Computational Optimization and Applications, 4, 99–125 (1995)


# Polynomial Auction Algorithms for Shortest Paths*

DIMITRI P. BERTSEKAS
*Laboratory for Information and Decision Systems, M.I.T, Cambridge, Mass. 02139*

STEFANO PALLOTTINO AND MARIA GRAZIA SCUTELLÀ
*Department of Computer Science, University of Pisa, Corso Italia 40, 56125 Pisa, Italy*

*Received August 30, 1993. Revised June 29, 1994*

**Abstract.** In this paper we consider strongly polynomial variations of the auction algorithm for the single origin/many destinations shortest path problem. These variations are based on the idea of graph reduction, that is, deleting unnecessary arcs of the graph by using certain bounds naturally obtained in the course of the algorithm. We study the structure of the reduced graph and we exploit this structure to obtain algorithms with $O(n \min\{m, n \log n\})$ and $O(n^2)$ running time. Our computational experiments show that these algorithms outperform their closest competitors on randomly generated dense all destinations problems, and on a broad variety of few destination problems.

**Keywords:** shortest path, network optimization, auction

## 1. Introduction

In this paper we focus on the auction algorithm for shortest path problems proposed by Bertsekas [4], [5]. This algorithm is closely related to auction algorithms for other network flow problems, and in particular to the naive auction algorithm for the assignment problem introduced in [1], and further discussed in [2] and in the tutorial paper [3]; see [5] for a detailed analysis of these relations. The auction algorithm can be viewed as a special type of dual coordinate ascent method, and is fundamentally different from the classical label setting and label correcting methods, which can be viewed as primal cost improvement methods. For the single origin and single destination case, the algorithm is very simple. It maintains a single path starting at the origin. At each iteration, the path is either *extended* by adding a new node, or *contracted* by subtracting its terminal node. When the destination becomes the terminal node of the path, the algorithm terminates.

The auction algorithm has a pseudopolynomial complexity, but in practice it outperforms its closest competitors by a broad margin for important types of problems with one origin and few destinations. However, for the many (or all) destinations case, the algorithm is apparently outperformed by state-of-the-art label setting and label correcting methods. It is possible to convert the auction algorithm to a weakly polynomial one by using the device of arc length scaling, but then unfortunately its practical performance tends to become worse.

Strongly polynomial versions of the auction algorithm were obtained by Pallottino and Scutellà [11] by adding to the extension and contraction operations a *reduction* operation. Here, each time a node $i$ becomes the terminal node of the path for the first time, all its

*Research supported by NSF under Grant No. DDM-8903385, by the ARO under Grant DAAL03-86-K-0171, by a CNR-GNIM grant, and by a Fullbright grant.

incoming arcs except the one of the path are deleted; since the path is shortest for $i$, these arcs can be deleted. The auction algorithm thus obtained has an $O(m^2)$ running time, where $m$ is the number of arcs. By using the idea of presorting the outgoing arcs of each node in order of non-decreasing length, the running time was reduced further to $O(mn)$, where $n$ is the number of nodes.

In this paper we strengthen the graph reduction idea by using upper bounds to the node shortest distances in order to delete arcs more effectively. We study the structure of the reduced graph thus obtained, and we exploit this structure to obtain algorithms with improved time complexity. In particular, we develop an algorithm with $O(n \min\{m, n \log n\})$ running time, and another one, somewhat more complicated, which runs in $O(n^2)$ time. These theoretical improvements have resulted in auction algorithms with substantially improved practical performance for single origin/all destinations problems, as well as for difficult single origin/few destinations problems, for which the original method exhibits pseudopolynomial behavior.

The paper is organized as follows: In Section 2, we describe the original auction algorithm. In Section 3, we describe the graph reduction process, and we observe that it creates an interesting graph structure, named the *extended tree*. This structure is useful in explaining the mechanism of graph reduction, in simplifying the proof of the associated complexity bounds, and in suggesting ways to improve the algorithm's theoretical and practical performance. In particular, in Section 4, with a small modification of the auction algorithm with graph reduction, we improve its running time from $O(n \min\{m, n \log n\})$ to $O(n^2)$. Finally, in Section 5, we discuss implementations of the various algorithms and present computational results. These results show that for dense all destinations problems, auction algorithms with graph reduction outperform by a modest margin their closest competitors. For broad classes of few destination problems, auction algorithms with graph reduction outperform their closest competitors by a large margin.

