# [1]Algorithms for Source-to-all Maximum Cost to Time Ratio Problem in Acyclic Networks

**Alexandra Makri** (makri@uiuc.edu)
**Diego Klabjan** (klabjan@uiuc.edu)
**Department of Mechanical and Industrial Engineering**
**University of Illinois at Urbana-Champaign**
**Urbana, IL**

## Abstract

The source-to-all maximum cost to time ratio problem is the problem of finding the maximum cost to time ratio path from a source node to every other node. The motivation comes from an application in large-scale linear programming. We present three algorithms for solving the problem. We give proofs of correctness and we analyze the running times. One of the algorithms is polynomial and the remaining two are pseudo-polynomial. We present extensive computational results on several networks.

*Subject classifications*: Networks/graphs/flow algorithms.

## 1. Introduction

We present three algorithms to solve the *source-to-all maximum cost to time ratio* (MCT) problem. Consider a directed network $D$=($N$, $A$) with $n = |N|$ nodes and $m = |A|$ arcs and let $s$ be a given node. Every arc $e \in A$ has a cost $a_e \in Z$ and a time $b_e \in Z$, where $b_e > 0$ for every $e \in A$. We define the cost and the time of a path in $D$ in the usual way as the sum of the costs or the times of the arcs in the path, respectively. The source-to-all maximum cost to time ratio problem is to find $\tau_i^*$ for every $i \in N \setminus \{s\}$, where

$$\tau_i^* = \max\{\frac{\sum_{e \in P} a_e}{\sum_{e \in P} b_e} \mid P \text{ is an } s\text{ - }i \text{ path}\}.$$

Note that the integrality of $a$ and $b$ is without loss of generality since we can always scale them. We also assume that for every node $i \in N \setminus \{s\}$ there is at least one $s$-$i$ path in $D$.

The application of MCT is in large-scale linear programming (LP), where every column $i$ corresponds to a path, denoted by $P(i)$, in an acyclic network and every row corresponds to a node in the network. Such LPs are typically solved by column generation, Barnhart et. al. (1998), i.e. in each iteration a smaller LP, called the subproblem, is solved and next columns are added to the subproblem and the procedure is repeated. For simplicity of discussion, we assume that the cost of a column is the sum of the positive costs $b$ along the corresponding path $P(i)$ in the network, and in addition, we assume that each entry of the constraint matrix is either 0 or 1. Makri and Klabjan (2002) define a score of a column $i$ as $\sum_{j \in P(i)} y_j \Big/ \sum_{j \in P(i)} b_j$, where $y$ is a dual optimal vector to the subproblem. Traditionally in the column generation algorithms columns with large

[1] Partially supported by a grant from the University of Illinois at Urbana-Champaign Campus Research Board.

$\sum_{j\in P(i)} y_j - \sum_{j\in P(i)} b_j$ are added to the subproblem, however it turns out that columns with high scores lead to faster objective value improvements of the subproblem and hence to fewer iterations in column generation. Therefore the goal is to obtain paths with high scores. If paths are generated by a depth-first search procedure, then a source-to-all maximum cost to time ratio algorithm can be used to efficiently prune the generation by providing upper bounds on the scores. For details see Makri and Klabjan (2002).

It is easy to see that given a node *i*, the decision version of computing $\tau_i^*$ is NP-complete on general networks. This can easily be shown by a transformation from the Hamiltonian path problem. To circumvent this obstacle and since the application of the problem is based on acyclic graphs, we assume that *D* is acyclic. We show in this paper that MCT is polynomial on acyclic graphs.

A related problem is the problem of finding the minimum cost to time ratio cycle, Ahuja et. al. (1993), which differs from MCT in the following. First, MCT does not deal with cycles but it considers paths, and second, we want to find a cost to time ratio path from a given node to any other node. Dasdan, Irani and Gupta (1998, 1999) and Dasdan (2001) list several algorithms for identifying a minimum cost to time ratio cycle in a network and they report computation results. In Section 2 we show how to use an algorithm for the minimum cost to time ratio cycle problem in solving MCT. We present three new algorithms that are based on algorithms for the minimum cost to time ratio cycle problem, namely Karp and Orlin (1981), Lawler (1976) and Cochet-Terasson et. al. (1998). For each algorithm, we give proof of correctness and a running time analysis. Dasdan, Irani and Gupta (1998) analyze the Howard's algorithm, called the *primal-dual* algorithm in this paper, from Cochet-Terasson et. al. (1998). They give a running time bound by using a cycle counting argument. In this paper we carry out a completely different analysis since acyclic networks do not have cycles.

Section 2 presents the algorithms. In Section 2.1 we present the primal-dual algorithm, in Section 2.2 we give a bisection algorithm, and in Section 2.3 we present a parametric longest path algorithm. Section 3 describes the computational experiments and results.

## 2. Algorithms

In the following, a *tree* is a subgraph of *D* with exactly one arc directed into each vertex of $N\backslash\{s\}$. Thus a tree is really an arborescence rooted at *s*. For every $i\in N$ we denote by $P^T(i)$ the unique *s*-*i* path in tree *T* and the *predecessor* of *i* in *T* is the tail of the unique arc directed into *i* in *T*.

Let $\tau$ be a rational number and *T* a tree. For each $e\in A$ we define $c_e = a_e - \tau b_e$ and let $d(i) = \sum_{e\in P^T(i)} c_e = \sum_{e\in P^T(i)} a_e - \tau \sum_{e\in P^T(i)} b_e$. It is well known that *T* is a longest path tree for weights *c* if and only if the longest path optimality condition $d(i)+c_e \le d(j)$ for every arc $e=(i,j)\in A$ holds. The following easy observation, called the *ratio path optimality condition*, is the basis of all our algorithms. If *T* is a longest path tree for weights *c*, then $\tau_i^*=\tau$ for all $i\in N$ such that $d(i) = 0$.

Note that on acyclic networks the shortest path problem can easily be solved by first topologically ordering the nodes and then scanning the nodes based on the obtained order, Ahuja et. al. (1993). (In our computational experiments the network is beforehand topologically sorted.) However, for generality, whenever we employ a shortest path algorithm we assume the Dijkstra's algorithm.

Given an algorithm for the minimum cost to time ratio cycle, we can use it as follows to solve MCT. Consider a node $i \in N \setminus \{s\}$ and let us add to the network an arc $e=(i,s)$ with $a_e = b_e = 0$. It is easy to see that the minimum cost to time ratio cycle in this modified network yields $\tau_i^*$. If we perform this reduction $n$-1 times, i.e. for every node $i \in N \setminus \{s\}$, we solve MCT. We are thankful to a reviewer for proposing this algorithm. Note that this simple reduction gives a polynomial algorithm for MCT if we use a polynomial algorithm for the minimum cost to time ratio cycle problem (see Dasdan, Irani and Gupta (1998, 1999) for a list of such algorithms). This algorithm allows using existing algorithms for the minimum cost to time ratio cycle problem but it has the drawback that it neglects the computed paths from previous nodes. The three algorithms presented next use the information from previously computed paths.

## 2.1 The Primal-Dual Algorithm

This algorithm has roots in the primal-dual simplex algorithm and it is also based on Howard's algorithm for the minimum cost to time ratio cycle problem, Cochet-Terasson et. al. (1998). At every iteration a tree $T$ is maintained and first for every node the cost to time ratio path based on the path in $T$ is computed. The largest ratio is denoted by $\tau$. Next, the longest path distances $d$ based on $T$ and weights $c_e = a_e - \tau b_e$ for all $e \in A$ are computed. These distances correspond to dual values of the dual problem to the longest path problem. If the longest path optimality criterion is met, then $T$ is the longest path tree with respect to weights $c$ and we use the ratio path optimality condition to fix some $\tau_i^*$. If there are arcs violating the longest path optimality condition, then $T$ is improved. The procedure is then repeated.

The algorithm is described in Figure 1. Given a tree $T$ we denote by $pred_i$ the predecessor of $i$ in $T$ and in every iteration $\overline{S}$ is the set of all nodes $i$ for which $\tau_i^*$ has not yet been computed. It is easy to see that $\tau$ computed in step 2 is an upper bound on $\max_{i \in N \setminus \{s\}} \{\tau_i^*\}$. The initial tree $T$ is the longest path tree with respect to weights $c_e = a_e - \tau b_e$ for every arc $e \in A$. In steps 6 and 7 we compute the new $\tau$. Step 8 computes the new longest path distances in $T$ and steps 10-16 improve the tree. If an arc $e=(i,j)$ violates the longest path optimality condition, the arc is added to the tree by updating the predecessor of $j$ and $d(j)$ is improved. These steps correspond to a single pass of the FIFO label-correcting algorithm for the longest path problem, Ahuja et. al. (1993). Note that if the tree is not modified in these steps, than $T$ is the longest path tree with respect to weights $c$ and in steps 18-21 we apply the ratio path optimality condition.

The algorithm is very simple to implement and each iteration requires at most $O(m)$ steps. Next we provide the proof of correctness and the running time analysis.

**Proof of correctness**

Unfortunately we are not able to analyze the primal-dual algorithm presented in Figure 1 but in what follows we consider a slight diversion of steps 9-16 shown in Figure 2. This modification has already been proposed in Dasdan, Irani and Gupta (1999). It is easy to see that after step 16a of the modified algorithm for every node $j$ we have $d`(j) = \max\{d(i) + c_{(i,j)} : (i, j) \in A\}$. However, the computed distances $\tilde{d}$ after step 16 of the original primal-dual algorithm from Figure 1 have the property $\tilde{d}(j) \geq \max\{d(i) + c_{(i,j)} : (i, j) \in A\}$.

```
Input: An acyclic network $D = (N, A)$
Output: $\tau_i^*$ for all $i \in N \setminus \{s\}$
1:  For all $i \in N$ compute $\overline{d}(i)$, $\underline{d}(i)$ corresponding to the longest, shortest $s$-$i$ path with
    respect to weights $a_e$, $b_e$, respectively.
2:  $\tau = \max_{i \in N} \frac{\overline{d}(i)}{\underline{d}(i)}$
3:  Let $c_e = a_e - \tau b_e$ for all $e \in A$ and use Dijkstra to obtain the longest path tree $T$ with
    respect to weights $c$.
4:  $\overline{S} = N \setminus \{s\}$
5:  while $\overline{S} \neq \emptyset$ do
6:      For all $i \in \overline{S}$ compute $x_i = \sum_{e \in P^T(i)} a_e \Big/ \sum_{e \in P^T(i)} b_e$.
7:      $\tau = \max_{i \in \overline{S}} \{x_i\}$
8:      Let $c_e = a_e - \tau b_e$ for all $e \in A$ and compute $d(i) = \sum_{e \in P^T(i)} c_e$ for all $i \in N$.
9:      $b$ = true
10:     for all $e = (i,j) \in A$ do
11:         if $d(i) + c_e > d(j)$ then
12:             $pred_j = i$
13:             $d(j) = d(i) + c_e$
14:             $b$ = false
15:         end if
16:     end for
17:     if $b$ = true then
18:         for all $i \in \overline{S}$ with $d(i) = 0$ do
19:             $\overline{S} = \overline{S} \setminus \{i\}$
20:             $\tau^*_i = \tau$
21:         end for
22:     end if
23: end while
```

**Figure 1.** The primal-dual algorithm

We first show that the two variants are identical if arcs are scanned based on the reverse topological order.

