Algorithms for Precomputing Constrained Widest Paths and Multicast Trees

Stavroula Siachalou, Leonidas Georgiadis

Abstract— **We consider the problem of precomputing constrained widest paths and multicast trees 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. 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 is sought with maximal bandwidth (widest multicast tree), which in addition satisfies given end-to-end delay constraints for each path on the tree from the source to a multicast destination.**

Keywords— **Precomputation, QoS Routing, Multicast Trees, Widest Paths, Widest Trees, Bottleneck Paths, Graph Theory.**

I. 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 the 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.

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 first 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, but not known a priori, end-to-end delay requirements. 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] using data structures that take advantage of 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.

Next, 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.

The rest of the paper is organized as follows. The problem is formulated in Section II. We present the three algorithms in Section III and in Section IV we examine the algorithms in 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 of the work 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 $p = (u_1, u_2, ..., u_k)$, such that $u_i \neq u_j$ for

S. Siachalou and L. Georgiadis are with Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki, Thessaloniki, Greece. E-mails: stavroula@psyche.ee.auth.gr, leonid@eng.auth.gr

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

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

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,

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

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 $P_{su}(d)$; 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 III-A). 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 - in Section V we present such a situation. 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 $W_u^*(d)$, $u \in V$.

III. 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 III-A. In Sections III-B and III-C 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 intend is to compare these algorithms in terms of worst case, average running times and space requirements.

A. 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 (d_i, W_i), $i = 1, 2$, as follows:

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

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 in Table I.

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,

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;

TABLE I
GENERIC ALGORITHM FOR SOLVING PROBLEM I

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 $(d, w, u, predecessor\_disc)$, where for the source node s, $predecessor\_disc = null$, and for any other node u, $predecessor\_disc$ 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

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

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 $g_1 < g_2 < ... < g_K$. 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.

• 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 in Table 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 increasing the time that the get_min operation takes in step 7 of Generic Algorithm I in Table I. The trade-off 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* $\widehat{P}$*, 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. Table II presents the pseudocode for Algorithm II. The real discontinuities $\widehat{D}(u)$, $u \in V$ are again implemented as FIFO queues.

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.

C. 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 in Table III. Again $\widehat{D}(u)$, $u \in V$ are implemented as FIFO queues.

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

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

Proof: We will show by induction that at iteration m, all discontinuities in $\widehat{D}$ 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 $(D(p_u), W(p_u))$ found 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 $W(q_{su}) > W_m^*$. Assume the contrary. Notice first

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. $\widehat{D}(u) = \{(-\infty, -\infty, u)\}$, $u \in V$;
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
(a) $e_{new} = (d+\delta_{sv}, \min\{k', r(w_{sv})\}, v)$; $k = e_{new}.width$;
(b) $A[v,k] = e_{new}$; insert$(e_{new}, H[k])$;
7. endo
8. Determine the largest index k' such that $H[k']$ is nonempty;
9. get_min$(e, H[k']), (d, k', u) = e$;
10. Set $A[u, k']$ to null;
11. $L = k'$;
12. /*End Initialization */
13. 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$(e_{new}, e, H[k])\}\}$
14. endo
15. If $L = 1$ and $H[1]$ is empty, then stop. Else
16. Find the largest $k' \leq L$ such that the heap $H[k']$ is nonempty
17. get_min$(e, H[k'])$; $(d, k', u) = e$;
18. Set $A[u, k']$ to null;
19. $L = k'$;
20. If $d \geq d_m$ where (d_m, k_m, u) is the last element of $\widehat{D}(u)$ then go to step 16. Else,
21. add e to $\widehat{D}(u)$;
22. go to step 13;

TABLE II
Pseudocode of Algorithm II

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.

TABLE III
Iteration Steps of Algorithm III

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

$$W(q_u) > W(p_u) \text{ and } D(q_u) = D(p_u).$$

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 [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.

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 $N^{in}_{\max}$ be the maximum in-degree of the nodes in V, that is,

$$N^{in}_{\max} = \max_{u \in V}\{V_{in}(u)\}.$$

Clearly, $N^{in}_{\max} \leq N$. In all three algorithms we assume a Fibonacci heap implementation [5]. In such implementation of a heap H, all operations except get_min(e, H) take $O(1)$ time. Operation get_min(e, H) takes $O(\log L)$ time, where L is the number of elements in the heap.

Algorithm I

The analysis of this algorithm has been presented in [17]. According to this analysis the following hold.