## 2. The Auction Algorithm for Shortest Paths

We first describe the original auction algorithm of [4] for the single origin and single destination case. We are given a directed graph $(\mathcal{N}, \mathcal{A})$. We assume that each node has at least one outgoing arc, and that there is at most one arc between two nodes in each direction, so that we can unambiguously refer to an arc $(i, j)$. In the following, by a *path* we mean a sequence of nodes $(i_1, i_2, \ldots, i_k)$ such that $(i_q, i_{q+1})$ is an arc for all $q = 1, \ldots, k-1$. If in addition the nodes $i_1, i_2, \ldots, i_k$ are distinct, the sequence $(i_1, i_2, \ldots, i_k)$ is called a *simple path*. A sequence of nodes $(i_1, i_2, \ldots, i_k)$ such that $i_1 = i_k$ and $(i_q, i_{q+1})$ is an arc for all $q = 1, \ldots, k-1$ is called a *cycle*. Each arc $(i, j)$ has a length $a_{ij}$ associated with it. The length of a path or of a cycle is defined to be the sum of its arc lengths. In this section, we assume that *all cycles have positive length*, although the initialization of the algorithm is greatly simplified if, in addition, all arc lengths are nonnegative. Of course, it is necessary to assume that there are no *negative* length cycles, because otherwise there may not exist a shortest path between the origin and the destination.

Let node 1 be the origin node and let $t$ be the destination node. The algorithm maintains at all times a simple path $P = (1, i_1, i_2, \ldots, i_k)$. The node $i_k$ is called the *terminal* node of $P$. The degenerate path $P = (1)$ may also be obtained in the course of the algorithm. If $i_{k+1}$ is a node that does not belong to a path $P = (1, i_1, i_2, \ldots, i_k)$ and $(i_k, i_{k+1})$ is an arc, *extending $P$ by $i_{k+1}$* means replacing $P$ by the path $(1, i_1, i_2, \ldots, i_k, i_{k+1})$. If $P$ does not consist of just the origin node 1, *contracting* $P$ means replacing $P$ with the path $(1, i_1, i_2, \ldots, i_{k-1})$.

The algorithm also maintains a variable $p_i$ for each node $i$ (called *price* of $i$) such that

$$p_i \le a_{ij} + p_j, \qquad \forall\, (i, j) \in \mathcal{A}, \tag{1a}$$

$$p_i = a_{ij} + p_j, \qquad \text{for all pairs of successive nodes } i \text{ and } j \text{ of } P. \tag{1b}$$

We denote by $p$ the vector of prices $p_i$. A pair $(P, p)$ consisting of a path $P$ and a price vector $p$, that satisfies the above conditions, is said to satisfy *complementary slackness* (or CS for short). Note that, by our assumption on the cycle lengths, if $(P, p)$ satisfies the CS conditions, then $P$ is a simple path. A basic fact is that if a pair $(P, p)$ satisfies the CS conditions, then the portion of $P$ between node 1 and any node $i \in P$ is a shortest path from 1 to $i$, while $p_1 - p_i$ is the corresponding shortest distance. To see this, observe that by Eq. (1b), $p_1 - p_i$ is the length of the portion of $P$ between 1 and $i$, and by Eq. (1a) every path connecting 1 and $i$ must have length at least equal to $p_1 - p_i$.

We will assume that an initial pair $(P, p)$ satisfying CS is available. This is not a restrictive assumption when all arc lengths are nonnegative, since then one can use the default pair

$$P = (1), \qquad p_i = 0, \ \forall\, i.$$

When some arcs have negative lengths, an initial choice of a pair $(P, p)$ satisfying CS may not be obvious or available but there are algorithms for obtaining such a pair (see [4], [5] for a discussion of this point).