**Proposition 1.** *If in step 10 the arcs are scanned in the reverse topological order, then for every node i we have* $\widetilde{d}(i) = d`(i)$ *and therefore the variant given in Figure 2 is not needed.*

*Proof.* Consider a node *j*. Due to the scanning order of the arcs, when we scan this node in step 10, all of the current distances $\widetilde{d}(i)$ of nodes *i* with $(i,j)\in A$ are equal to $d(i)$. If steps 12-14 are never executed, then $\widetilde{d}(j) \le \widetilde{d}(i) + c_{(i,j)} = d(i) + c_{(i,j)}$ and therefore by definition in step 8 it follows that $\widetilde{d}(j) = \max\{d(i) + c_{(i,j)} : (i,j) \in A\}$. If the condition in step 11 is true for at least one arc, then it is easy to see that $\widetilde{d}(j) = \max\{\widetilde{d}(i) + c_{(i,j)} : (i,j) \in A\} = \max\{d(i) + c_{(i,j)} : (i,j) \in A\}$, which shows the statement.

To analyze the algorithm we consider two types of iterations. In a type I iteration *b*=true in step 17 and in a type II iteration *b*=false. During an iteration of type II the tree is updated, i.e. there is at least one node for which the predecessor has been changed. We first show by studying steps 9a-16a that $\tau$ decreases after a type I iteration and it increases in every other iteration.

```
9a:     $d`=d$, $b$=true
10a:    for each $e=(i,j)\in A$ do
11a:        if $d(i)+c_e > d`(j)$ then
12a:            $pred_j = i$
13a:            $d`(j) = d(i)+c_e$
14a:            $b$=false
15a:        end if
16a:    end for
```

**Figure 2.** Modified steps 10-16 of the primal-dual algorithm

We say that a tree $\widetilde{T}$ is obtained from *T* by a 1-*opt exchange* if $\widetilde{T} = T \cup \{f\} \setminus \{e\}$, where $e\in T$, $f\notin T$ and *e,f* have the common head, see Figure 3. Since *D* is an acyclic network, it is easy to see that if *T* is a tree, then $\widetilde{T}$ is a tree as well. These tree updates are already considered by Karp and Orlin (1981). We denote by $T \underset{(e,f)}{\longrightarrow} \widetilde{T}$ the 1-opt exchange pictured in Figure 3.

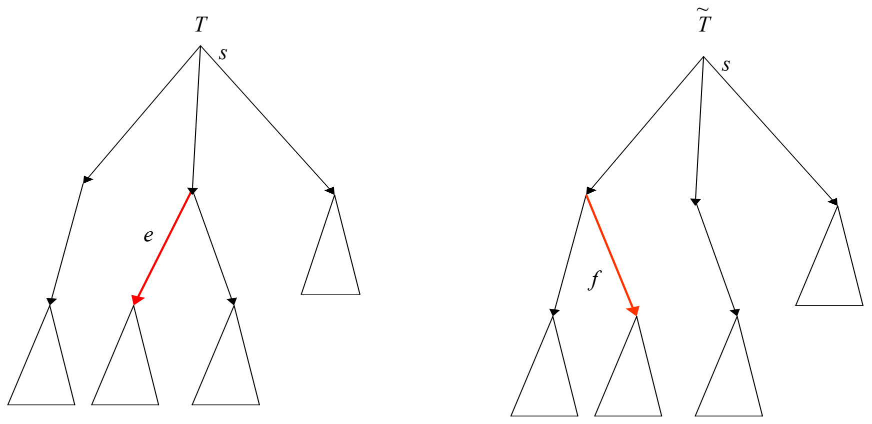

**Figure 3:** A 1-opt exchange

We can view steps 10a-16a as a sequence of 1-opt exchanges. We can write the tree $T'$ produces by steps 10a-16a as $T = T_1 \xrightarrow[(e_1,f_1)]{} T_2 \xrightarrow[(e_2,f_2)]{} T_3 \xrightarrow[(e_3,f_3)]{} \ldots \rightarrow T_g = T'$, where we denote by $T$ the tree before step 10a. We call such a sequence a 1-*opt representation of $T'$ from* $T$. Note that the 1-opt representation is not uniquely defined since there can be many 1-opt exchanges involving the same node. If $|\{e_1,\ldots,e_{g-1},f_1,\ldots,f_{g-1}\}|=2(g-1)$, i.e. all the arcs in the representation are different, then we call such a representation a *minimal* 1-*opt representation of $T'$*.

**Lemma 1**. *If $T'$ can be obtained from T by a sequence of 1-opt exchanges, then there exists a minimal 1-opt representation of $T'$. Any permutation of 1-opt exchanges in a minimal 1-opt representation is again a minimal 1-opt representation of $T'$.*

*Proof.* We denote by $\Phi$ the 1-opt representation of $T'$ from $T$. Let $pred_i^T$ be the predecessor of node $i$ in $T$ and let $pred_i^{T'}$ be the predecessor of node $i$ in $T'$. Let $k_1,\ldots,k_l$ be all the nodes with the property $pred_{k_i}^T \neq pred_{k_i}^{T'}$ for $i=1,\ldots,l$. Then

$$T = T_1 \xrightarrow[((pred_{k_1}^T,k_1),(pred_{k_1}^{T'},k_1))]{} T_2 \xrightarrow[((pred_{k_2}^T,k_2),(pred_{k_2}^{T'},k_2))]{} T_3 \rightarrow \ldots \rightarrow T_{l+1} = T'$$

is a 1-opt representation of $T'$ that is clearly minimal. This 1-opt representation is obtained from the original 1-opt representation $\Phi$ by considering for every node only the first and the last 1-opt exchange involving the node and therefore it yields $T'$.

It remains to be shown that any order of 1-opt exchanges in a minimal 1-opt representation yields again $T'$. It suffices to show that $T_3 = \overline{T}_3$, where $T_1 \xrightarrow[(e_1,f_1)]{} T_2 \xrightarrow[(e_2,f_2)]{} T_3$ and $T_1 \xrightarrow[(e_2,f_2)]{} \overline{T}_2 \xrightarrow[(e_1,f_1)]{} \overline{T}_3$ are 2 sequences of 2 arbitrary 1-opt exchanges**.** The general case then follows by induction. Since the two 2-opt exchanges are part of a minimal 1-opt representation, it follows by definition that $e_1 \neq f_2$ and $e_2 \neq f_1$. Then

$$T_2 = T_1 \cup \{f_1\} \setminus \{e_1\},\ \overline{T}_2 = T_1 \cup \{f_2\} \setminus \{e_2\}$$

$$T_3 = T_2 \cup \{f_2\} \setminus \{e_2\},\ \overline{T}_3 = \overline{T}_2 \cup \{f_1\} \setminus \{e_1\}.$$

Using elementary calculus it is easy to see that $T_3 = \overline{T}_3$.

For an arbitrary tree $\widetilde{T}$, general weights $c$ on arcs, and for any node $i \in N$ we denote $d_c^{\widetilde{T}}(i) = \sum_{e \in P^{\widetilde{T}}(i)} c_e$. To show the finiteness of the algorithm we need the following two claims.

**Claim 1**. *Let T be the tree before step 10a and let $T'$ be the tree after step 16a. Then for all $i \in N$ we have* $d_a^{T'}(i) = d_a^T(i) + \sum_{(u,v) \in P^{T'}(i) \setminus T} \Delta_1^T(u,v)$, *where* $\Delta_1^T(u,v) = d_a^T(u) - d_a^T(v) + a_{(u,v)}$.

In linear programming terminology, this claim states that the new dual prices of node $i$ can be obtained from old dual prices by adding the sum of the reduced cost of all the arcs along the path from $s$ to $i$ in the new tree $T'$ that are not present in the original tree $T$. The statement is very intuitive if the arcs are added to $T$ in the order based on the path from $s$ to $i$ in $T'$. However this might not be the case in the algorithm.

*Proof.* Let $T = T_1 \underset{(e_1,f_1)}{\to} T_2 \underset{(e_2,f_2)}{\to} T_3 \underset{(e_3,f_3)}{\to} \dots \to T_g = T'$ be a minimal 1-opt representation, which exists by Lemma 1. We prove the claim by induction on $g$. Given $v \in N$ and a tree $\widetilde{T}$, let $\widetilde{T}(v)$ denote the subtree of $\widetilde{T}$ rooted at $v$. Thus $\widetilde{T} = \widetilde{T}(s)$. It is easy to see that if $\widetilde{T} \underset{((k,l),(r,l))}{\to} \hat{T}$, then

$$d_a^{\hat{T}}(i) = \begin{cases} d_a^{\widetilde{T}}(i) & i \notin \widetilde{T}(l) \qquad (1) \\ d_a^{\widetilde{T}}(i) + \Delta_1^{\widetilde{T}}(r,l) & i \in \widetilde{T}(l). \qquad (2) \end{cases}$$

Using this property the claim for $g$=2 follows.

Suppose now that the claim holds for all trees obtained from $T$ by a sequence of 1-opt exchanges having a minimal 1-opt representation with at most $g$-1 1-opt exchanges, and for any $i \in N$. Assume that $T_{g-1} \underset{((k,l),(r,l))}{\to} T_g$ for $(k,l) \in E$, $(r,l) \in E$, and let $i$ be an arbitrary node. If $i \notin \widetilde{T}(l)$ then

$$d_a^{T_g}(i) = d_a^{T_{g-1}}(i) = d_a^T(i) + \sum_{(u,v) \in P^{T'}(i) \setminus T} \Delta_1^T(u,v),$$

where the first equality holds by (1) and the second by induction since $P^{T_g}(i) = P^{T_{g-1}}(i)$.

