Efficient QoS Routing

Stavroula Siachalou, Leonidas Georgiadis

***Abstract*— We consider the problem of routing in a network where QoS constraints are placed on network traffic. We provide two optimal algorithms that are based on determining the discontinuities of functions related to the optimization at hand. The proposed algorithms have pseudopolynomial worst case running time and for a wide variety of tested networks they have fairly satisfactory running times. They perform significantly better than the algorithm based on the direct application of the Dynamic Programming equations and can also be used in conjunction with known polynomial-time approximation algorithms to provide good average case behavior, in addition to guaranteeing polymonial worst-case running time.**

***Index Terms*— Network Routing, QoS Routing, Graph theory, Simulations**

I. Introduction

Transmission of multimedia traffic presents many challenges to the network designer. Voice and video packet streams require certain bandwidth as well as bounds on delay, loss probability and jitter in order to maintain reception quality. These issues give rise to the problem of routing multimedia traffic so that Quality of Service (QoS) is maintained, [11], [13],[14] otherwise known as the Constrained Shortest Path Routing Problem.

The QoS Routing Problem consists in finding an optimal-cost path from a source to a destination subject to one or more constraints (e.g., total delay and loss probability) on the path. It is well known that this problem is NP-complete [5] and several heuristics have been proposed for its solution [2], [3], [8]. On the other hand, fully polynomial ϵ-approximate solution schemes for the problem exist [7], [15]. These results were recently improved in [9] and were applied to related problems in [10]. In [6], polynomial algorithms have been proposed that find the optimal solution from one source to all destinations, within ϵ-deviation from the set path constraint. The algorithms in [6] and [9] use as a subroutine the iterations implied by the Dynamic Programming equation related to the problem at hand.

In this paper we provide two optimal algorithms for the QoS Routing problem. The algorithms consist in finding the discontinuity points of functions related to the optimization problem. Although pseudopolynomial, tests with a wide variety of networks, link costs and link constraints, show that the proposed algorithms have fairly satisfactory performance and can be used in practical systems. Moreover, if guaranteed polynomial worst-case running time is also desired, the algorithms can replace the dynamic programming recursions in the approximate polynomial-time algorithms in [6] and [9], to improve their average running time.

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

The rest of the paper is organized as follows. In Section II we define the problem and provide the optimal algorithms. In Section III we provide a set of numerical experiments that evaluate the performance of the proposed algorithms. Conclusions are presented in Section IV.

II. Problem Formulation and Algorithms

Let $G = (N, L)$ be a graph with node set N and link set L. A link with origin node m and destination node n is denoted by (m, n). With $N_+(n)$ and $N_-(n)$ we denote the set of incoming and outgoing neighbors to node n, that is, respectively,

$$N_+(n) = \{m \in N : \ (m, n) \in L\},$$
$$N_-(n) = \{m \in N : \ (n, m) \in L\}.$$

With each link $l = (m, n)$, $m, n \in N$ there is an associated cost $c_{mn} \geq 0$ and delay $d_{mn} \geq 0$. If $p = (m_1, ..., m_k)$ is a directed path (a subgraph of G consisting of nodes $m_1, ..., m_k$, $m_i \neq m_j$ for all $1 \leq i, j \leq k$, $i \neq j$, and links (m_i, m_{i+1}), $1 \leq i \leq k-1$) then we define the cost and delay of the path respectively,

$$C(p) = \sum_{(m,n)\in p} c_{mn},$$
$$D(p) = \sum_{(m,n)\in p} d_{mn}.$$

The set of all paths with origin node s, destination node n and delay less than of equal to d is denoted by $P_{sn}(d)$. The set of all paths from s to n is denoted simply by P_{sn}. For any d, we are interested in finding a path $p^* \in P_{sn}(d)$ such that

$$C(p^*) \leq C(p) \text{ for all } p \in P_{sn}(d).$$

Let $C_n^*(d)$ be the minimum of the costs of the paths $p \in P_{sn}(d)$. If $P_{sn}(d) = \varnothing$, we define $C_n^*(d) = \infty$. For node s, we also define

$$C_s^*(d) = \begin{cases} \infty & \text{if} \quad d < 0 \\ 0 & \text{if} \quad d \geq 0 \end{cases}.$$

The algorithms to be presented below depend heavily on the properties of the functions $C_n^*(d)$. These properties are presented in the lemmas below.

Lemma 1: The functions $C_n^*(d)$, $n \in N$, $n \neq s$, satisfy the following equations.

$$C_n^*(d) = \min_{m \in N_+(n)} \{c_{mn} + C_m^*(d - d_{mn})\}$$

Proof: These are the dynamic programming equations for the problem at hand [1, Problem 4.46.]. We only note that due to the fact that the links costs are nonnegative, we can use the equations as stated in the lemma instead of

$$C_n^*(d) = \min \left\{ \begin{array}{c} C_n^*(d-1), \\ \min_{m \in N_+(n)} \{c_{mn} + C_m^*(d - d_{mn})\} \end{array} \right\},$$

and we do not need to make any integrality assumptions on d. ■

Lemma 2: For any $n \in N$, $C_n^*(d)$ has the following properties

1) $C_n^*(d)$ is nonincreasing
2) If $C_n^*(d) < \infty$ for some d, then $C_n^*(d)$ is a piecewise constant, right continuous function of d. If $C_n^*(d)$ is discontinuous at d_0, then
   a) There is a path p_d^* such that $C_n^*(d) = C(p_d^*)$ and $D(p_d^*) = d_0$.
   b) There is an $m \in N_+(n)$ such that $C_m^*(d)$ is discontinuous at $d_0 - d_{mn}$ and $C_n^*(d_0) = C_m^*(d_0 - d_{mn}) + c_{mn}$.
3) The set D_n, of points (real number pairs) $(d, C_n(d))$ at which discontinuities of $C_n(d)$ occur, is finite.

Proof: 1) The fact that $C_n^*(d)$ is nonincreasing follows directly from the definition.

2.a) Observe that if $C_n^*(d) < \infty$, then there must exist a path $p \in P_{sn}(d)$ with $C(p) = C_n^*(d)$. Let p_d^* be a path with smallest delay among the paths that have cost $C_n^*(d)$. By definition,

$$C_n^*(d) = C(p_d^*). \tag{1}$$

Moreover it holds that

$$C_n^*(d') > C_n^*(d) \text{ for } d' < D(p_d^*). \tag{2}$$

To see this note that if $C_n^*(d') = C_n^*(d)$ then there must exist a path q with delay at most $d' < D(p_d^*)$ such that $C(q) = C_n^*(d') = C_n^*(d)$, which contradicts the definition of p_d^*. Since $C_n^*(d)$ is nonincreasing, the only possibility is that $C_n^*(d') > C_n^*(d)$ for $d' < D(p_d^*)$.