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

Memory Requiremens: 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 13-14 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 $\widehat{P}(u)$, $u \in V$ 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 Dijkstra Algorithm 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 [13]. 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 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. A more detailed analysis of ALG II will also reveal 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.

	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	$O\left(MN \log N + M^2\right)$	$O(MN)$

TABLE IV
WORST CASE RUNNINING TIMES AND 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 discontinuities and it is not immediately clear whether this can happen. Next we present an instance where indeed the memory requirements are $\Omega(N^3)$.

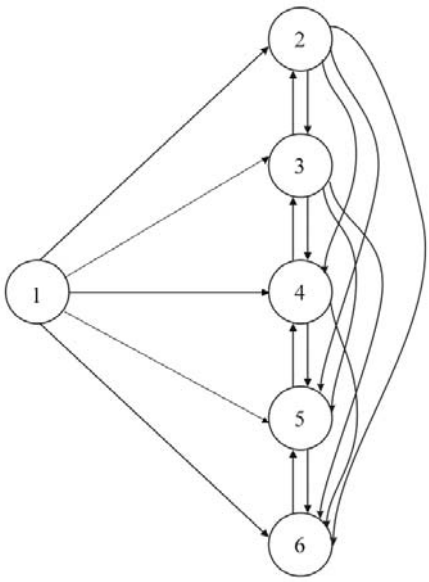

Fig. 1. Example of a network with $\Omega(N^3)$ memory requirements.

Consider a network consisting of n nodes and the following edges

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

Figure 1 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

$$w_{i+1,i}=\infty,\ \delta_{i+1,i}=0, i=2,...,n-1 \tag{1}$$

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

$$w_{1,i}>w_{1,i+1},,\ \delta_{1,i}>\delta_{1,i+1},\ i=2,...n-1 \tag{2}$$

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

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

and in general

$$\begin{aligned} &w_{1,i}>w_{i,j}>w_{j,j+1}>w_{1,i+1}, \\ &\delta_{1,i}>\delta_{i,j}>\delta_{i,j+1}>\delta_{1,i+1}, \\ &i=2,...,n-1,\ j=i+1,...,n \end{aligned} \tag{5}$$

It is easy to see that widths and delays satisfying relations (1)-(5) can be specified.

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

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

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

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

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

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

Note that since any precomputation algorithm has to create all 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 time at least $O(n^4)$. Whether this gap can be closed is an open problem.

In all three algorithms the node discontinuities $\widehat{D}(u)$ 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 $\widehat{D}(u)$ which is at most M. In several applications one is interested in discontinuities of the 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 $\widehat{D}(u)$ are stored either in decreasing or in increasing order of discontinuity delay and width, it is helpful to store $\widehat{D}(u)$ 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 $\widehat{D}(u)$, i.e., $O(\log M) = O(\log N)$. Of course, we also have to determine the associated path using the pointers in *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 quick solution to a problem related to multicast communication. We assume that we already obtained the discontinuities $\widehat{D}(u)$, $u \in V$, and that $\widehat{D}(u)$ is implemented as an array. Elements of $\widehat{D}(u)$ are stored in the array either in decreasing or in increasing order of delay and width.

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 $\overline{d}_u$. Let $\mathbf{d}_U = \left\{\overline{d}_u : u \in U\right\}$ 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 $\overline{d}_u$. The analogous problem to 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

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

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 $p_u^* \in P_u(\overline{d}_u)$ that solves Problem I by finding the discontinuity in $\widehat{D}(u)$ (if one with finite delay exists) whose delay is the largest among the delays of the discontinuities in $\widehat{D}(u)$ not exceeding $\overline{d}_u$. Assume that the paths p_u^* exist of all $u \in U$ - otherwise Problem II does not have a solution. Let

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

Let $\overline{G}$ 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 $\overline{G}$ are the same as in G. Graph $\overline{G}$ is not a tree in general, as the following example shows.

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

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

which is definitely not a tree.