Let now $i \in \widetilde{T}(l)$. Consider the arc $(p,q) \in P^{T_g}(i) \setminus T$, which is the closest arc to $i$ in $P^{T_g}(i)$ with this property, see Figure 4. Thus $P^{T_g}(i) - P^{T_g}(q) \subseteq T$. If $(p,q) \neq (r,l)$,

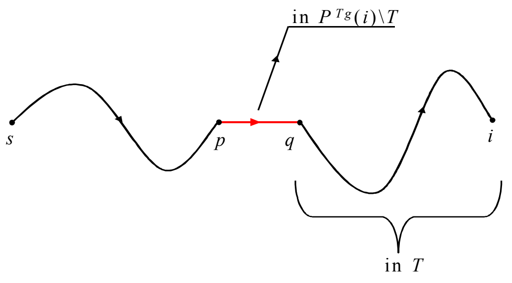

**Figure 4.** The definition of $(p,q)$

then by Lemma 1 we can change the order of the 1-opt exchanges to meet this requirement. Therefore we can assume that $(p,q)=(r,l)$. Now we have

$$d_a^{T'}(i)=d_a^{T_g}(i)=d_a^{T_{g-1}}(i)+\Delta_1^{T_{g-1}}(r,l)=d_a^{T_{g-1}}(i)+d_a^{T_{g-1}}(r)-d_a^{T_{g-1}}(l)+a_{(r,l)} \tag{3}$$

$$=d_a^{T_1}(i)+\sum_{(u,v)\in P^{T_{g-1}}(i)\backslash T_1}\Delta_1^{T_1}(u,v)+d_a^{T_{g-1}}(r)-d_a^{T_1}(l)-\sum_{(u,v)\in P^{T_{g-1}}(i)\backslash T_1}\Delta_1^{T_1}(u,v)+a_{(r,l)} \tag{4}$$

$$=d_a^{T_1}(i)+d_a^{T_{g-1}}(r)-d_a^{T_1}(l)+a_{(r,l)}$$

$$=d_a^{T_1}(i)+d_a^{T_1}(r)+\sum_{(u,v)\in P^{T_{g-1}}(r)\backslash T_1}\Delta_1^{T_1}(u,v)-d_a^{T_1}(l)+a_{(r,l)} \tag{5}$$

$$=d_a^{T_1}(i)+\sum_{(u,v)\in P^{T_g}(i)\backslash T_1}\Delta_1^{T_1}(u,v)=d_a^{T}(i)+\sum_{(u,v)\in P^{T'}(i)\backslash T}\Delta_1^{T}(u,v),$$

where (3) holds from (2), and (4) and (5) follow by induction and from $P^{T_g}(i)\backslash T_1=(P^{T_{g-1}}(r)\backslash T_1)\cup\{(r,l)\}, P^{T_{g-1}}(i)\backslash T_1=P^{T_{g-1}}(l)\backslash T_1$.

Given $(u,v)\in A$ and arbitrary weights $c$, let $\Delta_c^T(u,v)=d_c^T(u)-d_c^T(v)+c_{(u,v)}$, and for simplicity of notation we denote $\Delta_2^T(u,v)=\Delta_b^T(u,v)$. In addition we denote by $\tau^j$ the $\tau$ computed in step 7 of iteration $j$.

**Claim 2.** *Let i be a type I iteration and let l be the next type I iteration. Then*

1) $\tau^{i+1}\le\tau^i$,
2) $\tau^{j-1}\le\tau^j$ *for all* $j=i+2, \dots, l$,
3) *if* $\tau^{j-1}<\tau^j$ *for a* $j\in\{i+2, \dots, l\}$, *then* $\tau^j-\tau^{j-1}\ge\dfrac{1}{n^2\overline{B}^2}$, *where* $\overline{B}=\max_{e\in A}\{b_e\}$, *and*
4) *if* $\tau^j=\tau^{j+1}=\dots=\tau^k$, *where* $i<j<k<l$, *then* $k-j\le n$.

*Proof*. For any iteration $q$ let $T_q$ be the tree in steps 6-9.

Statement 1 holds since at the end of iteration $i$ the tree is optimal and steps 19 and 20 are evaluated at least for the node where the maximum is attained in step 7. Therefore in iteration $i$+1 the tree stays the same and the maximum in step 7 is over a smaller subset.

To prove statement 2 of the claim we assume that $x_k=\max_{i\in S}\{x_i\}$ in iteration $j$-1. Then

$$\tau^j\ge\frac{\sum_{e\in P^{T_j}(k)}a_e}{\sum_{e\in P^{T_j}(k)}b_e}=\frac{d_a^{T_j}(k)}{d_b^{T_j}(k)}=\frac{d_a^{T_{j-1}}(k)+\sum_{(u,v)\in P^{T_j}(i)\backslash T_{j-1}}\Delta_1^{T_{j-1}}(u,v)}{d_b^{T_{j-1}}(k)+\sum_{(u,v)\in P^{T_j}(i)\backslash T_{j-1}}\Delta_2^{T_{j-1}}(u,v)}, \tag{6}$$

where the second equality holds by Claim 1. By induction it is easy to see that from step 11a it follows $d`(p)\ge d(p)$ for every $p\in N$. Also, for every $p\in N$ after step 16a we have $d`(p)=\max\{d(q)+c_{(q,p)}:(q,p)\in A\}$. Combining these two observations we obtain that if $(u,v)\in T_j\backslash T_{j-1}$, then $d`(v)=d(u)+c_{(u,v)}>d(v)$ and therefore from step 8, for all arcs $(u,v)\in T_j\backslash T_{j-1}$ we have $d_c^{T_{j-1}}(u)+c_{(u,v)}>d_c^{T_{j-1}}(v)$, which in turn implies by definition of $c$

that $d_a^{T_{j-1}}(u) - \tau^{\text{j-1}} d_b^{T_{j-1}}(u) + a_{(u,v)} - \tau^{\text{j-1}} b_{(u,v)} >$ $d_a^{T_{j-1}}(v) - \tau^{\text{j-1}} d_b^{T_{j-1}}(v)$. This inequality can be rewritten as

$$\Delta_1^{T_{j-1}}(u,v) > \tau^{\text{j-1}} \Delta_2^{T_{j-1}}(u,v). \tag{7}$$

Using (6) and (7), and $\tau^{j-1} = d_a^{T_{j-1}}(k) \big/ d_b^{T_{j-1}}(k)$, we get

$$\tau^j \geq \frac{d_a^{T_{j-1}}(k) + \sum\limits_{(u,v)\in P^{T_j}(i)\setminus T_{j-1}} \Delta_1^{T_{j-1}}(u,v)}{d_b^{T_{j-1}}(k) + \sum\limits_{(u,v)\in P^{T_j}(i)\setminus T_{j-1}} \Delta_2^{T_{j-1}}(u,v)} \geq \frac{d_a^{T_{j-1}}(k) + \tau^{j-1} \sum\limits_{(u,v)\in P^{T_j}(i)\setminus T_{j-1}} \Delta_2^{T_{j-1}}(u,v)}{d_b^{T_{j-1}}(k) + \sum\limits_{(u,v)\in P^{T_j}(i)\setminus T_{j-1}} \Delta_2^{T_{j-1}}(u,v)} = \tau^{j-1},$$

which yields statement 2 of the claim.

Now we show statement 3 of the claim by using an argument from Lawler (1976). Suppose $\tau$ increases in iteration $j$. Then

$$\tau^j - \tau^{j-1} = \frac{\sum\limits_{e\in P^{T_j}(k)} a_e}{\sum\limits_{e\in P^{T_j}(k)} b_e} - \frac{\sum\limits_{e\in P^{T_{j-1}}(l)} a_e}{\sum\limits_{e\in P^{T_{j-1}}(l)} b_e} > 0,$$

where $k$ and $l$ are the nodes where $\max\limits_{i\in \bar{S}}\{x_i\}$ is attained in iterations $j$ and $j$-1 respectively. We have

$$\frac{\sum\limits_{e\in P^{T_j}(k)} a_e}{\sum\limits_{e\in P^{T_j}(k)} b_e} - \frac{\sum\limits_{e\in P^{T_{j-1}}(l)} a_e}{\sum\limits_{e\in P^{T_{j-1}}(l)} b_e} = \frac{\sum\limits_{e\in P^{T_j}(k)} a_e \sum\limits_{e\in P^{T_{j-1}}(l)} b_e - \sum\limits_{e\in P^{T_{j-1}}(l)} a_e \sum\limits_{e\in P^{T_j}(k)} b_e}{\sum\limits_{e\in P^{T_j}(k)} b_e \sum\limits_{e\in P^{T_{j-1}}(l)} b_e} \neq 0$$

and therefore

$$\left| \frac{\sum\limits_{e\in P^{T_j}(k)} a_e \sum\limits_{e\in P^{T_{j-1}}(l)} b_e - \sum\limits_{e\in P^{T_{j-1}}(l)} a_e \sum\limits_{e\in P^{T_j}(k)} b_e}{\sum\limits_{e\in P^{T_j}(k)} b_e \sum\limits_{e\in P^{T_{j-1}}(l)} b_e} \right| \geq \frac{1}{n^2 \overline{B}^2},$$

since the numerator is integral and nonzero.

Now we show the last statement. If $\tau^j = \tau^{j+1} = \ldots = \tau^k$, then in all these iterations the weights $c$ do not change. Let $p$, $j \leq p \leq k$-1 be an iteration and let $d`$ be computed after step 16a at iteration $p$.