Next, let $\widehat{p}_d$ be the path with smallest delay among the paths in the set

$$\overline{P}(d) = \{p \in P_{sn} : C(p) < C_n^*(d)\}.$$

In case $\overline{P}(d) = \varnothing$ define $D(\widehat{p}_d) = \infty$. Then, it holds

$$C_n^*(d') = C(p_d^*) \text{ for } D(p_d^*) \le d' < D(\widehat{p}_d). \tag{3}$$

To see this note that if $C_n^*(d') < C(p_d^*)$ for some d' such that $D(p_d^*) \le d' < D(\widehat{p}_d)$, then there must exist a path q with delay at most $d' < D(\widehat{p}_d)$ such that $C(q) = C_n^*(d') < C_n^*(p_d^*) = C_n^*(d)$, which contradicts the definition of $\widehat{p}_d$. Since $C_n^*(d)$ is nondecreasing, the only possibility is that $C_n^*(d') = C_n^*(d)$ for $D(p_d^*) \le d' < D(\widehat{p}_d)$.

The fact that $C_n^*(d)$ is a piecewise constant, right continuous function, as well as 2.a, follow from (1), (2) and (3).

2.b) Let $\mathcal{M}$ be the set of nodes $k \in N_+(n)$ for which it holds

$$C_n^*(d_0) = C_k^*(d_0 - d_{kn}) + c_{kn}.$$

It follows from Lemma 1 that $\mathcal{M}$ is non empty and

$$C_n^*(d_0) < C_k^*(d_0 - d_{kn}) + c_{kn}, \text{ for } k \in N_+(n) - \mathcal{M} \tag{4}$$

Since $C_n^*(d)$ is discontinuous at d_0, the continuity of the function $\min\{\bullet\}$ implies that for at least one node m in $\mathcal{M}$, $C_m^*(d)$ is discontinuous at $d_0 - d_{mn}$.

3) According to 2), if $C_n^*(d)$ is discontinuous a d_l, there exist a path $p_{d_l}^*$ such that $D(p_{d_l}^*) = d_l$. Hence to different discontinuities (i.e., different d_l) correspond different paths and the statement follows from the fact that the number of paths in the network is finite. ■

The following observations that follow from the lemmas are important in the development of the algorithms below.

- Since $C_n^*(d)$ is piecewise constant, knowing its discontinuity points in fact determines the whole function.
- Let $C_n^*(d)$ be discontinuous at d_0. According to Lemma 2, 2.b, there is an $m \in N_+(n)$ such that $C_m^*(d)$ is discontinuous at $d_0 - d_{mn}$ and $C_n^*(d_0) = C_m^*(d_0 - d_{mn}) + c_{mn}$. Moreover, according to Lemma 2, 2.a, there is a path $q \in P_{sm}(d_0 - d_{mn})$ such that $C_m^*(d_0 - d_{mn}) = C(q)$ and $D(q) = d_0 - d_{mn}$. Therefore, the path p_d^* obtained by adjoining link (m, n) to q, is a path with delay d_0 and cost $C_n^*(d_0)$. The triple $(d_0 - d_{mn},\ C_m^*(d_0 - d_{mn}),\ m)$ is called *predecessor* of $(d_0, C_n^*(d_0), n)$, and $(d_0, C_n^*(d_0), n)$ *successor* of $(d_0 - d_{mn},\ C_m^*(d_0 - d_{mn}), m)$.
- Suppose that we know that for $m \in N$, $C_m^*(d_0)$ is discontinuous at d_0 and in addition we know a path q for which $C_m^*(d_0) = C(q)$, $d_0 = D(q)$. Then the possible successors of $(d_0, C_m^*(d_0), m)$ are the triplets $(d_0 + d_{mn}, C_m^*(d_0 + d_{mn}), n)$ for $n \in N_-(m)$. If we find a way of deciding which of these *possible* successors are *actual* ones, then we will immediately know the corresponding path by adjoining node n to p.

In the following we also make use of the lexicographic order between pairs of real numbers. We say that the pair (d_1, c_1) of real numbers is lexicographically smaller (or simply smaller if there is no possibility for confusion) than (d_2, c_2) and write

$$(d_1, c_1) < (d_2, c_2),$$

if either $d_1 < d_2$, or $d_1 = d_2$ and $c_1 < c_2$.

A. *ALGORITHM I*

Let $D = \cup_{n \in N} D_n$ be the set of all discontinuities of the functions $C_n^*(d)$, $n \in N$. The proposed algorithm determines the discontinuities in D in nondecreasing lexicographic order. Clearly, a smallest discontinuity occurs at $(0, 0)$ for the function $C_s(d)$. We assume the implementation of queues and heap structures with the following operations [1].

- Queue structure Q
  - head(e, Q): returns (i.e., shows or points to, without removing) the first element e of the Q.
  - tail(e, Q): returns the last element e of the Q.
  - enqueue(e, Q): inserts element e at the end of Q.
  - dequeue(e, Q): removes and returns the head element e of Q.
  - size_of(Q): returns the size of Q.

  All previous operations on Q take $O(1)$ time
- A heap H, using key K.
  - create_heap(H): creates an empty heap H.
  - insert(e, H): inserts element e to H.
  - find_min(e, H): returns an element e in H with the smallest key.

- 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}. Element e_{new} has smaller key than e.
- increase_key((e_{new}, e, H): replaces in H element e with e_{new}. Element e_{new} has larger key than e.

We assume a Fibonacci heap implementation [4]of H, so that all previous operations with the exception of get_min(e, H) and increase_key(e_{new}, e, H) take $O(1)$ time. Operations get_min(e, H) and increase_key(e_{new}, e, H) take $O(\log N)$ time, where N is the size of the heap.

The following structures are maintained during the execution of the algorithm.

- An array $A_b[n]$ of queues. Queue $A_b[n]$ holds the currently known discontinuities of $C_n^*(d)$ in nondecreasing order. Its elements are of the form e_b =(*delay*, *cost*, *predecessor, discontinuity_node*), where (*delay, cost*) is a discontinuity pair of $C_n^*(d)$, n =*discontinuity_node* and *predecessor* is the predecessor node of (*delay,cost*, n). Parameter *discontinuity_node* is of course redundant, but we keep it in order to simplify the presentation of the code.
- A heap H_a. This heap contains *possible* discontinuities that may be successors of some of the already known discontinuities. Each element e_a of this heap is of the same form as the elements of $A_b[n]$. The key for the heap elements is the pair (*delay,cost*) of e_a.