We now describe the algorithm. Initially, $(P, p)$ is any pair satisfying CS. The algorithm proceeds in iterations, transforming a pair $(P, p)$ satisfying CS into another pair satisfying CS. At each iteration, the path $P$ is either extended by a new node or else is contracted by deleting its terminal node. In the latter case, the price of the terminal node is increased strictly. A degenerate case occurs when the path consists of just the origin node 1; in this case the path is either extended, or else is left unchanged with the price $p_1$ being strictly increased. The iteration is as follows:

*Typical Iteration of the Auction Algorithm*

Let $i$ be the terminal node of $P$. If

$$p_i < \min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}, \tag{2}$$

go to Step 1; else go to Step 2.

**Step 1: (Contraction)** Set

$$p_i := \min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}, \tag{3}$$

and if $i \ne 1$, contract $P$. Go to the next iteration.

**Step 2: (Extension)** Extend $P$ by node $j_i$, where

$$j_i = \arg\min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}. \tag{4}$$

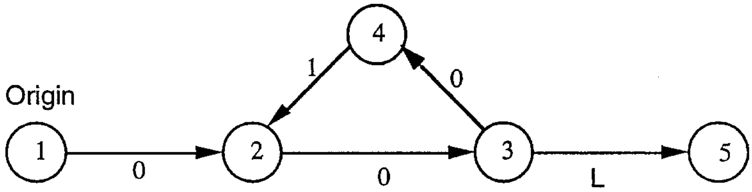


*Figure 1.* Illustration of pseudopolynomial behavior of the auction algorithm. Here the origin is node 1 and the destination is node 5. The arc lengths are shown next to the arcs. $L$ is a large integer. Beginning with $p_i = 0$, for all $i$, the price $p_i$ is increased $L$ times, by 1 unit, for $i = 1, 2, 3, 4$. In fact, only when $p_3$ reaches $L$, an extension to the destination 5 is possible.

If $j_i$ is the destination $t$, stop: $P$ is the desired shortest path. Otherwise, go to the next iteration.

Following the extension Step 2, $P$ is a simple path from 1 to $j_i$. Indeed, if this were not so, then adding $j_i$ to $P$ would create a cycle, and for every arc $(i, j)$ of this cycle we would have $p_i = a_{ij} + p_j$. Thus, the cycle would have zero length, which is not possible by our assumptions. If on the other hand it were known that there is no arc $(i, j)$ such that $j$ belongs to $P$, the algorithm would also be valid in the case where there are zero length cycles; this observation will be useful in the next section.

It is shown in [4] that if there exists at least one path from 1 to $t$, then $t$ will become the terminal node of the path $P$ in a finite number of iterations, at which time $P$ is a shortest path from 1 to $t$. The algorithm's worst-case running time was shown to be pseudopolynomial in [4], assuming that the arc lengths $a_{ij}$ are integer; in fact, it is not difficult to see that the number of iterations is bounded by $n^2L$, where $L = \max_{(i,j)\in\mathcal{A}} |a_{ij}|$ is the arc length range. An example of the worst-case behavior of the algorithm is shown in Fig. 1. For randomly generated problems, the average running time does not seem to depend on $L$. However, one may anticipate practical situations where pseudopolynomial behavior occurs because there are many cycles with small length in the graph, such as the one illustrated in Fig. 1.

The single destination algorithm can be used to find a shortest path tree to all destinations; in fact, one can switch to a new destination once a shortest path to a given destination is found, while leaving the pair $(P, p)$ unchanged. Equivalently, in order to find a shortest path tree, one can just continue to iterate until every destination becomes the terminal node of $P$.

The preceding algorithm may be termed "forward" in that the path $P$ gradually extends in the forward direction from the origin towards the destination. It is possible to combine this forward algorithm with a "reverse" version that maintains a path $R$ ending at the destination, which is gradually extended backward towards the origin. When the paths $P$ and $R$ meet, a shortest path is found. Also in the case where there are several destinations, it is possible to maintain a separate path for each destination. Such combined forward/reverse versions are much faster than the forward version described earlier when the number of destinations is relatively small. Apparently this is due to the fact that the number of nodes that become terminal is much smaller in forward/reverse versions. For simplicity, in the following sections we restrict attention primarily to forward algorithms, but these algorithms admit forward/reverse versions as well.