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

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

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

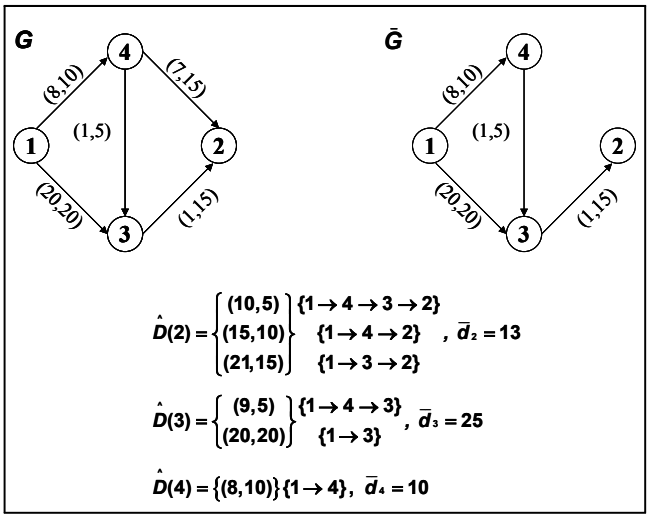

Fig. 2. Example of a graph G that shows $\bar{G}$ is not a tree.

Lemma 3: It holds $W(T_U^*) = W(\overline{T}_U) = W_{\min}$. Hence $\overline{T}_U$ solves Problem II.

Proof: We first show that

$$W_{\min} \geq W(T_U^*). \tag{6}$$

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

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

Hence

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

Next we show that

$$W_{\min} \leq W(\overline{T}_U) \leq W(T_U^*), \tag{7}$$

which together with (6) implies the lemma.

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

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

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

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

and hence (7) is true. ■

According to Lemma 3, we can obtain a solution to Problem II by finding the shortest-path tree $\overline{T}_U$ in $\overline{G}$. This process involves constructing first the graph $\overline{G}$, which in the worst case takes time $O(n\,|U|)$. In addition one has to still determine the tree $\overline{T}_U$. 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 $\overline{G}$ and reduce the computation time for determining $\overline{T}_U$ 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 3 is $W_{\min}$, and the discontinuities $\widehat{D}(u)$, $u \in V$, 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 $\widehat{D}(u)$ 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 $\widehat{D}(u)$ whose width is larger than or equal to $W_{\min}$. Let $\widehat{p}_u$ be the path from s to $u \in U$ that causes discontinuity (d_u, W_u).

Let $\widehat{p}_u = (i_1 = s, i_2, ..., i_{k-1}, i_k = u)$. 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}), where $(D_{i_{n-1}}, W_{i_{n-1}})$ is the predecessor discontinuity of (D_{i_n}, W_{i_n}). The following Lemma will be useful in the sequel.

Lemma 4: If (d_u, W_u) satisfies Property I and $\widehat{p}_u = (i_1 = s, i_2, ..., i_{k-1}, i_k = u)$, 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 $(D_{i_n}, W_{i_n}) = (D_{i_{n-1}} + \delta_{i_{n-1},i_n}, \min\{W_{i_{n-1}}, w_{i_{n-1},i_n}\})$. Since $W_{i_n} \geq W_{\min}$ it also holds $\min\{W_{i_{n-1}}, w_{i_{n-1},i_n}\} \geq W_{\min}$, 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.,

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

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

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

Theorem 5: The graph $\widehat{G}$ is a tree that solves Problem II.

Proof: Assume that a node $v \neq s$ in graph $\widehat{G}$ has m outgoing neighbors, hence this node belongs to at least m of the paths $\widehat{p}_u$, $u \in U$. According to Lemma 4, 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 $\widehat{G}$ other than s has a unique incoming neighbor and by construction there is a path from s to any node in $\widehat{G}$, it follows [1] that $\widehat{G}$ is a tree.

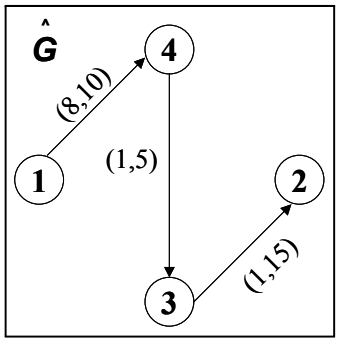

Fig. 3. Tree $\hat{G}$ obtained by the application of Algorithm IV to the graph G shown in Figure 2.

Algorithm IV
Input: The array $\hat{D}(u)$ and the delay requirements $\overline{\mathbf{d}}_u$.
Output: The tree $\widehat{G}$.
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 $\widehat{G}$ using the predecessor node information that is included in $\widehat{D}(u)$, $u \in U$.

TABLE V
Algorithm For Computing Constrained Bottleneck Multicast Trees

By construction, the width of $\widehat{G}$ is $W_{\min}$. Also, since the path $\widehat{p}_u$ has delay smaller than or equal to that of p_u^* (both paths have width larger than or equal to $W_{\min}$ and $\widehat{p}_u$ satisfies Property I), we conclude that $\widehat{G} \in \mathbf{T}_U(\mathbf{d}_U)$ and hence it solves Problem II. ■

According to Theorem 5, to determine the tree $\widehat{G}$ we can employ the algorithm shown in Table V. In Figure 3 we show the tree $\widehat{G}$ obtained by applying Algorithm IV to the graph G shown in Figure 2.