The proposed algorithm is shown in Figure 1. The algorithm starts by initializing the queues $A_b[n]$, $n \in N$ and the heap H_a (steps 1-4). At this stage only the queue corresponding to the source node s is nonempty, containing the single discontinuity at $(0, 0)$ and with null predecessor. The rest of the queues are initialized to $(-\infty, \infty, null, n)$. The latter initialization is done in order to facilitate the description of the code. The heap H_a contains the possible successor discontinuities of $(0, 0, s)$ (steps 5-7). The latter (possible discontinuities) consist of one possible discontinuity of the form (d_{sn}, c_{sn}, s, n) for each of the outgoing neighbors of s.

In the **while** loop, line 8, the algorithm removes the minimum key discontinuity e_a among the possible discontinuities in H_a. Next, it compares the cost parameter of the key of e_a with the cost parameter of the key of the tail element e_b in the queue that corresponds to the *discontinuity_node* of e_a . If e_a.*cost* is larger than or equal to e_b.*cost*, then e_a is discarded. Else the discontinuity represented by e_a is enqueued to the discontinuities of the queue corresponding to $m = e_a$.*discontinuity_node*. Next, in the **for all** loop, line 14, a possible discontinuity is added to H_a for each outgoing neighbors of m. A further optimization step is taken here by avoiding to create possible discontinuities for the node $n \in N_-(m)$ for which $n = e_a$.*predecessor*, since such a discontinuity is impossible.

The algorithm stops when $H_a = \varnothing$, that is, when there are no possible discontinuities left to be examined.

Algorithm I finds all the discontinuities of the functions $C_n^*(d)$, $n \in N$, that is, all optimal paths from s to any node

ALGORITHM I *Inputs:* Graph G with link costs c_{ij} and delays d_{ij}. *Outputs*: The array $A_b[n]$ of queues, which contains the discontinuities of each node. /* begin initialization */

1) create_heap(H_a)
2) **for all** $n \in N - \{s\}${
3) $\quad A_b[n] = (-\infty, \infty, null, n)$;}
4) $A_b[s] = (0, 0, null, s)$;
5) **for all** $n \in N_-(s)${
6) $\quad e_a = (d_{sn}, c_{sn}, s, n)$;
7) $\quad$insert(e_a, H_a);} /* end initialization */
8) **while** $H_a = \varnothing$ **do {**
9) $\quad$get_min(e_a, H_a);
10) $\quad m = e_a$.*discontinuity_node;*
11) $\quad$tail($e_b, A_b[m]$);
12) $\quad$**if** (e_b.*cost*$>$ e_a.*cost*) **then{**
13) $\quad\quad$enqueue($e_a, A_b[m]$); }
14) $\quad$**for all** $n \in N_-(m)$, $n \neq e_a$.*predecessor*{
15) $\quad\quad e'_a = (e_a$.*delay*$+d_{mn}$, e_a.*cost*$+c_{mn}$, $m, n)$;
16) $\quad\quad$insert(e'_a, H_a);}}

Fig. 1. Algorithm I

$n \in N$ and for any possible delay. If we are interested only in finding a path from node s to a given node n, with delay at most $\bar{d}$, then the algorithm can be made to stop as soon as for function $C_n^*(d)$ a discontinuity is found whose delay is larger than or equal to $\bar{d}$. Below we prove the correctness of Algorithm I and analyze its computational complexity.

Correctness of Algorithm I

At the L^{th} iteration of the loop that begins on line 8, let $D_L = \cup_{n \in N} A_b[n]$. We will establish by induction that.

1) The L^{th} smallest discontinuity e_a in D is added to the appropriate queue $A_b[n]$.
2) The keys of the elements in heap H_a are larger than or equal to the keys of the elements in D_L.
3) The heap H_a contains all the discontinuities that may be successors of the discontinuities in D_L.

The statement is correct at the initialization step $L = 0$. Assume next that the statement is correct at step L. Let e_a be the element with smallest key in H_a and e_b the last element in $A_b[m]$, where $m = e_a$.*discontinuity_node*. If e_b.*cost*$\leq$ e_a.*cost*, then e_a cannot be a discontinuity of $C_m^*(d)$. To see this, note that by assumptions 1 and 2, we know that e_a.*delay*$\geq$ e_b.*delay*. Therefore, for e_a to represent a discontinuity of $C_m^*(d)$, by Lemma 2 we must have e_b.*cost*$>$ e_a.*cost*. Assume now that e_b.*cost*$>$ e_a.*cost*. Then, e_b.*delay*$<$ e_a.*delay* since otherwise the key of e_a will be (strictly) smaller than the key of e_b, which contradicts assumption 2. By assumption 3, there is a predecessor of e_a in D_L. There can be no discontinuity e'_a for $C_m^*(d)$ with delay smaller than e_a.delay. To see this, assume that such an e'_a exists. According to Lemma 2.b, there must be a path $p'_a = (m_1, m_2, ...m_k)$, $m_1 = s$ $m_k = e'_a$.*discontinuity_node*, such that 1) $C(p) = e'_a$.*cost*, $D(p) = e'_a$.*delay* and 2) path $p_a^l = (m_1, ..., m_l)$, $1 \leq l \leq k-1$ corresponds to a discontinuity smaller than or equal to e'_a. Let m_l be the smallest index node such that the discontinuity

e_a^l corresponding to $p_a^l = (m_1, ..., m_l)$ does not belong to D_L. The predecessor of e_a^l must belong to D_L and therefore, according to assumption 3, e_a^l must belong to H_a. Since e_a'.*delay*$\geq$ e_a^l.*delay*, and e_a.*delay*$>$ e_a'.*delay*, we conclude that e_a.*delay*$>$ e_a^l.*delay*, which contradicts the fact that e_a has the minimum key in H_a. Since by construction there is a path with delay and cost respectively e_a.*delay* and e_a.*cost*, from the discussion above we conclude that e_a represents a discontinuity of $C_m^*(d)$, $m = e_a$.*discontinuity_node* and assumption 1 is satisfied for $L+1$. Assumptions 2 and 3 are also satisfied since the insert operations in the **for all** loop, line 14,creates keys that are larger than or equal to e_a, as well as all the discontinuities that may be ancestors of e_a.