## 3. The Auction Algorithm with Graph Reduction

We now describe a strongly polynomial auction algorithm for the all destinations shortest path problem. This algorithm differs from the one of Section 2 in that each contraction or extension is preceded by a graph reduction operation, that deletes unnecessary nodes or arcs of the graph.

Each iteration starts with a subgraph $\mathcal{G}$ of the original graph $(\mathcal{N}, \mathcal{A})$ and a pair $(P, p)$ satisfying CS, and generates a subgraph $\overline{\mathcal{G}}$ of $\mathcal{G}$ and another pair $(\overline{P}, \overline{p})$ satisfying CS. Note that $P$ is a path of $\mathcal{G}$, while $\overline{P}$ is a path of $\overline{\mathcal{G}}$. Furthermore, $p$ consists of a price for each node of $\mathcal{G}$, while $\overline{p}$ consists of a price for each node of $\overline{\mathcal{G}}$.

In the following, we call $\mathcal{G}$ the *reduced graph*, to distinguish it from $(\mathcal{N}, \mathcal{A})$, which we call the *original graph*. The set of arcs of the reduced graph is denoted by $\mathcal{A}_r$. In the absence of a contrary statement, when we refer to arcs and nodes in the course of the algorithm, we imply that they belong to the (current) reduced graph. A node of the reduced graph that has already become the terminal node of $P$ at least once is referred to as a *tree node*. A node $j$ that is not a tree node but is connected with an arc $(i, j) \in \mathcal{A}_r$ to a tree node $i$ will be referred to as a *border node*. An arc $(i, j)$ of the reduced graph will be called a *tree arc* if both $i$ and $j$ are tree nodes, and it will be called a *border arc* if $i$ is a tree node and $j$ is a border node. It will be shown later that throughout the algorithm, each tree node other than the origin has a unique incoming arc in the reduced graph, and that this arc is a tree arc. Furthermore, the origin has no incoming arcs in the reduced graph. Thus the tree nodes and tree arcs form a tree, justifying our terminology.

The typical iteration of the auction algorithm with graph reduction is essentially the same as the one of the auction algorithm of the preceding section, except that the basic contraction or extension step is preceded by the graph reduction step. There are two ways in which the graph reduction step can delete arcs or nodes:

(a) When the current terminal node has no outgoing arcs, in which case the node is deleted, and the iteration is terminated.

(b) When the current terminal node $i$ has some outgoing arcs, but this is the first iteration at which $i$ is a tree node. In this case, all the incoming arcs of $i$, except the tree arc that belongs to $P$, are deleted, and some new border arcs may be created, which can cause some additional arcs to be deleted.

Central to the graph reduction process is a variable $u_j$ for each node $j$, which in the course of the algorithm is an upper bound to the shortest distance from 1 to $j$; $u_j$ is monotonically nonincreasing, and it is equal to the shortest distance once $j$ becomes the terminal node of $P$. As will be shown shortly [Prop. 1(g)], $u_j$ behaves exactly as the temporary label of node $i$ that is generated by Dijkstra's algorithm; see e.g. [10], [7], [5]. It will be shown that at all times, $u_j$ is equal to the length of the shortest path from 1 to $j$ in the subgraph consisting of the tree and border nodes. In fact, an arc $(k, j)$ is deleted if the path from 1 to $j$ going through the terminal node of $P$ is shorter than the current "best" path going through $k$.

We now describe the algorithm. Initially,

$$u_i = \begin{cases} 0 & \text{if } i = 1, \\ \infty & \text{if } i \neq 1, \end{cases} \tag{5}$$

$P = (1)$, $p$ is any vector such that $p_i \leq a_{ij} + p_j$ for all $(i, j) \in \mathcal{A}$, and the reduced graph $\mathcal{G}$ is equal to the original graph.

*Typical Iteration of the Auction Algorithm with Graph Reduction*

Let $i$ be the terminal node of $P$. If $i$ has no outgoing arcs and $i = 1$ stop (the problem is infeasible); else go to Step 1.

**Step 1: (Graph Reduction)** If $i$ has no outgoing arcs, go to Step 1(a); else go to Step 2 or Step 1(b) depending on whether $i$ was the terminal node of $P$ at some earlier iteration or not.