We first show that $d_c^{T_{p+1}}(i) \geq d`(i)$ for any node $i$. Let $i$ be a node. We have already argued that $d`(i) \geq d_c^{T_p}(i)$. Assume first that $d`(i) = d_c^{T_p}(i)$. Then

$$d_c^{T_{p+1}}(i) = d_c^{T_p}(i) + \sum_{(u,v)\in P^{T_{p+1}}(i)\setminus T_p} \Delta_c^{T_p}(u,v) = d_c^{T_p}(i) + \sum_{(u,v)\in P^{T_{p+1}}(i)\setminus T_p} (\Delta_1^{T_p}(u,v) - \tau^p \Delta_2^{T_p}(u,v))$$
$$\geq d_c^{T_p}(i) = d`(i),$$

where we have used Claim 1 and (7). Let now $d`(i) > d_c^{T_p}(i)$. In this case $d`(i) = d_c^{T_p}(j) + c_{(j,i)}$ for an arc $(j,i) \in A$. Clearly $(j,i) \in P^{T_{p+1}}(i) \setminus T_p$ and therefore again by Claim 1 and (7) we obtain

$$d_c^{T_{p+1}}(i) = d_c^{T_p}(i) + \sum_{(u,v)\in P^{T_{p+1}}(i)\setminus T_p} \Delta_c^{T_p}(u,v) \geq d_c^{T_p}(i) + \Delta_c^{T_p}(j,i) = d_c^{T_p}(j) + c_{(j,i)} = d`(i).$$

For any $j \in N$, let $r^k(j)$ be the value of the longest $s$-$j$ path with exactly $k$ arcs and with respect to $c$. Clearly by dynamic programming we have $r^{k+1}(j) = \max\{r^k(i) + c_{(i,j)} : (i,j) \in A\}$. By induction and by combining this dynamic programming equation with $d`(j) = \max\{d_c^{T_p}(i) + c_{(i,j)} : (i,j) \in A\}$, we get that if $d_c^{T_p}(j) \geq r^k(j)$ for an integer $k$, then $d`(j) \geq r^{k+1}(j)$. Since $d`(j) \leq d_c^{T_{p+1}}(j)$, it follows that $d_c^{T_{p+1}}(j) \geq r^{k+1}(j)$. Since $k \leq n$, either we find an optimal longest path tree in at most $n$ iterations or $\tau$ changes. This proves the last statement in the claim.

Using Claim 2 it is easy to prove the correctness and finiteness of the algorithm. After an iteration of type I, $\tau$ decreases. In all consecutive iterations of type II $\tau$ either increases or it remains the same for at most $n$ iterations. After at most $n$ iterations either $\tau$ increases or the longest path tree is found and therefore $|\overline{S}|$ decreases. Now it is clear that the algorithm terminates in a finite number of steps.

Note that only in proving the fourth statement of Claim 2 we needed the variant given in Figure 2. Thus Claim 1 holds for the more general case and the variant is required only in the degenerate case when $\tau$ does not change from iteration to iteration.

**Running time**

Let $\overline{A} = \max_{e\in E}\{a_e\}$, $\underline{A} = \min_{e\in E}\{a_e\}$, $\underline{B} = \min_{e\in E}\{b_e\}$, and $\widetilde{A} = \max_{e\in E}\{|a_e|\}$. Consider two consecutive type I iterations $i$ and $l$, $i<l$ and let $x_j = \max_{i\in\overline{S}}\{x_i\}$ in iteration $i$+1. Then we have

$$\tau^l - \tau^{i+1} \leq \max_{i\in N\setminus\{s\}}\{\tau_i^*\} - \frac{\sum_{e\in P^{T_{i+1}}(j)} a_e}{\sum_{e\in P^{T_{i+1}}(j)} b_e} \leq \frac{n\overline{A}}{\underline{B}} - \frac{\underline{A}}{n\overline{B}} \leq \frac{2n\max\{|\overline{A}|,|\underline{A}|\}}{\underline{B}} = \frac{2n\widetilde{A}}{\underline{B}}. \tag{8}$$

At every type II iteration between iterations $i$ and $l$, either $\tau$ increases by at least $1/(n^2\overline{B}^2)$ or it remains the same for at most $n$ iterations. Therefore there are at most $n \cdot n^2\overline{B}^2 \cdot 2n\widetilde{A}/\underline{B}$ consecutive type II iterations. Since there are at most $n$ type I iterations and the time per iteration is $O(m)$, the overall running time is $O(n^5 m\overline{B}^2\widetilde{A}/\underline{B})$. Therefore the primal-dual algorithm is pseudo-polynomial.

**2.2 The Bisection Algorithm**

The main idea of the bisection algorithm is taken from the mean cost to time cycle algorithm by Lawler (1976). In the bisection algorithm, each $\tau_i^*$ has a lower and an upper bound and we use bisection to find the optimal $\tau_i^*$. At every iteration, the gap between the lower and the upper bound of the selected node is decreased by at least a half, but in addition, the gap of other nodes might decrease as well.

The bisection algorithm is described in Figure 5. In the initialization stage of the algorithm the upper and lower bounds $\bar{\tau}$ and $\underline{\tau}$ are computed as follows. For all $i \in N$ we compute $\bar{d}(i)$ and $\underline{d}(i)$ as in step 1 in Figure 1. Denote by $T_1$ the longest path tree induced by $\bar{d}$ and by $T_2$ the shortest path tree induced by $\underline{d}$. For every $i \in N$ we set $\bar{\tau}_i = \frac{\bar{d}(i)}{\underline{d}(i)}$ and $\underline{\tau}_i = \max\{\frac{d_a^{T_1}(i)}{d_b^{T_1}(i)}, \frac{d_a^{T_2}(i)}{d_b^{T_2}(i)}\}$, which is clearly an upper and a lower bound on $\tau_i^*$.

```
Input: An acyclic network $D = (N, A)$
Output: $\tau_i^*$ for all $i \in N \setminus \{s\}$
1:   Initialize lower bounds $\underline{\tau}$ and upper bounds $\bar{\tau}$.
2:   $\bar{S} = N \setminus \{s\}$
3:   while $\bar{S} \neq \emptyset$ do
4:       Select a node $j \in \bar{S}$.
5:       $\tau = \frac{\bar{\tau}_j + \underline{\tau}_j}{2}$
6:       Let $c_e = a_e - \tau b_e$ for all $e \in E$ and use Dijkstra to obtain the longest path distances $d$
         and the corresponding longest path tree $T$ with respect to weights c.
7:       for all $i \in \bar{S}$ do
8:           if $d(i) = 0$ then
9:               $\bar{S} = \bar{S} \setminus \{i\}$
10:              $\tau_i^* = \tau$
11:          else if $d(i) < 0$ then
12:              $\bar{\tau}_i = \min\{\tau, \bar{\tau}_i\}$
13:          else
14:              $\underline{\tau}_i = \max\{\frac{\sum_{e \in P^T(i)} a_e}{\sum_{e \in P^T(i)} b_e}, \underline{\tau}_i\}$
15:          end if
16:      end for
17:  end while
```

**Figure 5.** The bisection algorithm

In step 4 we first select a node from $\bar{S}$, and in steps 5 and 6 we perform bisection. In step 6 we run Dijkstra with respect to weights $c_e = a_e - \tau b_e$ for every arc $e \in A$, and we obtain the longest path values $d(i)$ for all $i \in N$ and the corresponding longest path tree $T$. Let $i \in \bar{S}$. If $d(i)=0$, then by the ratio path optimality condition we have $\tau_i^* = \tau$ and we remove the node from $\bar{S}$. This corresponds to steps 9 and 10. If $d(i)<0$, then for any *s-i* path $P$ in the network we have $\sum_{e\in P} a_e - \tau \sum_{e\in P} b_e \le d(i) < 0$ and therefore $\tau$ is an upper bound on $\tau_i^*$. If $\tau$ is a better upper bound, then in step 12 we update the upper bound. Finally, if $d(i)>0$, then $\sum_{e\in P^T(i)} a_e \Big/ \sum_{e\in P^T(i)} b_e > \tau$. Clearly $\sum_{e\in P^T(i)} a_e \Big/ \sum_{e\in P^T(i)} b_e$ provides a lower bound on $\tau_i^*$ and we update the lower bound in step 14.

**Proof of Correctness and the Running Time Analysis**

Let us denote $t_i = \overline{\tau}_i - \underline{\tau}_i$. To prove the correctness observe that for the selected node $j$ in step 4, the gap between the upper and the lower bound decreases by at least a half if $d(j)\neq 0$. Therefore in every iteration, either the gap of the selected node decreases by at least a half, or the node is removed from $\overline{S}$. By statement 3 of Claim 2, if $t_j \le 1\big/(n^2\overline{B}^2)$, then there is a unique $s$-$j$ path $P$ with $\sum_{e\in P} a_e \Big/ \sum_{e\in P} b_e \in [\underline{\tau}_j, \overline{\tau}_j]$, which is clearly optimal. The initial $t_j$ can be upper-bounded as in (8) by $2n\widetilde{A}/\underline{B}$ and we obtain $\tau_j^*$ if $t_j \le 1\big/(n^2\overline{B}^2)$. Since every time $j$ is selected in step 4, $t_j$ decreases by at least a half, overall $j$ can be selected at most $\log(\frac{2n\widetilde{A}/\underline{B}}{1/(n^2\overline{B}^2)}) = \log(\frac{2n^3\widetilde{A}\overline{B}^2}{\underline{B}})$ times. Since there are $n$ nodes and $O(m)$ steps per iteration (assuming we solve the shortest path problem in Step 6 by topologically sorting the nodes), the running time of the algorithm is $O(nm\log(\frac{n\widetilde{A}\overline{B}}{\underline{B}}))$. This algorithm is polynomial.

**Implementation**

In step 4 of the algorithm we do not specify which node to select among all the nodes in $\overline{S}$. We have performed computational experiments with the following node selection strategies.

1) Select a random node from $\overline{S}$, which yields a randomized algorithm.
2) Select always the first node in $\overline{S}$. Using this strategy, the execution flow depends on the initial order of nodes.
3) Select the node $j$ with the smallest $t_j$. The intuition here is that for the selected $j$ we are close to optimality.
4) Select the node $j$ with the largest $t_j$. By selecting such a node we hope that significantly changing $\tau$ would substantially improve the bounds for other nodes as well.
5) Select the node $j$ with the smallest $t_j$ and keep selecting $j$ until it is removed from $\overline{S}$. This strategy is a mixture of strategies 2 and 3.
6) Select the node $j$ with the largest $t_j$ and keep selecting $j$ until it is removed from $\overline{S}$. This strategy combines strategies 2 and 4.

Computational results have shown that strategy 4 outperforms the others and therefore it is the default node selection strategy.