Computational Complexity of Algorithm I.

In the following we analyze the worst case running time of the algorithm provided that we intend to find all the discontinuities in D. A similar analysis holds for the worst case analysis when a bound $\overline{d}$ is specified on the delays. We express these bounds in terms of parameters that are revealing of the performance of the algorithms in the average case. If desired, these bounds can also be expressed in terms of node, link numbers and the delay bound.

Let $R(n)$ be the number of discontinuities of $C_n^*(d)$. Denote,

$$R_{\max} = \max_{n \in N} \{R(n)\},$$
$$R_S = \sum_{n \in N} R(n),$$
$$E = \sum_{m \in N} |N_-(m)| \, R(m).$$
$$N_{\max}^- = \max_{n \in N} \{N_-(n)\}.$$

Each discontinuity that is added to $A_b[m]$ inserts $|N_-(m)|$ elements to H_a. Therefore, E elements are inserted in H_a and its size is at most E. The get_min operation is executed once for each element of H_a. Since the get_min operation takes $O(\log E)$ time in the worst case, the worst case running time due to the get_min operation is $O(E \log E)$. The insert operation takes constant time and is executed E times. Hence the worst case running time of the algorithm is $O(E \log E)$.

We can express the worst case bound of the algorithm in terms of other relevant parameters as follows. Observe that

$$E \leq R_{\max} \sum_{n \in N} |N_-(n)| = R_{\max} |L|,$$

where $|L|$ is the cardinality of the set L. Hence the worst case performance of the proposed algorithm is,

$$O(R_{\max} |L| (\log |L| + \log R_{\max}))$$

Note that no integrality assumptions are made regarding the link delays. For comparison, assuming that the delays are positive integers and using the dynamic programming recursive equation in Lemma 1, the function $C_n^*(d)$ can be determined in the worst case running time, [6],

$$O(D_{\max} |L|),$$

where $D_{\max}$ is the maximum delay at which a discontinuity in D may occur. If delays can take zero values, then the worst case running time of the dynamic programming recursive equations becomes

$$O(D_{\max}(|L| + |N| \log |N|).$$

As we will see in Section III, numerical results show that for a wide range of networks, $D_{\max}$ is much larger than $R_{\max}$. As a result, the running time of the proposed algorithm is in general significantly better than the algorithm obtained by a direct application of the dynamic programming equation.

B. ALGORITHM II

The performance of Algorithm I presented in the previous section can be improved by a more efficient organization of the heap H_a containing all possible successors of the already known discontinuities. We present this approach in the current section.

Instead of the heap H_a we consider the following structures.

- An array $B[l]$ of queues,$l \in L$. An element e of $B[l]$ is of the form,

  $$e = (\textit{delay, cost, predecessor, discontinuity\_node}),$$

  where m=*predecessor* is the origin of link l , n=*discontinuity_node* is the destination of link l and (*delay,cost*) signifies a possible discontinuity of $C_n^*(d)$ with predecessor node m. As in the previous section, n and m are redundant here, but we keep them for simplicity in the description. Hence queue $B[l]$ contains all possible discontinuities of $C_n^*(d)$ that may be successors of the already known discontinuities of $C_m^*(d)$. The elements in $B[l]$ are stored in increasing order of keys, where key is the pair (*delay,cost*).
- An array of heaps $H_a[n]$. Heap $H_a[n]$ contains the head elements of the queues $B[(m,n)]$, $m \in N_+(n)$. An element e_a of $H_a[n]$ is of the same form as the elements in array$B[l]$.
- A heap H_g containing the minimum key elements of $H_a[n]$, $n \in N$. An element e_g of H_g is of the same form as the elements in array $B[l]$. Operations performed during the update of H_g ensure that each element e in H_g has smaller delay than any of the discontinuities in $H_a[n]$, $n = e$.*discontinuity_node*. This, combined with the correctness proof of Algorithm I, ensures that the minimum key element e_g in H_g is a real discontinuity for $C_n^*(d)$, $n = e_g$.*discontinuity_node.*

We also need the following subroutines

- obtain_minimum(e_g, H_g): this subroutine returns the minimum-key element e_g in H_g. At the same time, it updates $H_a[n]$, $n = e_g$.*discontinuity_node* and $B[l]$, $l = (m,n)$, $m \in N_+(n)$, and either removes or updates element e_g.
- update_new(e_a): This subroutine inserts a possible discontinuity e_a in queue $B[(m,n)]$, where $m = e_a$.*predecessor* and $n = e_a$.*discontinuity_node*. At the same time, $H_a[n]$ and H_g are updated.

The modified algorithm is presented in Figure 2. It is assumed without loss of generality that $N_+(s) = \varnothing$.

ALGORITHM II *Inputs:* Graph G with link costs c_{ij} and delays d_{ij}. *Outputs*: The array $A_b[n]$ of queues, which contains the discontinuities of each node. /* begin initialization */

create_heap(H_g);
$A_b[s] = (0, 0, null, s)$;
for all $l \in L$ **{**
    $B[l] = \varnothing$;**}**
for all $n \in N - \{s\}$**{**
    create_heap($H_a[n]$);
    $A_b[n] = (-\varnothing, \varnothing, null, n)$;**}**
for all $n \in N_-(s)$ **{**
    $e_a = (d_{sn}, c_{sn}, s, n)$;
    update_new(e_a);} /* end initialization */
while $H_g \neq \varnothing$ **do {**
    obtain_minimum(e_g, H_g);
    $m = e_g$.*discontinuity_node*;
    enqueue($e_g, A_b[m]$);
    for all $n \in N_-(m)$**{**
        $e'_a = (e_g.\mathit{delay}+d_{nm},\ e_g.\mathit{cost}+c_{nm},\ m, n)$;
        update_new(e'_a) ;**}}**

Fig. 2. Algorithm II

subroutine update_new(e_a) *Inputs:* Element e_a, the arrays B, H_a,A_b and the heap H_g.