Worst Case Analysis of Algorithm IV

Recall that the discontinuities in $\widehat{D}(u)$ 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, 3 take time $O(|U| N)$. Step 2 takes $O(|U|)$ time. Finally, step 4 takes $O(N)$ time since $\widehat{G}$, 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:

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.

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 three 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, 16. For each N and α we generate 10 different networks and for each network we generate the link widths according to the two methods previously described (Width 1 and Width 2).

The experiments were run on a Pentium PC IV, 1.7GHz, 256MB RAM.

In figures 4-5 we present the average running times (in seconds) of the three algorithms for Power Law Networks. We make the following observations.

- For a given algorithm and for 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 - see Figures 8 and 9.
- 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 Real Internet Networks have $N = 2107$, 4120, 6474 nodes and $M = 9360$, 16568, 27792 links 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 10 experiments, where in each experiment we picked randomly a source node. Figures 6-7 show the average running time of the three algorithms. We notice

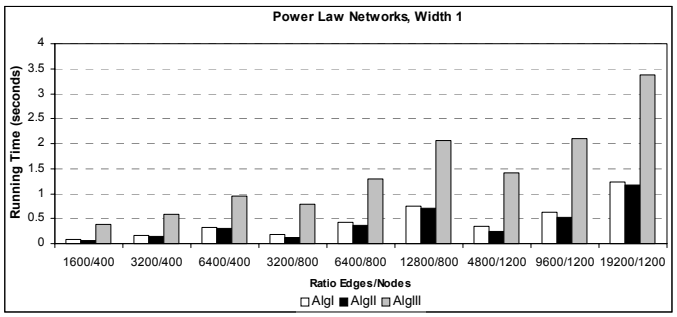

Fig. 4. Running Time for Power Law Networks with width independent of delays.

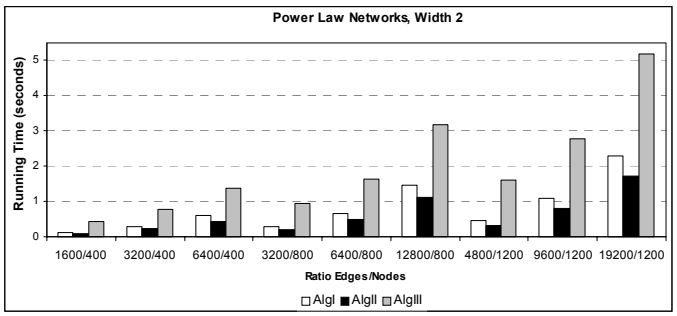

Fig. 5. Running Time for Power Law Networks with width correlated to delays.

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 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.

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, $M = 27792$ edges and Width 2 the running time with and without the optimization is respectively 3.1 and 2.89 seconds.

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 $\widehat{D}(u)$, $u \epsilon V$ 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

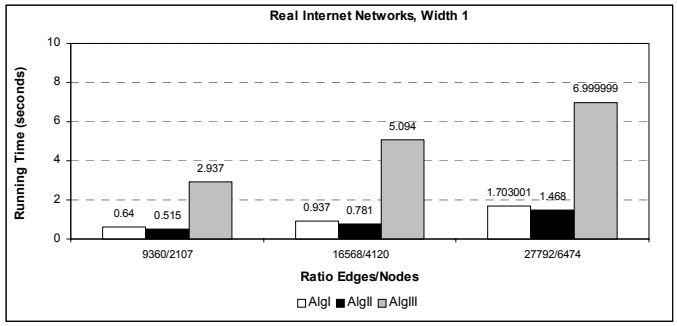

Fig. 6. Running Time for Real Internet Networks with width independent of delays.

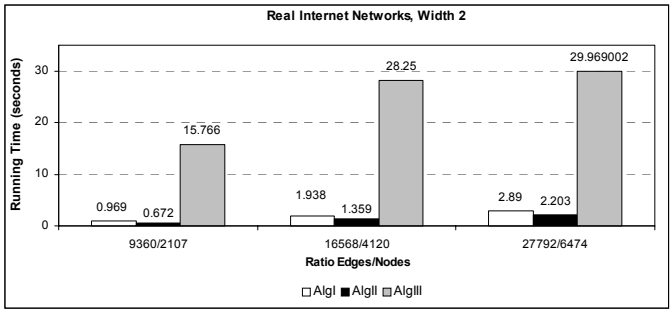

Fig. 7. Running Time for Real Internet Networks with width correlated to delays.