Another implementation issue, which arises by using any of the strategies 3-6, is how to efficiently find the node with the largest or the smallest $t_j$ among all the nodes in $\overline{S}$. We have implemented the bucket approach first proposed by Dial (1976) in the context of the Dijkstra's shortest path algorithm. The nodes are stored into $C$ buckets depending on $t_j$ for $j \in \overline{S}$. Bucket $i$ contains nodes $j$ with $t_j \in \left[\underline{t} + il, \underline{t} + (i+1)l\right)$, where $\underline{t}$ is a lower bound on initial $t_i$ for all $i \in N$, $\overline{t}$ is an upper bound on initial $t_i$ for all $i \in N$, and $l = \frac{\overline{t} - \underline{t}}{C}$. To find the node with the maximum $t_j$, we only need to scan the buckets, starting from the last visited bucket, until the first nonempty bucket is found. After finding the first nonempty bucket, we scan all the nodes in this bucket to obtain the maximum $t_i$. Every time a bound of a node is updated, the node is removed from the bucket and is placed in the bucket corresponding to its new $t_j$. The bucket approach speeds up considerably the algorithm even though this implementation is no longer polynomial. In our implementation we use 200 buckets, i.e. $C$=200.

### 2.3 The Parametric Longest Path Algorithm

The parametric longest path algorithm is based on Karp and Orlin (1981). The following proposition is key to the understanding of the algorithm.

**Proposition 2.** *Let T be the longest path tree with respect to weights* $c_e = a_e - \overline{\tau} b_e$ *for all* $e \in A$. *Then T is the longest path tree with respect to weights* $c_e = a_e - \tau b_e$, $e \in A$ *for any* $\tau$ *such that*

$$\max_{\substack{e=(u,v)\in E \\ d_b^T(u) - d_b^T(v) + b_e > 0}} \left\{ \frac{d_a^T(u) - d_a^T(v) + a_e}{d_b^T(u) - d_b^T(v) + b_e} \right\} \leq \tau \leq \overline{\tau}.$$

*Proof.* Let $\tau$ be such that $\hat{\tau} = \max\left\{ \frac{d_a^T(u) - d_a^T(v) + a_e}{d_b^T(u) - d_b^T(v) + b_e} \right\} \leq \tau \leq \overline{\tau}$. We need to show that $T$ is the longest path tree with respect to weights $c_e = a_e - \tau b_e$ for all $e \in A$. By the longest path optimality conditions it suffices to show that $d_c{}^T(u) + a_e - \tau b_e \leq d_c{}^T(v)$ for any $e=(u,v) \in A$. This inequality is equivalent to

$$d_a^T(u) - d_a^T(v) + a_e \leq \tau(d_b^T(u) - d_b^T(v) + b_e). \tag{9}$$

If $\tau = \overline{\tau}$, then (9) holds by assumption.

Next we show that (9) holds for all $\tau$, $\hat{\tau} \leq \tau \leq \overline{\tau}$. Let $e=(u,v) \in A$ be an arc. Suppose first that $d_b^T(u) - d_b^T(v) + b_e \leq 0$. Then since $\tau \leq \overline{\tau}$ and since (9) holds for $\overline{\tau}$, we have

$$d_a^T(u) - d_a^T(v) + a_e \leq \overline{\tau}(d_b^T(u) - d_b^T(v) + b_e) \leq \tau(d_b^T(u) - d_b^T(v) + b_e).$$

If $d_b^T(u) - d_b^T(v) + b_e > 0$, then (9) holds by definition of $\hat{\tau}$ and since $\hat{\tau} \leq \tau$.

The parametric longest path algorithm computes $\tau_i^*$ in a decreasing order of the values. First $\tau_i^*$ is computed for node $i$ with the largest $\tau_i^*$. In the next step the second largest $\tau_i^*$ is computed and so forth. The algorithm produces a sequence of trees $T_j$ and

$\tau^j$, where $j$ is the iteration index. In every iteration $j$ the tree $T_j$ is the longest path tree for any weights $c_e = a_e - \tau b_e$, $e \in A$ with $\tau^{j+1} \le \tau \le \tau^j$. Suppose we have $\tau^j$ and the corresponding tree $T_j$. By Proposition 2, $T_j$ is an optimal longest path tree for all $\tau$ such that $\tau \ge \max\{\frac{d_a^{T_j}(u) - d_a^{T_j}(v) + a_e}{d_b^{T_j}(u) - d_b^{T_j}(v) + b_e}\}$ and therefore

$$\tau^{j+1} = \max_{\substack{e=(u,v)\in E \\ d^{T_j}{}_b(u) - d^{T_j}{}_b(v) + b_e > 0}} \{\frac{d_a^{T_j}(u) - d_a^{T_j}(v) + a_e}{d_b^{T_j}(u) - d_b^{T_j}(v) + b_e}\}. \quad (10)$$

If this maximum is attained for $e=(u,v)\in A$, then clearly $T_{j+1} = T_j \cup \{(u,v)\} \setminus \{(pred_v, v)\}$.

The algorithm is given in Figure 6. At every iteration and for each $i\in N$ we maintain numbers $d_a(i) = \sum_{e\in P^T(i)} a_e$ and $d_b(i) = \sum_{e\in P^T(i)} b_e$, where $T$ is the current tree. $PQ$ is a priority queue (see e.g. Cormen, Leiserson and Rivest (1989)), where each element of $PQ$ encodes an arc $e=(u,v)$ and the corresponding key of the element is $\frac{d_a(u) - d_a(v) + a_e}{d_b(u) - d_b(v) + b_e}$. An element of $PQ$ is denoted by <*key*,*e*>, where $e$ is an arc and *key* is the corresponding key. The initial tree $T$ is obtained as in step 3 of the primal-dual algorithm. Next in steps 5-10 we initialize $PQ$. In step 13 we find the arc with the largest key. In steps 14-19 we use Proposition 2 and the ratio path optimality condition to obtain some $\tau_i^*$. If $\tilde{\tau} = \sum_{e\in P^T(i)} a_e \Big/ \sum_{e\in P^T(i)} b_e$ for a node $i\in \overline{S}$ is between $\bar{\tau}$ and $\tau$, then $T$ is the longest path tree for weights $c_e = a_e - \tilde{\tau} b_e$, $e\in A$. But then the distance in $T$ of node $i$ is 0 and therefore we can use the ratio path optimality condition. In steps 21-42 we update the keys of $PQ$. First in steps 22-25 we update $d_a$ and $d_b$ based on (2). Let $e=(i,j) \in A$ be an arbitrary arc. If $i \notin T(v)$ and $j \notin T(v)$, then by (1) the key does not change. If $i \in T(v)$ and $j \in T(v)$, then by (2) the key does not change. Steps 27-42 update the keys if either $i \in T(v), j \notin T(v)$ or $i \notin T(v)$, $j \in T(v)$.

Since $D$ is acyclic, in every iteration $T$ is a tree and by above discussion, it is clear that the algorithm is correct and it terminates in a finite number of steps.

---

**Input:** An acyclic network $D = (N, A)$
**Output**: $\tau_i^*$ for all $i \in N\backslash\{s\}$

```
1.   For all $i\in N$ compute $\overline{d}(i)$, $\underline{d}(i)$ corresponding to the longest, shortest s-i path with
     respect to weights $a_e$, $b_e$, respectively.
2.   $\tau = \max_{i\in N} \frac{\overline{d}(i)}{\underline{d}(i)}$
3.   Let $c_e=a_e - \tau b_e$ for all $e\in A$ and use Dijkstra to obtain the longest path tree T with
     respect to weights c.
4.   For all $i\in N$ compute $d_a(i)$, $d_b(i)$.
5.   for all $e=(i,j)\in A$ do
```

```
6.      if $d_b(i) - d_b(j) + b_e > 0$ then
7.          $key = \frac{d_a(i) - d_a(j) + a_e}{d_b(i) - d_b(j) + b_e}$
8.          Insert <$key$, e> in $PQ$.
9.      end if
10. end for
11. $\bar{S} = N \setminus \{s\}$
12. while $\bar{S} \neq \emptyset$ do
13.     Find the pair with the maximum key in $PQ$. Let <$\bar{\tau}$,$e$> be such a pair, where
        $e$=($u,v$).
14.     for all $i \in \bar{S}$ do
15.         if $\bar{\tau} \le \frac{d_a(i)}{d_b(i)} \le \tau$ then
16.             $\tau_i^* = \frac{d_a(i)}{d_b(i)}$
17.             $\bar{S} = \bar{S} \setminus \{i\}$
18.         end if
19.     end for
20.     $\tau = \bar{\tau}$
21.     Let $T(v)$ be the subtree of $T$ rooted at $v$.
22.     for all $i \in T(v)$ do
23.         $d_a(i) = d_a(i) + \Delta_1^T(u,v)$
24.         $d_b(i) = d_b(i) + \Delta_2^T(u,v)$
25.     end for
26.     $pred_v = u$
27.     for all $i \in T(v)$ do
28.         for all $e=(i,j) \in A$ and $j \notin T(v)$ do
29.             Remove <$\tau$ ,$e$> from $PQ$.
30.             if $d_b(i) - d_b(j) + b_e > 0$ then
31.                 $key = \frac{d_a(i) - d_a(j) + a_e}{d_b(i) - d_b(j) + b_e}$
32.                 Insert <$key$,$e$> in $PQ$.
33.             end if
34.         end for
35.         for all $e=(j, i) \in A$ and $j \notin T(v)$ do
36.             Remove <$\tau$ ,$e$> from $PQ$.
37.             if $d_b(j) - d_b(i) + b_e > 0$ then
38.                 $key = \frac{d_a(j) - d_a(i) + a_e}{d_b(j) - d_b(i) + b_e}$
39.                 Insert <$key$,$e$> in $PQ$.
40.             end if
```

```
41.             end for
42.         end for
43.     end while
```

**Figure 6.** The parametric longest path algorithm

**Running Time**

Here we assume that *PQ* is either a binary heap or a red-black tree. Since by assumption the cost *a* and the time *b* are integral, all the denominators in the keys of *PQ* are greater or equal to 1. Therefore in step 24 at every iteration $\Delta_2^T(u,v) \geq 1$. Clearly $d_b(i) \leq n\overline{B}$ for all $i \in N$. We conclude that the key of an arc is updated at most $2n\overline{B}$ times. The factor 2 is present since an arc has two endpoints and the value $n\overline{B}$ is included since $d_b(i)$ can change at most $n\overline{B}$ times for any $i \in N$. Every operation on *PQ* requires at most $O(\log(m))=O(\log(n))$ time. Therefore the running time of the algorithm is $O(mn\overline{B}\log(n))$. The parametric longest path algorithm is a pseudo-polynomial algorithm.