1) $n = e_a$.*discontinuity_node*; $m = e_a$.*predecessor*;
2) $l = (m, n)$;
3) tail($e_t, A_b[n]$);
4) **if** e_t.*cost* $\leq e_a$.*cost***then { return; }**
5) **if** $H_a[n] = \varnothing$ **then{**
6) enqueue($e_a, B[l]$); insert($e_a, H_a[n]$);
7) insert(e_a, H_g); **return;}**
8) find_min($e_{\min}, H_a[n]$);
9) **switch**
10) **case:**$e_{\min}$.*delay* $\leq e_a$.*delay* **and** $e_{\min}$.*cost* $\leq e_a$.*cost***{return;}**
11) **case:**$e_{\min}$.*delay*$< e_a$.*delay***and** $e_{\min}$.*cost* $> e_a$.*cost***{**
12) **if** $B[l] = \varnothing$ **then** {insert($e_a, H_a[n]$);**}**
13) enqueue($e_a, B[l]$);**}**
14) **case**:$e_{\min}$.*delay* $> e_a$.*delay* **and** $e_{\min}$.*cost* $< e_a$.*cost***{**
15) enqueue($e_a, B[l]$); insert($e_a, H_a[n]$);
16) decrease_key($e_a, e_{\min}, H_g$);**}**
17) **case:(**$e_{\min}$.*delay* $= e_a$.*delay* **and** $e_{\min}$.*cost* $> e_a$.*cost*) **or** ($e_{\min}$.*delay*$> e_a$.*delay* **and** $e_{\min}$.*cost*$\geq e_a$.*cost*)**{**
18) decrease_key($e_a, e_{\min}, H_a[n]$);
19) decrease_key($e_a, e_{\min}, H_g$);
20) enqueue($e_a, B[l]$);
21) $k = e_{\min}$.*predecessor_node;*
22) $l = (k, n)$;
23) dequeue($e_{\min}, B[l]$);
24) **while** $B[l] \neq \varnothing$ **do{**
25) head($e, B[l]$);
26) **if** e.*cost* $< e_a$.*cost* **then {**
27) insert($e, H_a[n]$); **return;}**
28) **else {**dequeue($e, B[l]$);**}}**
29) **return;**

Fig. 3. Subroutine update_new

The initialization of $A_b[n]$ with a single element $(-\varnothing, \varnothing, null, n)$ is made in order to simplify the description of the code.

It is instructive at this point to compare Algorithm II with Dijkstra's algorithm [1, page 109]. In the latter algorithm, at each step there is a set $S \subseteq N$ that consists of the nodes whose minimum distance from the source node is known, and there is a label associated with each node representing the shortest distance path from the node to the source, *provided that* only the nodes in S can be used as intermediate nodes in a path. At each step of the algorithm, a node n in $\overline{S} = N - S$ with smallest label is moved to S and the labels of all outgoing neighbors of n are updated.

There is a direct correspondence between Dijkstra's algorithm and Algorithm II as follows

- S corresponds to the union of the elements in $A_b[n]$, $n \in N$.
- N corresponds to the set D of discontinuities of the functions C_n^* (d)
- The label of node n corresponds to the set of possible discontinuities of n that are located in $B((m, n))$, $m \in N_+(n)$.

In this sense, we may say that the proposed algorithm is a generalization of Dikjstra's algorithm.

We present the subroutine update_new in Figure 3, and we describe the various steps of the pseudocode.

If the cost of e_a is larger than or equal to the cost of the tail element e_t in $A_b[n]$ (line 4), then e_a is not a possible discontinuity and therefore it is discarded. If the cost of e_a is smaller than the cost of e_t, then, as with Algorithm I, it is known that e_a.*delay* $>e_t$.*delay*. If the heap $H_a[n]$ is empty, then the new element is inserted in $B[l]$, $H_a[n]$ and H_g (lines 5 to 7) and the subroutine ends. Else, (lines 8 and below) the key of e_a is compared with the minimum-key element $e_{\min}$ in $H_a[n]$ in order to decide whether e_a can be a possible discontinuity of $C_n^*(d)$ and whether $e_{\min}$ should be updated and certain possible discontinuities can be discarded. Specifically,

1) If $e_{\min}$.*delay* $\leq e_a$.*delay* **and** $e_{\min}$.*cost* $\leq e_a$.*cost* (line 10)**,** then e_a cannot be a possible discontinuity of $C_n^*(d)$ and therefore it is discarded.
2) If ($e_{\min}$.*delay*$< e_a$.*delay* **and** $e_{\min}$.*cost* $> e_a$.*cost*) (line 11), then e_a is a possible discontinuity and in case $B[l]$ is empty, e_a must be inserted in $H_a[n]$. Moreover, e_a is added to $B[l]$.
3) If ($e_{\min}$.*delay* $> e_a$.*delay* **and** $e_{\min}$.*cost* $< e_a$.*cost*) (line 14), then e_a is a possible discontinuity. Since the key of e_a is smaller than the key of $e_{\min,}$ $B[l]$ must be empty (the key of e_a is always larger than the key of the head element in $B[l]$ which in turn is larger than or equal to the key of $e_{\min}$). Therefore e_a must be added to$B[l]$, must be inserted in $H_a[n]$ and must replace $e_{\min}$ in H_g.
4) If ($e_{\min}$.*delay* $= e_a$.*delay* **and** $e_{\min}$.*cost* $> e_a$.*cost*) or

subroutine obtain_minimum(e_g, H_g) *Inputs:* The arrays B, $H_a$and the heap H_g. *Outputs:* The minimum key element e_g in H_g.

1) find_min(e_g, H_g);
2) $\hat{e}_g = e_g$;
3) $n = e_g$.*discontinuity_node*; $m = e_g$.*predecessor*;
4) $l = (m, n)$;
5) dequeue($e_g, B[l]$);
6) get_min($e_g, H_a[n]$);
7) **if** $(B[l] \neq \varnothing)$ **then {**
8)   head($e_h, B[l]$);
9)   insert($e_h, H_a[n]$);**}**
10) **while** $(H_a[n] \neq \varnothing)$ **do{**
11)   find_min($e_{\min}, H_a[n]$);
12)   **if** e_g.*delay*$<$ $e_{\min}$.*delay* and e_g.*cost* $>$ $e_{\min}$.*cost* **then {**
13)   increase_key($e_{\min}$, e_g, H_g);
14)   **return** $\hat{e}_g$**;}**
15)   **else{**$k = e_{\min}$.predecessor;
16) $l = (k, n)$;
17)   dequeue($e_{\min}, B[l]$);
18)   get_min($e_{\min}, H_a[n]$) ;
19)   **while** $(B[l] \neq \varnothing)$ **do{**
20)   head($e, B[l]$);
21)   **if** e.cost $<$ e_g.cost **then {**
22)   insert($e, H_a[n]$);
23)   increase_key(e, e_g, H_g);
24)   **return** $\hat{e}_g$**;}**
25)   **else {**dequeue($e, B[l]$);**}}}}**
26) get_min(e_g, H_g);
27) **return** $\hat{e}_g$;

Fig. 4. Subroutine obtain minimum

($e_{\min}$.*delay* $>$ e_a.*delay* **and** $e_{\min}$.*cost* $\geq$ e_a.*cost* (line 17) then e_a is a possible discontinuity while $e_{\min}$ is not. Therefore, e_a replaces $e_{\min}$ both in $H_a[n]$ and in H_g (lines 18, 19) and $e_{\min}$ is removed from $B((k, n))$, $k = e_{\min}$.*discontinuity_node* (line 23). Next, (line 24) the queue $B[(n, k)]$, is scanned and all impossible discontinuities in this queue are removed.

