991.
[2] D. Blokh and G. Gutin, "An approximation algorithm for combinatorial optimization problems with two parameters," *Australasian J. Combinatorics*, vol. 14, pp. 157–164, 1996.
[3] S. Chen and K. Nahrstedt, "On finding multi-constrained paths," in *Proc. IEEE Int. Conf. Communications*, Atlanta, GA, Jun. 1998, pp. 874–879.
[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1990.
[5] Y. Cui, K. Xu, and J. Wu, "Precomputation for multi-constrained QoS routing in high speed networks," in *Proc. IEEE INFOCOM*, vol. 2, 2003, pp. 1414–1424.
[6] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. New York: Wiley, 2001.
[7] L. Georgiadis, "Bottleneck multicast trees in linear time," *IEEE Commun. Lett.*, vol. 7, no. 11, pp. 564–566, Nov. 2003.
[8] A. Goel, K. G. Ramakrishnan, D. Kataria, and L. Logothetis, "Efficient computation of delay-sensitive routes from one source to all destinations," in *Proc. IEEE INFOCOM*, Anchorage, AL, Apr. 2001, pp. 854–858.
[9] R. Guerin and A. Orda, "Computing shortest paths for any number of hops," *IEEE/ACM Trans. Netw.*, vol. 10, no. 5, pp. 613–620, Oct. 2002.
[10] R. Guerin, A. Orda, and D. Williams, "QoS routing mechanisms and ospf extensions," in *Proc. IEEE GLOBECOM*, vol. 3, 1997, pp. 1903–1908.
[11] R. Hassin, "Approximation shemes for the restricted shortest path problem," *Math. Oper. Res.*, vol. 17, no. 1, pp. 36–42, Feb. 1992.
[12] T. Korkmaz and M. Krunz, "Source-oriented topology aggregation with multiple QoS parameters in hierarchical networks," *ACM Trans. Modeling Comput. Simulation*, vol. 10, no. 4, pp. 295–325, Oct. 2000.
[13] T. Korkmaz, M. Krunz, and S. Tragoudas, "An efficient algorithm for finding a path subject to two additive constraints," *Comput. Commun. J.*, vol. 25, no. 3, pp. 225–238, Feb. 2002.
[14] T. Korkmaz and M. Krunz, "Bandwidth-delay constrained path selection under inaccurate state information," *IEEE/ACM Trans. Netw.*, vol. 11, no. 3, pp. 384–398, Jun. 2003.
[15] D. H. Lorenz and D. Raz, "A simple efficient approximation scheme for the restricted shortest path problem," *Oper. Res. Lett.*, vol. 28, no. 5, pp. 231–219, Jun. 2001.
[16] K.-S. Lui, K. Nahrstedt, and S. Chen, "Routing with topology aggregation in delay-bandwidth sensitive networks," *IEEE/ACM Trans. Netw.*, vol. 12, no. 1, pp. 17–24, Feb. 2004.
[17] K. Mehlhorn and S. Naher, *Leda: A Platform for Combinatorial and Geometric Computing*. Cambridge, U.K.: Cambridge Univ. Press, 2000.
[18] P. Van Mieghem, H. De Neve, and F. A. Kuipers, "Hop-by-hop quality of service routing," *Comput. Networks*, vol. 37, no. 3–4, pp. 407–423, Nov. 2001.
[19] P. Narvaez, K.-Y. Siu, and H.-Y. Tzeng, "New dynamic algorithms for shortest path tree computation," *IEEE/ACM Trans. Netw.*, vol. 8, no. 6, pp. 734–746, Dec. 2000.
[20] A. Orda and A. Sprintson, "Precomputation schemes for QoS routing," *IEEE/ACM Trans. Netw.*, vol. 11, no. 4, pp. 578–591, Aug. 2003.
[21] ——, "A scalable approach to the partition of QoS requirements in unicast and multicast," in *Proc. IEEE INFOCOM*, 2002, pp. 685–694.
[22] S. Siachalou and L. Georgiadis, "Efficient QoS routing," *Comput. Networks J.*, vol. 43, no. 3, pp. 351–367, 2003.
[23] J. L. Sobrino, "Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet," in *Proc. IEEE INFOCOM*, vol. 2, Anchorage, AL, Apr. 2001, pp. 727–735.

Stavroula Siachalou was born in Thessaloniki, Greece, in 1976. She received the M.Sc. degree from Aristotle University of Thessaloniki, Thessaloniki, Greece, in 2000, where she is currently working toward the Ph.D. degree.

Her research interests are in the area of QoS provisioning, routing, and multicast communication.

Ms. Siachalou is a member of the Technical Chamber of Greece.

Leonidas Georgiadis (M'86–SM'95) received the Diploma degree in electrical engineering from Aristotle University of Thessaloniki, Thessaloniki, Greece, in 1979 and the M.S. and Ph.D. degrees in electrical engineering from the University of Connecticut, Storrs, in 1981 and 1986, respectively.

From 1981 to 1983, he was with the Greek army. From 1986 to 1987, he was a Research Assistant Professor with the University of Virginia, Charlottesville. In 1987, he joined IBM T. J. Watson Research Center, Yorktown Heights, NY, as a Research Staff Member. Since October 1995, he has been with the Telecommunications Department, Aristotle University of Thessaloniki. His interests are in the area of wireless networks, high-speed networks, routing, scheduling, congestion control, modeling, and performance analysis.

Prof. Georgiadis is a Senior Member of the IEEE Communications Society. In 1992, he was the recipient of the IBM Outstanding Innovation Award for his work on goal-oriented workload management for multi-class systems.