## 3. Computational Experiments

The computational experiments were conducted on an SGI Origin200 workstation with a RISC 12000 processor running at the clock speed of 270 MHz. The operating system is IRIX, version 6.5, and the workstation is equipped with 512 MB of main memory. The algorithms are implemented in C++ by using the MIPSpro, version 7.3, development environment.

The algorithms were tested on several random acyclic networks and on instances resulting from large-scale linear programming, Makri and Klabjan (2002). We define the density *d* of a network as $d = m \Big/ (\frac{n(n-1)}{2})$, where the denominator is the maximum number of arcs on *n* nodes. For a given density we generate random networks as follows. Let $k=d(n-1)/2$ represent the average out degree of every node. For every node we first generate a random number *s* from the uniform discrete distribution from $[k-t,k+t]$, where *t* is selected in such a way that the variance is a given number (50 in our case). The neighbors of the node correspond to a random subset of cardinality *s* from $\{i+1,\ldots,n\}$. The cost of each arc is a random number in the range $[-\widetilde{A},\widetilde{A}]$ and the time is a random number from $[\underline{B},\overline{B}]$. We selected a symmetric interval for the cost since the running time analysis suggest that the running time should depend only on the magnitude of the cost. The computational experiments were carried out with various interval ranges.

We choose $s$=1, i.e. the source is the node with label 1. Note that based on our generation method some nodes *i* might not have an *s*-*i* path. For all such nodes *i* we add arcs to random nodes $k,k<i$ that are connected to *s*. Experiments have shown that on average less than 10% of the nodes are not connected to *s*.

In Section 2.1 we presented two variants of the primal-dual algorithm. Computation experiments have revealed that the version with steps 9a-16a is more efficient. This has already been observed in Dasdan, Irani and Gupta (1998). We came to the conclusion that changing the tree too much in these steps is not beneficial. Therefore we have designed a third variant of the primal-dual algorithm, which is the same as the algorithm in Figure 1 except that step 13 is not carried out. It means that in steps 9-16 we do not update the distances but only the tree. It turns out that this version of the primal-dual algorithm is the most efficient one and therefore in what follows the primal-dual algorithm corresponds to this variant.

Our network generation method yields networks that are already topologically sorted. Therefore arc scanning in the primal-dual algorithm is based on this order and step 6 of the bisection algorithm requires a single scan of the arcs. To study the impact of different node orders, we ran the primal-dual algorithm with random topological orders. The running times increased. Note that the parametric longest path algorithm is indifferent to the node order.

In the parametric longest path algorithm we use the binary heap as the priority queue $PQ$. For efficiency, in the implementation an element of the heap stores also references to the arcs (in addition to storing arcs and keys).

In the remaining part of this section we first compare the running times of the three algorithms and then we give a more detailed analysis of the primal-dual and the parametric longest path algorithms. For each experiment with a fixed density, number of nodes, and $\widetilde{A}, \underline{B}, \overline{B}$ we have performed 20 runs and we took the average of the observed values. All the computational times are in seconds.

**3.1 Algorithm Comparisons**

We show later in Section 3.2 and Section 3.3 that the primal-dual and the parametric longest path algorithms are strongly polynomial on random networks and therefore its running time depends only on $n$ and $d$. The results shown in this section were obtained by using $\underline{B} = 1, \overline{B} = 500, \widetilde{A} = 1000$.

Young, Tarjan, Orlin (1991) propose an efficient implementation of the parametric shortest path algorithm. Their implementation stores in the priority queue only one element for each node. This element corresponds to the incident arc with the largest key. This modification can easily be embedded in our algorithm and it leads to an implementation that requires $O(n)$ space instead of $O(m)$. We denote the resulting algorithm as PLP and the algorithm presented in Figure 6 as PLP_A. In Figure 7 we compare the running time in seconds of the two implementations. We see that PLP is substantially faster than the original algorithm and therefore from this point on we consider only PLP.

Next we show the running time comparison of the primal-dual algorithm (denoted by PD in the figures that follow) and the parametric longest path algorithm. The running times are plotted in Figure 8. We see that the parametric shortest path algorithm outperforms the primal-dual algorithm for all the densities. The number of the iterations of the primal-dual algorithm is lower than the number of the iterations of the parametric shortest path algorithm (this is shown later), however, the time per iteration is much lower in the latter algorithm. This is due to the fact that in the primal-dual algorithm we have to scan all the edges at every iteration but on the other side only few heap updates in the parametric shortest path algorithm are required per iteration.

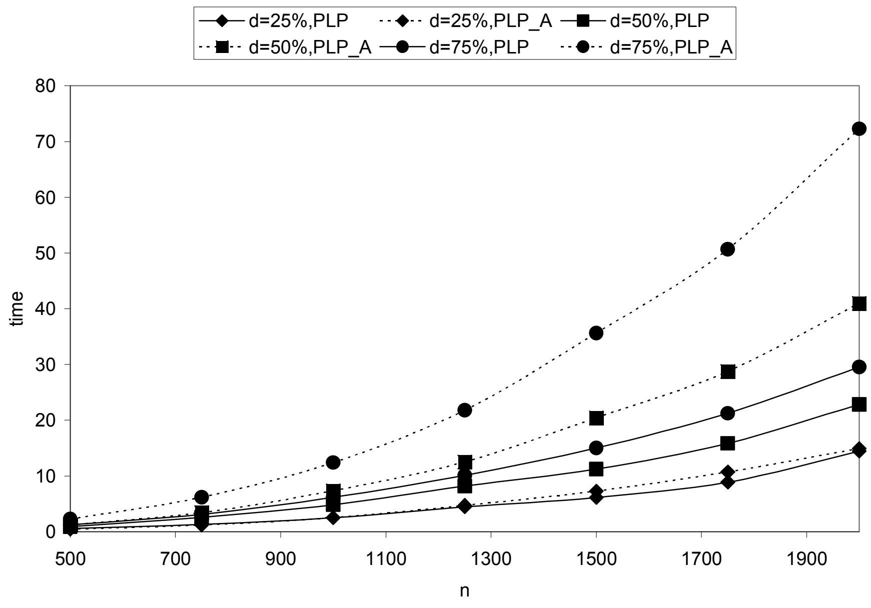

**Figure 7.** The comparison of the Young-Tarjan-Orlin implemention of the parametric longest path algorithm (PLP) and the original presentation (PLP_A)

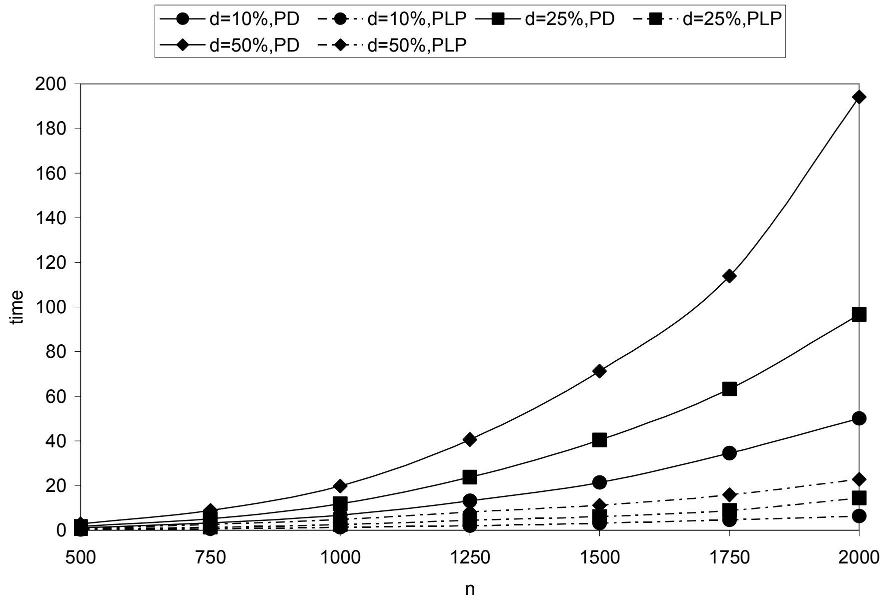

**Figure 8.** Time comparison of the primal-dual algorithm (PD) and the parametric longest path algorithm (PLP)

Figure 9 shows the computational time comparison of the bisection algorithm and the primal-dual algorithm. Regardless of the density, the bisection algorithm is inferior and therefore it is the least efficient algorithm among the three. On sparse networks its running time is slower by a factor of 4 and for denser networks the factor is 8. The drawback of the bisection algorithm is a large number of iterations. Note that the time per iteration is comparable to the time per iteration of the primal-dual algorithm since it essentially requires scanning all of the arcs. Based on the empirical evidence of the bisection algorithm, we believe the algorithm that reduces the problem to the minimum cost to time cycle problem (see introduction to Section 2) is not efficient.

Table 1 shows the computational times on networks that are typical in large-scale linear programs solved in Makri and Klabjan (2002). The cost and the time in these instances are as well taken from these linear programs. We can see that these networks tend to have a high number of nodes but are extremely sparse. The parametric longest path algorithm is the winner. Due to the large execution times, we did not test the bisection algorithm for the largest 2 instances. During the execution of the linear programming solver designed in Makri and Klabjan (2002), MCT has to be performed several times and therefore low execution times are needed. The running times of the parametric longest path algorithm are within the acceptable limits. Since these networks are sparse, we have performed additional experiments on random networks with *d*=2.5%, which are given in Figure 10. In all of the cases the parametric longest path algorithm outperforms the primal-dual algorithm.