Finally, we present the obtain_minimum(e_g, H_g) subroutine in Figure 4.

Initially, the minimum-key element e_g in H_g is obtained. Next, e_g is dequeued from queue $B[l]$ and $H_a[n]$, where $l = (m, n)$, $n = e_g$.*discontinuity_node*, $m = e_g$.*predecessor*, and if $B[l]$ is not empty, the head element in $B[l]$ is inserted in $H_a[n]$. The rest of the pseudocode (starting from line 10) determines the element in $H_a[n]$ that is going to replace e_g and removes from queues $B[(k, n)]$, $k \in N_+(n)$ impossible discontinuities. The minimum-key element $e_{\min}$ in $H_a[n]$ is determined. In order to be inserted to H_g, $e_{\min}$ must have strictly larger delay and strictly smaller cost than e_g. If this is true (line 12), then $e_{\min}$ is inserted in H_g in place of e_g and the subroutine ends. Otherwise, since it is always true that the keys of the elements in $H_a[n]$ are larger than of equal to the key of e_g, the only possibilities are that e_g.*delay*$\leq$ $e_{\min}$.*delay* and e_g.*cost* $\leq$ $e_{\min}$.*cost*. In such a case, $e_{\min}$ cannot be a possible discontinuity of $C_n^*(d)$ and must be removed. The rest of the code in the **while** loop performs this removal and at the same time removes impossible discontinuities from $B[(k, n)]$, where $k = e_{\min}$.*predecessor*. On exit from the **while** loop (line 26), $H_a(n)$ must be empty and therefore e_g is removed from H_g instead of being replaced with another element in $H_a(n)$ as was done in lines 13, 23.

Computational Complexity of Algorithm II

We use the same notation as in Section II-A. Subroutine enqueue($e_g, A_b[m]$) takes $O(1)$ time and is executed once for each discontinuity in D. Subroutine obtain_minimum() is invoked once for each discontinuity in D, that is R_S times. Each such invocation involves a get_min or increase_key operation on H_g, which takes $O(\log |N|)$ time in the worst case. Hence, the worst case running time for all these operations is $O(R_S \log |N|)$. There are also other computations involving $H_a[n]$ and $B[l]$, which are taken into account below.

Subroutine update_new() is invoked once per discontinuity in D. Each invocation causes $|N_-(m)|$ updates, to $H_a[m]$ and $B[(m, n)]$ corresponding to the outgoing neighbor m of the node $n = e_a.discontinuity\_node$. These latter updates involve in the worst case an enqueue and dequeue operation to one of the queues in $B[l]$, a get_min, insert, or decrease_key operation on $H_a[m]$ and decrease_key operation on H_g. The get_min operations on $H_a[m]$ take worst-case time $O(\log |N_+(m)|)$ since the size of $H_a[m]$ is at most $|N_+(m)|$, while the rest of the operations take worst-case time $O(1)$. Hence, the total worst case running time of the algorithm is

$$O\left(R_S \log |N| + \sum_{n \in N} R(n) \sum_{m \in N_+(n)} \log \left| N_+(m) \right| \right).$$

Since

$$R_S \leq |N| \, R_{\max},$$

and, denoting $N_{\max}^+ = \max_{m \in N} \{N_+(m)\}$,

$$\sum_{n \in N} R(n) \sum_{m \in N_+(n)} \log |N_+(n)| \leq$$

$$\leq R_{\max} \log \left(N_{\max}^+ \right) \sum_{n \in N} \sum_{m \in N_+(n)} 1 =$$

$$= R_{\max} \log \left(N_{\max}^+ \right) |L| ,$$

we can express the previous bound as

$$O\left(R_{\max} \left(|N| \log |N| + |L| \log N_{\max}^+ \right) \right). \quad (5)$$

Note that if all delays are zero, then we have in effect the unconstrained shortest path routing problem. In this case, each $C_n^*(d)$ has a single discontinuity at 0, and $H_a[n]$ never has more than a single element. That means that the factor $\log N_{\max}^+$ can be removed from (5) and therefore the worst-case running time in this case becomes

$$O\left(|N| \log |N| + |L| \right),$$

TABLE I

WORST CASE RUNNING TIMES

DP	$O\left(D_{\max}(\vert L\vert + \vert N\vert \log \vert N\vert)\right)$
ALG I	$O(R_{\max} \vert L\vert (\log \vert L\vert + \log R_{\max}))$
ALG II	$O\left(R_{\max}(\vert N\vert \log \vert N\vert + \vert L\vert \log N^{+}_{\max})\right)$

that is, identical to the worst-case running time of Dijkstra's algorithm. This is to be expected since in this case Algorithm II reduces in effect to Dijkstra's algorithm.

In table I we summarize the worst-case running times of Algorithms I, II (ALG I and ALG II respectively) and the algorithm that results from the direct application of the dynamic programming equation. (DP)

III. NUMERICAL RESULTS

We run three sets of experiments. Each set corresponds to different methods of network generation, as follows.

Uniform Networks: A number $|N|$ of nodes and a number $|L| = \alpha |N|$ of edges, $a > 1$ is chosen. We use the graph generator *random_graph( )* from the LEDA package [12]. A random edge is generated by selecting a random element from a candidate set *C* defined as follows:

- *C* is initialized to the set of all $|N|(|N|-1)$ pairs (u, w) of distinct nodes.
- Upon a selection of a pair (u, w) from *C,* the pair is removed from *C*.

For each edge, a delay is picked randomly with uniform distribution among the integers $[1, 100]$.