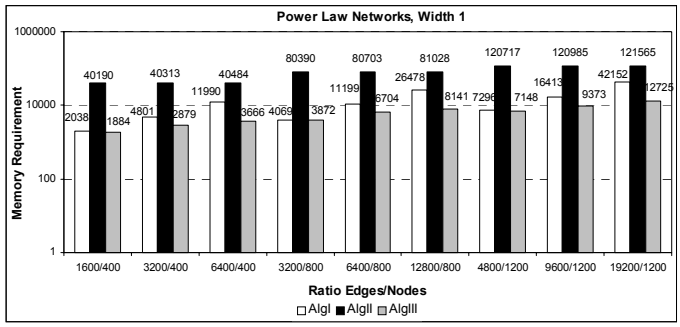

Fig. 8. Memory Requirements for Power Law Networks with width independent of delays.

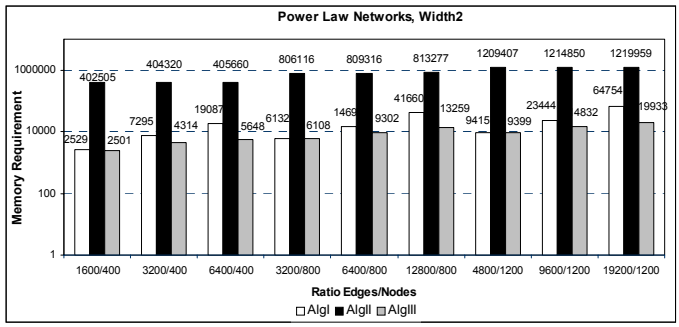

Fig. 9. Memory Requirements for Power Law Networks with width correlated to delays.

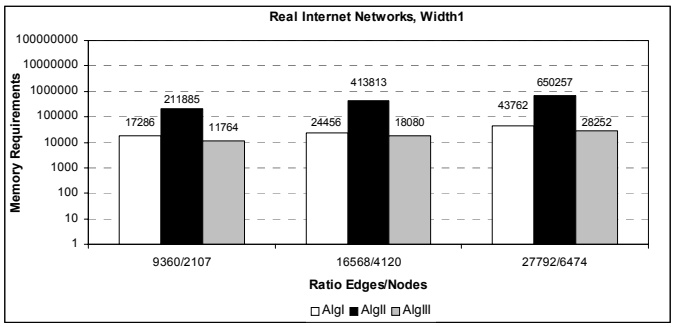

Fig. 10. Memory Requirements for Real Internet Networks with width independent of delays.

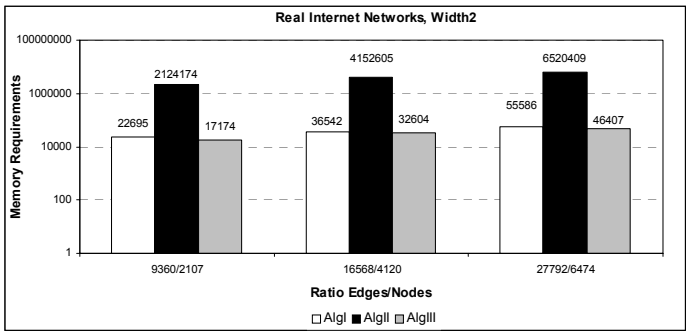

Fig. 11. Memory Requirements for Real Internet Networks with width correlated to delays.

of Algorithm II are significantly larger than those of the other two algorithms. This is indicated in Figures 8-11 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.

VII. Conclusions

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 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 that provides a constrained widest multicast tree using the precomputed constrained widest paths.

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] Roch Guerin, Ariel Orda, "Computing Shortest Paths for Any Number of Hops", *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, October 2002.
[9] Roch Guerin, Ariel Orda and Williams D., "QoS Routing Mechanisms and OSPF Extensions", *IEEE INFOCOM 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] Kurt Mehlhorn, Stefan 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] Paolo 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", *IEEE INFOCOM 2003*, to be published in *Computer Networks Journal.*
[18] The Power Law Simulator, *http://*www.cs.bu.edu/brite.
[19] The Real Networks, http://moat.nlanr.net/Routing/raw-data.