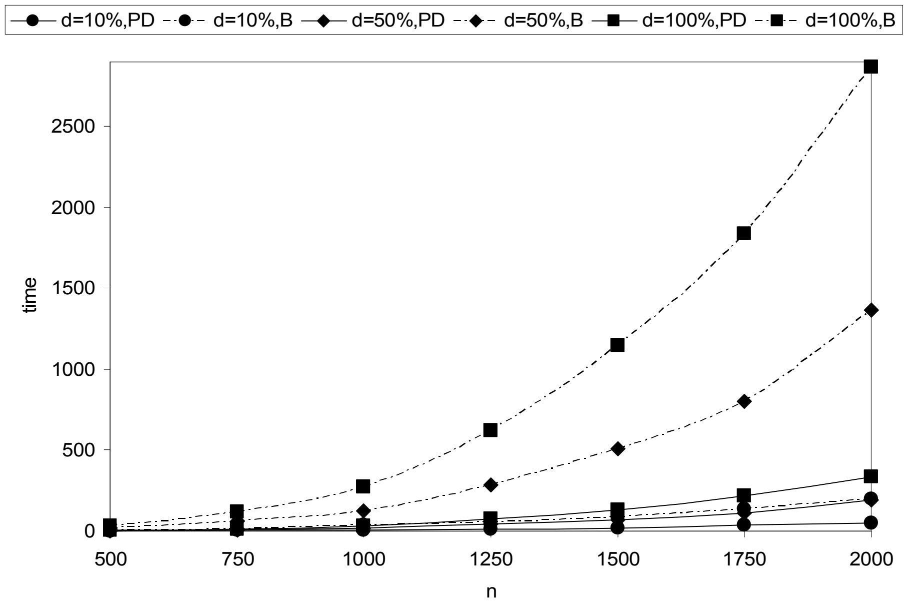

**Figure 9.** Time comparison of the primal-dual algorithm and the bisection algorithm (B)

| (*n*, *m*, *d*) | primal-dual algorithm | Bisection algorithm | parametric longest Path algorithm |
|---|---|---|---|
| **(2358, 75487, 2.7%)** | 51 | 173 | 22 |
| **(2397, 74757, 2.6%)** | 38 | 134 | 19 |
| **(3058, 109422, 2.3%)** | 97 | 327 | 55 |
| **(6496, 302949, 1.4%)** | 428 | 1475 | 20 |
| **(8120, 281280, 0.9%)** | 504 | - | 21 |
| **(11844, 661164, 0.9%)** | 1308 | - | 58 |

**Table 1.** Running times on instances from large-scale linear programming

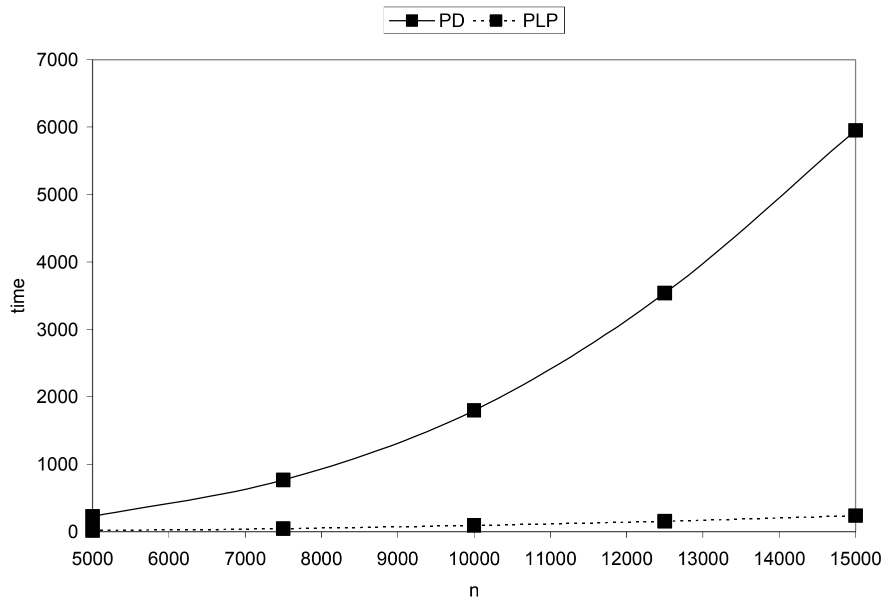

**Figure 10.** The comparison of PD and PLP on networks with *d*=2.5%

Since the bisection algorithm has poor performance, we did not carry out additional experiments with this algorithm. In the remaining two sections we give more insight into the parametric longest path algorithm and the primal-dual algorithm.

## 3.2 Computational Analysis of the Primal-Dual Algorithm

We first show that the running time of the primal-dual algorithm does not depend on the range of the input data *a* and *b*. In Figure 11 we first plot the running time versus the largest absolute value of the cost. In these experiments we have selected various values of $\underline{B}$ and $\overline{B}$ and the density of either 25% or 50%. The running time analysis from Section 2.1 suggests that the running time depends also on the ratio $\overline{B}/\underline{B}$ and therefore we show in Figure 12 the dependency of the running time and this ratio. We show the dependency for various numbers of nodes and the density of either 25% or 50%. As we can observe from these two figures, the running times do not depend on these two quantities. For each observation in Figure 12 we have made several runs with equal ratio but different $\underline{B}$ and $\overline{B}$. In addition to these experiments, we have measured the running times by varying independently $\underline{B}$ and $\overline{B}$. These data, which are not shown here, comply with the above findings that the running times do not depend on these two values. All of these experiments suggest that on random networks the primal-dual algorithm is strongly polynomial.

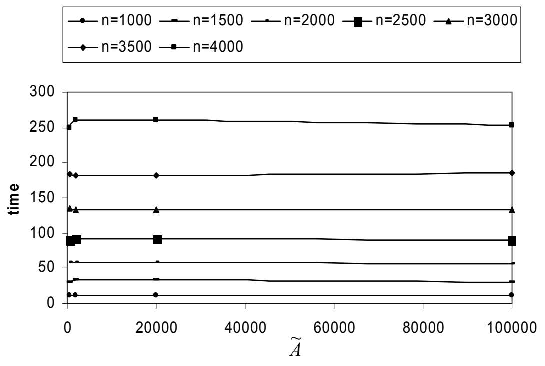

**Figure 11.** Dependency of the cost range and running time for PD

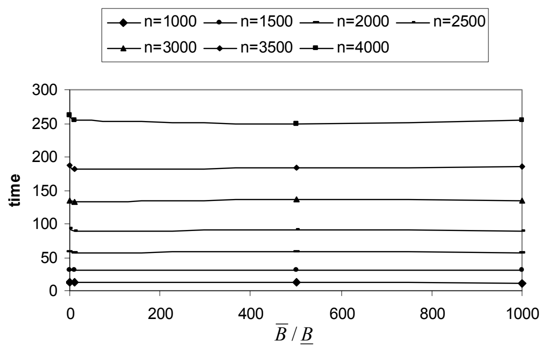

**Figure 12.** Dependency of the time range and the running time for PD

Figure 13 shows the number of iterations of the primal-dual algorithm with respect to the density of the network. The number of iterations depends only on the number of nodes and not on the number of arcs. By using linear regression we have fitted the number of iterations as the function $cn^k m^l$. We obtained that the number of iterations is $1.607n^{0.96}m^{-0.13}$. The ratio of the actual number of iterations over these estimated values is shown in Figure 14. As we can observe, the error is always less than 3%. The computational experiments show that the number of iterations on random networks is $O(n)$ and therefore it seems that the complexity of the algorithm is $O(nm)$ on random networks.

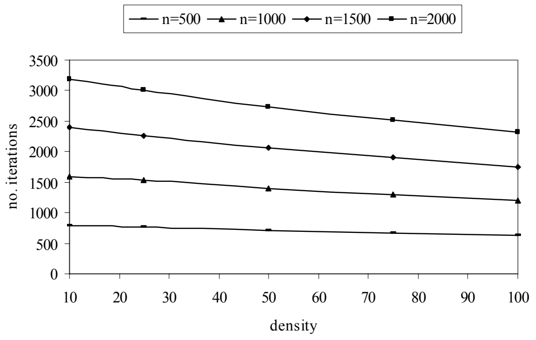

**Figure 13**. The number of iterations of the primal-dual algorithm

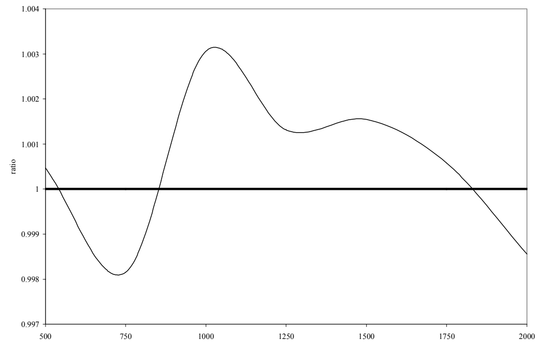

**Figure 14.** The ratio number of iterations over estimated number of iterations in PD

We conclude this section by showing the growth of the actual running time with respect to the number of nodes and the density of the network. Figure 15 shows the execution times for a wide range of densities. We can observe a linear growth in terms of the number of arcs and a slightly super linear growth in the number of nodes, which is a consequence of a linear growth of the number of iterations and time per iteration with respect to the number of nodes.

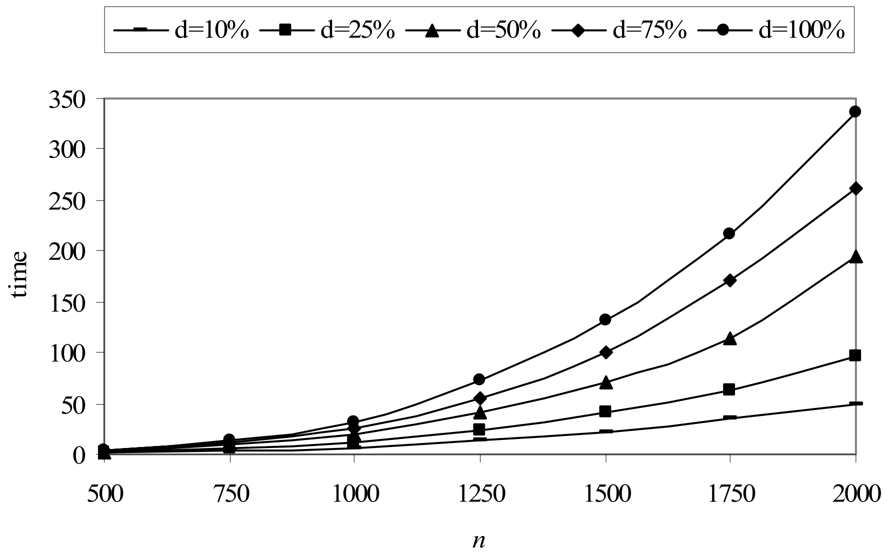

**Figure 15.** Running times of the primal-dual algorithm

### 3.3 Computational Analysis of the Parametric Longest Path Algorithm

The running time analysis of the parametric longest path algorithm suggests that the execution time does not depend on $\widetilde{A}$. In Figure16 we show that the running time of the parametric longest path algorithm does not depend on $\widetilde{A}$. In this figure, due to the high running times, we have plotted the logarithms of the running times.