Power Law Networks: A number $|N|$ of nodes and a number $|L| = \alpha |N|$, $\alpha > 1$ of links are chosen. The $|L|$ links are used to connect nodes randomly in such a manner that the node degrees follow a power law [16]. This is one of the methods that attempt to generate network topologies that are "Internet like". The nodes are placed randomly on a grid and the link delays are taken to be proportional to the distance between the nodes joined by the link under consideration.

Real Internet Network Topology: This network topology was taken from [17] and is based on the network topology observed in *01/02/2000*. For each edge, a delay is picked randomly with uniform distribution among the integers $[1, 100]$.

For all the experiments, link costs are generated by one of the following methods.

COST 1: A cost c_l for link l is picked randomly with uniform distribution among the integers $[1, 100]$.

COST 2: A parameter σ_l for link l is picked randomly with uniform distribution among the integers $[1, 5]$. The cost for the link under consideration is then $c_l = \sigma_l(101 - d_l)$. This method of cost generation reflects the the situation where link costs are decreasing as link delays are increasing.

For each of the experiments, we determine all discontinuities of the functions $C^*_n(d)$, $n \in N$, that is, in effect we find all optimal paths from a node s to all nodes in the network, under any possible delay constraint. We denote by $D_{\max}$ the maximum delay at which discontinuities in D occur. If one is interested only in finding a path with delay at most d, then for the given network, $D_{\max}$ corresponds to the delay d that causes the longest running time for all three algorithms tested.

We generated random uniform and power law networks with 400, 800, and 1200 nodes, and with ratios $\alpha = |L| / |N|$ equal to 4, 8 16, which are close to or larger than the ratios between 3 and 4 commonly found in today's networks. For each $|N|$ and α, we generated 10 random networks. For each of the generated networks we created link costs according to COST 1 and COST 2. In tables II -V we present the parameters of the generated networks. The parameter values are the averages of the values obtained for each of the 10 random networks. We observe that in general $R_{\max}$ is much smaller than $D_{\max}$. For the same method of link cost generation, the variations of parameter values for random and power law networks, (with the exception of $N^{+}_{\max}$) are not significant. However, networks where the COST 2 method is used for link cost generation generally have larger $R_{\max}$ and $D_{\max}$ than corresponding networks where the COST 1 method is used. As we will see, these observations have an effect on the performance of all three algorithms considered.

The tested Real Network consisted of 6474 nodes with 12572 bidirectional links, therefore $a = 3.88$ (2x12572/6474). Link delays are again picked randomly with uniform distribution among the integers $[1, 100]$. Both COST 1 and COST 2 methods for generating link costs were tested. In this network, we performed 10 experiments, where in each experiment a node is picked randomly to represent the source. The quantity $D_{\max}$ is defined in the same way as with the randomly generated networks. Table VI shows the relevant parameters in these experiments.

The experiments were run on a Pentium PC IV, 1.7GHz. The average running times (in seconds) for the three algorithms and for the various experiments are shown in tables VII - XI. Also, in these tables we present the ratio of the average running time of the DP algorithm to the average running time of ALG I and ALG II. The following observation are worthwhile.

- For the same link cost generation method, the variations of running times of a given algorithm for Random and Power Law networks with fixed $|N|$ and α are not significant.
- For a given algorithm, link costs generated by method COST 2 induce longer running times than link costs generated by method COST 1, for fixed $|N|$ and a.
- The performance of ALG I is comparable to ALG II for $\alpha = 4$, however for larger values the running time of ALG II can be about two times shorter than that of ALG I.
- For all experiments the running times of ALG I and ALG II are significantly better than DP (18 to 100 times shorter).

It is also worth noting that the performance of DP algorithm depends on the desired accuracy of link delays, while this is not true for both ALG I and ALG II. For example, we picked link delays between 1 and 100, which implies that we require two-digit accuracy. If the required accuracy is increased, then both $D_{\max}$ and $R_{\max}$ may increase, but the increase in $R_{\max}$ is much slower, and therefore the difference in the running times of the algorithms becomes even more significant. To

TABLE II

PARAMETERS FOR RANDOM UNIFORM NETWORKS, COST 1.

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
$D_{\max}$	679.7	679.1	558.9	810.1	767.3	695	823.7	743.8	750
$R_{\max}$	11.7	16.1	18.8	13.3	17.5	21.4	13.7	18.5	21.6
$N^{+}_{\max}$	10.9	16.8	28.1	12.1	19.2	29.7	12.3	18.9	30.4

TABLE III

PARAMETERS RANDOM POWER LAW NETWORKS, COST 1

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
$D_{\max}$	516.3	485	453	591	631	579	609	623	590
$R_{\max}$	12.9	17	18	14	18	22	15	20	23
$N^{+}_{\max}$	48	68	94	68	98	141	83	129	178

TABLE IV

PARAMETERS FOR RANDOM UNIFROM NETWORKS, COST 2.

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
$D_{\max}$	899.8	1027.2	1073	1186.1	1188.5	1229	1233	1256	1280
$R_{\max}$	19.4	35	61.9	23.3	45.5	73.8	26	45.6	74.8
$N^{+}_{\max}$	10.9	16.8	28.1	12.1	19.2	29.7	12.3	18.9	30.4

TABLE V

PARAMETERS RANDOM POWER LAW NETWORKS, COST 2

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
$D_{\max}$	710	833	839	862.8	924	975	960	1026	1067
$R_{\max}$	21.7	37.8	86.4	27.2	42	73	32	63	78
$N^{+}_{\max}$	48	68	94	68	98	141	83	129	178

TABLE VI

PARAMETERS FOR REAL INTERNET TOPOLOGY (6474 NODES, 25144 LINKS).