**Step 1(a): (Deletion of Terminal Node)** Contract $P$, delete $i$ and the arc of $P$ that was incoming to $i$, and go to the next iteration.

**Step 1(b): (First Scan of Terminal Node)** Delete all incoming arcs of $i$, except (if $i \neq 1$) for the arc of $P$ that is incoming to $i$. Also, for each outgoing arc $(i, j) \in \mathcal{A}_r$, if

$$u_i + a_{ij} \geq u_j, \tag{6}$$

delete $(i, j)$; else delete the arc $(k, j) \in \mathcal{A}_r$ for which $k$ is a tree node other than $i$, and set

$$u_j := u_i + a_{ij}. \tag{7}$$

If $i$ has no outgoing arcs left, contract $P$, delete $i$ and all its incoming arcs, and go to the next iteration; otherwise go to Step 2.

**Step 2: (Decide on Contraction or Extension)** If

$$p_i < \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\}, \tag{8}$$

go to Step 3; else go to Step 4.

**Step 3: (Contraction)** Set

$$p_i := \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\},$$

and if $i \neq 1$, contract $P$. Go to the next iteration.

**Step 4: (Extension)** Extend $P$ by node $j_i$ where

$$j_i = \arg \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\}.$$

If the number of the tree nodes is equal to $n$, then stop: the set of the tree arcs defines a shortest path tree. Otherwise, go to the next iteration.

Note that the algorithm now includes an infeasibility test. We will show shortly that, in a finite number of iterations, the algorithm either produces a shortest path tree rooted at the origin, or else it verifies that some nodes are not connected to 1. Figure 2 illustrates how as a result of graph reduction, the number of iterations to solve the problem of Fig. 1 is dramatically reduced.

There are two structures underlying the algorithm, which will also prove particularly important in the complexity analysis:

(a) The set $T$ of tree nodes and tree arcs, which will be called *the shortest path tree*; this set will be shown to be a tree in the following Prop. 1(e).

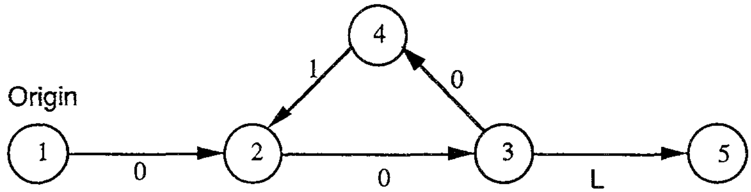


*Figure 2.* Illustration of the algorithm with zero initial prices for the example problem of Fig. 1. In the auction algorithm with graph reduction, the first iteration is an extension along the arc (1, 2) at which arc (4, 2) is deleted. The second and third iterations are extensions along arcs (2, 3) and (3, 4), respectively. At the fourth iteration, node 4 and arc (3, 4) are deleted. At the fifth, sixth, and seventh iterations, a contraction occurs at nodes 3, 2, and 1, respectively. At the eighth, nineth, and tenth iterations, an extension occurs at nodes 1, 2, and 3, respectively, and the algorithm terminates since then all nodes will have become tree nodes. If arc (3, 4) were not present, the nodes 1, 2, 3, and 5 would become tree nodes in that order, and then nodes 5, 3, 2, and 1 would be deleted in that order, indicating that the problem is infeasible (there is no path from 1 to 4).

(b) The set $E$ of the tree arcs and the border arcs, which will be called *the extended tree*; this set will also be shown to be a tree in the following Prop. 1(f).