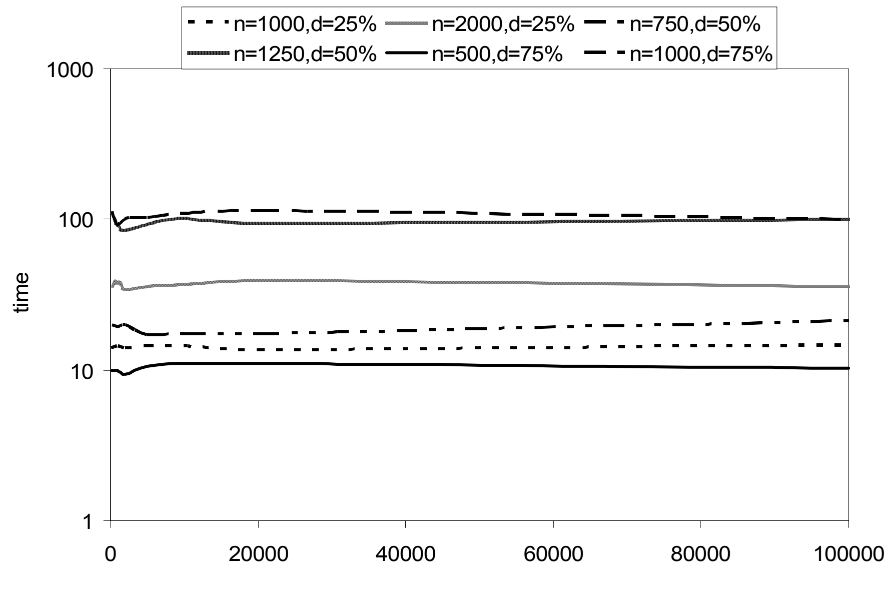

**Figure 16.** Running time of the parametric longest path algorithm with respect to the range of the cost

The dependency of the running times with respect to $\overline{B}$ is shown in Figure 17 and Figure 18 for $d$=2.5% and $d$=50%, respectively. We chose to study a sparse family since the networks in linear programs are sparse. Similar to the primal-dual algorithm, we see that the times do not depend on $\overline{B}$ .

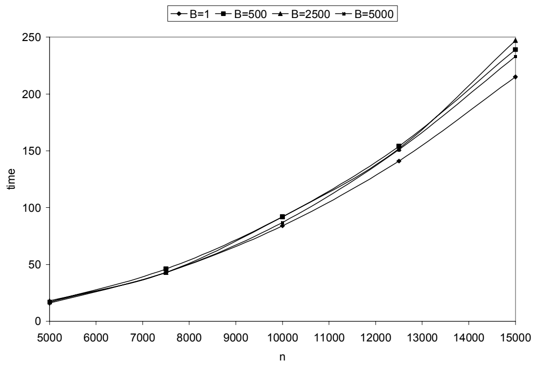

**Figure 17.** Execution time of PLP with *d*=2.5%

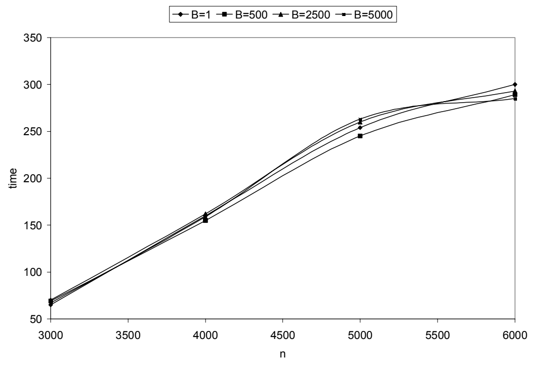

**Figure 18.** Execution time of PLP with *d*=50%

Figure 19 and Figure 20 show the number of iterations of the algorithm and the number of heap updates for *d*=2.5% and *d*=50%, respectively. The number of iterations is scaled by 10. On average the number of heap updates per iteration is 14 for *d*=2.5% and 23 for *d*=50%. Note that this is substantially less than the overall number of arcs and this is the primal reason of success of the parametric longest path algorithm over the primal-dual algorithm. By applying linear regression similar to the one we carried out in the primal-dual algorithm analysis we conclude that the number of heap updates equals to $1.9n^{1.2}\ m^{0.1}$. If we compare this relation multiplied by *log(n)*, which is the worst time of a heap operation, with $1.607n^{0.96}m^{0.87}$ obtained by the primal-dual algorithm and multiplied by *m* (the time per iteration of the primal-dual algorithm), we see that the exponent of *m* of the parametric longest path algorithm is lower than that of the primal-dual algorithm. This argument gives us an additional explanation about the dominance of the parametric longest path algorithm.

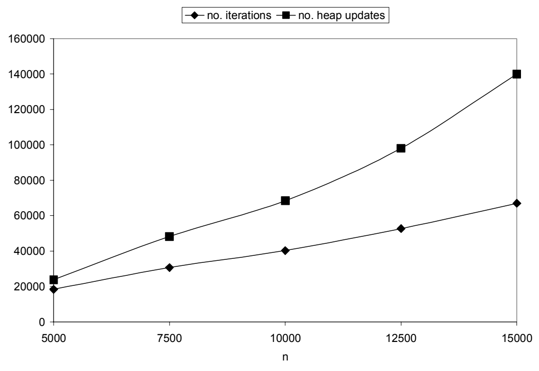

**Figure 19.** The number of iterations and heap updates for *d*=2.5%

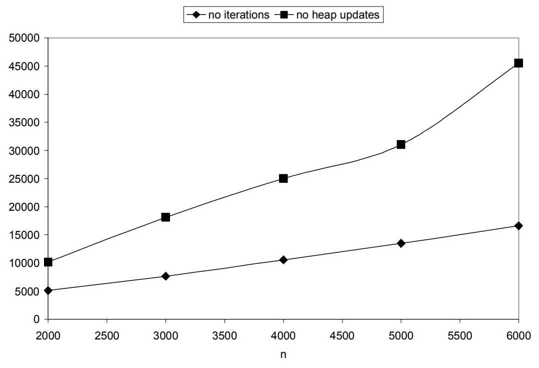

**Figure 20.** The number of iterations and heap updates for *d*=50%

In Figure 21 and Figure 22 we show the ratio of the time spent in updating the heap over the overall execution time. For the sparse family less than 50% of the time is spent on these updates but this ratio grows as *n* increases. This is understandable since the size of the heap depends on *n*. On the other hand, for the denser network heap updates dominate and form on average 80% of the time.

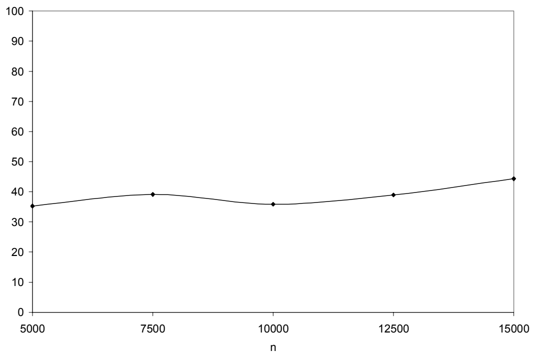

**Figure 21.** The percentage of heap updates for *d*=2.5%

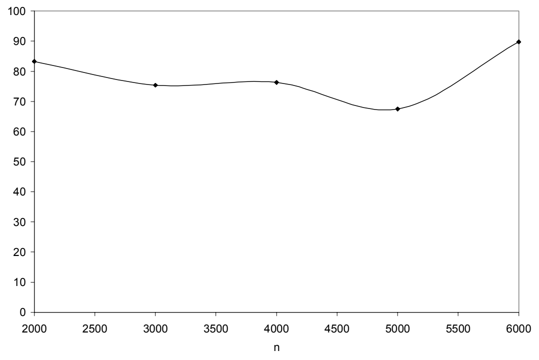

**Figure 22.** The percentage of heap updates for *d*=50%

## 4. Conclusions

We presented three algorithms for solving MCT: a primal-dual algorithm, a bisection algorithm, and a parametric longest path algorithm. For each one of them we show the correctness and we carry out a running time analysis. The first and the third algorithms are pseudo polynomial whereas the bisection algorithm is polynomial.

We performed extensive computational experiments. The bisection algorithm is by far the least efficient one and the parametric longest path algorithm outperforms the primal-dual algorithm. On random networks the computational experiments showed that the algorithms are polynomial.

**Acknowledgment**

We are grateful to two anonymous referees whose comments have substantially improved the paper, particularly the computational part.

## References

AHUJA, R., MAGNANTI, T. and ORLIN, J. *Network flows*. Prentice Hall (1993).

BARNHART, C., JOHNSON, E., NEMHAUSER, G., SAVELSBERGH, M. and VANCE, P. Branch-and-Price: Column generation for solving huge integer programs. *Operations Research* **46**, 316-329 (1998).

COCHET-TERRASSON, J., COHEN, G., GAUBERT, S., McGETTRICK, M., and QUADRAT, J. Numerical computation of spectral elements in max-plus algebra. In *Proceedings of IFAC Conference on System Structure and Control* (1998).

CORMEN, T. , LEISERSON, C., and RIVEST, R. *Introduction to Algorithms.* McGraw-Hill (1989).

DASDAN. A. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *Synopsys Technical Report 2001-10-22-01,* Synopsys Inc. (2001).

DASDAN, A., IRANI, S. and GUPTA, R. Efficient algorithms for optimum mean and optimum cost to time ratio problems. *Proceedings of the 1999 36th Annual Design Automation Conference*, 37-42 (1999).

DASDAN, A., IRANI, S. and GUPTA, R. An experimental study of minimum mean cycle algorithms. *Technical Report #98-32*, University of California, Irvine (1998).

DIAL, R. Algorithm 360: Shortest path forest with topological ordering. *Communications of ACM* **12**, 632-633 (1969).

KARP, R. and ORLIN, J. Parametric shortest path algorithms with an application to cyclic staffing. *Discrete Applied Mathematics* **3,** 37-45 (1981).

LAWLER, E. *Combinatorial optimization: Networks and matroids*. Holt, Reinhart, and Winston (1976).

MAKRI, A. and KLABJAN, D. A new pricing scheme for airline crew scheduling. To appear in *INFORMS Journal on Computing* (2002). Available from http://www.staff.uiuc.edu/~klabjan/professional.html.

YOUNG, N., TARJAN, R and ORLIN, J. Faster parametric shortest path and minimum-balance algorithms. *Networks* **21**, 205-221 (1991).