	COST 1	COST 2
$D_{\max}$	776	1001
$R_{\max}$	26	93
$N^{+}_{\max}$	1458	1458

TABLE VII

AVERAGE RUNNING TIMES FOR RANDOM UNIFROM NETWORKS, COST 1

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
ALG I	0.0593	0.214	0.625	0.146	0.495	1.52	0.225	0.812	2.44
ALG II	0.0579	0.1485	0.3296	0.145	0.34	0.789	0.21	0.545	1.25
DP	5.875	11.029	17.56	14.66	26.08	47.16	22.8	38.65	76.58
DP/ALG I	99.07	51.5	28.1	99.88	52.68	31	101.4	47.6	31.3
DP/ALG II	101.4	74.27	53.29	100.84	76.7	59.7	108.5	70.8	61.2

TABLE VIII

AVERAGE RUNNING TIMES FOR RANDOM POWER LAW NETWORKS, COST 1

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
ALG I	0.049	0.183	0.529	0.125	0.457	1.348	0.208	0.765	2.176
ALG II	0.065	0.143	0.298	0.154	0.354	0.731	0.253	0.583	1.189
DP	4.35	7.56	13.58	10.5	20.96	37.17	16.47	31.5	58.13
DP/ALG I	87.3	41.2	25.6	83.9	45.7	27.5	79.2	41.1	26.7
DP/ALG II	66.4	52.5	45.5	67.8	59	50.8	65	54	48.8

TABLE IX

AVERAGE RUNNING TIMES FOR RANDOM UNIFORM NETWORKS, COST 2

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
ALG I	0.0845	0.3858	1.276	0.223	1.01	3.2	0.357	1.46	5.21
ALG II	0.0826	0.2577	0.6188	0.2124	0.58	1.56	0.329	0.947	2.48
DP	7.91	17.098	34.88	21.84	41.1	83.63	34.8	66.9	135
DP/ALG I	93.6	44.31	27.32	97.9	40.6	26	97.4	45.6	25.9
DP/ALG II	95.77	66.35	56.37	102.8	70.8	53.	105.7	70.6	54.4

TABLE X

AVERAGE RUNNING TIMES FOR RANDOM POWER LAW NETWORKS, COST 2

$\|N\|$	400			800			1200		
α	4	8	16	4	8	16	4	8	16
ALG I	0.077	0.346	1.11	0.195	0.854	2.99	0.335	1.48	4.84
ALG II	0.101	0.262	0.583	0.346	0.617	1.53	0.393	1.057	2.5
DP	6.12	13.4	26.1	15.6	31.2	63.89	26.6	53.3	108
DP/ALG I	78.6	38.7	23.5	80	36.5	21.3	79.3	35.8	22.3
DP/ALG II	60.3	51.1	44.7	45	50.5	41.7	67.6	50.4	43.2

TABLE XI

AVERAGE RUNNING TIMES FOR REAL INTERNET NETWORK

	COST 1	COST 2
ALG I	2.641	8.188
ALG II	2.765	8.531
DP	132.15	153.68
DP/ALG I	50	18.7
DP/ALG II	47.8	18

demonstrate this we run the following experiment. We generate a Uniform network with $|N| = 800$ nodes and $|L| = a\,|N| = 8 * 800 = 6400$ edges. In the first experiment we pick delays between $[1, 100]$ and in the second between $[1, 1000]$. In table XII we present the relevant parameters and the running times for the tested network. We observe that the running time of DP algorithm increases tenfold while the increases of the running times of ALG I and ALG II are insignificant .

IV. CONCLUSIONS

In this paper we addressed the QoS routing problem. We provided two algorithms for finding the optimal solution. The basic idea of the proposed algorithms consists in finding in an iterative fashion the discontinuities of the functions $C_n^*(d)$. The algorithms operate under nonnegative link costs and delays and do not require any integrality assumptions on the delays. Numerical results show that for a wide range of tested networks the proposed algorithm outperforms significantly the algorithm based on the direct implementation of the Dynamic Programming equations. The running time of the proposed algorithms is satisfactory even for relatively large network sizes. If worst case performance is also of concern, the algorithms can replace the DP recursions of the approximate polynomial-time algorithms proposed in [9] and [6], in order to improve their average running time. The algorithms can be extended to the case where multiple constraints on the paths exist (e.g., maximum delay and loss probabilities).

Acknowledgement 1: The code for generating Power Law

TABLE XII

PARAMETERS AND RUNNING TIMES FOR A RANDOM UNIFORM NETWORK, COST 2

$\|N\|$	800	
a	8	
$delay$	(1,100)	(1,1000)
$D_{\max}$	1425	16611
$R_{\max}$	42	45
$N^{+}_{\max}$	18	18
ALG I	0.797	0.922
ALG II	0.531	0.594
DP	49.26	599.64
DP/ALG I	61.8	650.3
DP/ALG II	92.7	1009.5

Networks was downloaded from site [16], and the data for the Real Network from [17]. We would like to thank the authors of these sites.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, 1993.

[2] David Blokh, Gregory Gutin. "An Approximation Algorithm for Combinatorial Optimization Problems with Two Parameters," IMADA preprint PP-1995-14, May 1995.

[3] S. Chen and C. Nahrstedt. "On Finding Multi-constrained Paths," International Journal of Computational Geometry and Applications, 1998.

[4] M.L. Fredman, and R.E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". Journal of the ACM, Vol. 34, 596-615, 1987

[5] M.R. Garey and D.S. Johnson. "Computer and Intractability: A guide to the theory of NP-completeness," Freeman , San Francisco, 1978.

[6] Ashish Goel,K.G. Ramakrishnan, D. Kataria, and D. Logothetis. "Efficient Computation of Delay-sensitive Routes from One Source to All Destinations," In Proceedings of IEEE INFOCOM'01, Anchorage, AK, April 2001.

[7] R. Hassin, "Approximation Schemes for the Restricted Shortest Path Problem," Mathematics of Operations Research, vol. 17(1), pp.36-42, February 1992.

[8] 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.

[9] Dean H. Lorenz, Danny Raz. "A Simple Efficient Approximation Scheme for the Restricted Shortest Path Problem." Operations Research Letters, vol. 28, No. 5, pp. 213-219, June 2001.

[10] D.H. Lorenz, A. Orda, D. Raz, and Y. Shavitt. "Efficient QoS Partition and Routing of Unicast and Multicast." In Proceedings IEEE/IFIP IWQoS, Pittsburgh, PA, June 2000.

[11] Q. Ma and P. Steenkiste, "Quality of Service Routing for Traffic with Performance Guarantees." IWQoS'97, May 1997.

[12] Kurt Mehlhorn, Stefan Naher, *Leda:A Platform for Combinatorial and Geometric Computing*, Cambridge University Press, 2000.

[13] Q. Sun and H. Langendorfer, "A New Distributed Routing Algorithm with End-to-end Delay Guarantee," IWQoS'97, May 1997.

[14] Z. Wang and J. Crowford, "QoS Routing for Supporting Resource reservation,"IEEE JSAC, vol.14(7), pp.1228-1234, September 1996.

[15] Q. Warburton, "Approximation of Pareto Optima in Multiple Objective Shortest Path Problems," Operations Research,vol. 35, pp. 70-79, 1987

[16] The Power Law Simulator, *http://*www.cs.bu.edu/brite.

[17] The Real Networks, *http://moat.nlanr.net/Routing/rawdata.*