Figure 3 provides another illustration of the algorithm, the shortest path tree $T$, and the extended tree $E$. The original graph is shown in Fig. 3(a). First, the price $p_1$ is raised to 1, node 1 becomes a tree node, and nodes 2 and 6 become border nodes; an extension to node 2 is then performed, and nodes 3 and 4 become border nodes. Then an extension to node 3 is performed, the arcs (5, 3) and (6, 3) are deleted, and node 7 becomes a border node. After successive contractions of $P$ all the way to the origin, and successive extensions all the way to node 5, arcs (8, 5) and (3, 7) are deleted, as shown in Fig. 3(b). As far as the deletion of the border arc (3, 7) is concerned, it is due to the improvement of $u_7$ because a shorter path, through node 5, has been found. The extended tree $E$ is shown in Fig. 3(c). After further successive contractions of $P$ all the way to the origin, the subsequent extension is to node 6, and then to node 7; node 8 enters $E$ as a border node, while arcs (5, 7) and (8, 7) are deleted. Note that the current reduced graph coincides with the final shortest path tree, even if 8 is still a border node. In the following iterations, successive contractions of $P$ all the way to the origin are performed, followed by successive extensions all the way to node 5, and by another sequence of contractions to node 1. During these operations, nodes 4 and 5, together with the arcs (2, 4) and (4, 5), are deleted. Another sequence of extensions to node 3, followed by a sequence of contractions to the origin, causes the deletion of nodes 2, 3, and arcs (1, 2) and (2, 3). After that, $P$ is extended all the way to the last node 8, which thus enters the shortest path tree $T$ as the algorithm terminates.

Before proceeding with the analysis, we note that the graph reduction allows us to solve a more general problem. First, we do not need to assume that each node has at least one outgoing arc, since such a node will automatically be deleted in the graph reduction step. Second, we can relax the positivity assumption on the cycle lengths to nonnegativity. The purpose of disallowing zero length cycles was to ensure that the extension step did not lead to a node $j$ that is already in $P$, thereby closing a cycle (see the remark following the description of the auction algorithm in the preceding section). Since, however, all incoming arcs of a tree node except its unique predecessor arc on $P$ are deleted in the graph reduction step, it is impossible to close such a cycle even when there are zero length cycles.

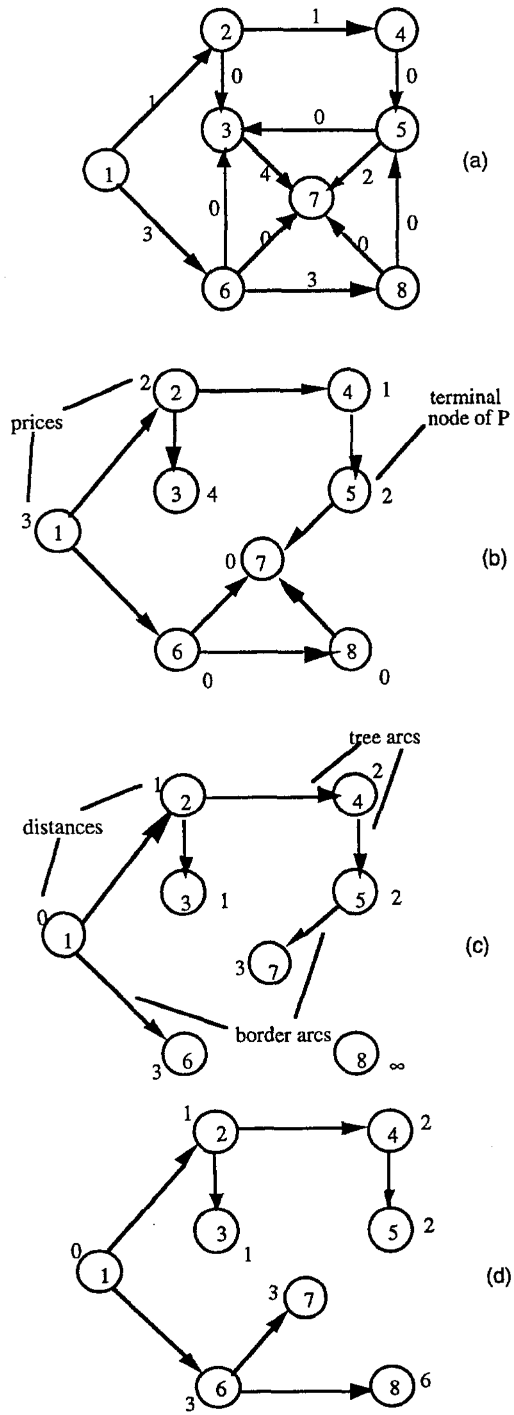


*Figure 3.* Illustration of the algorithm for an example problem.

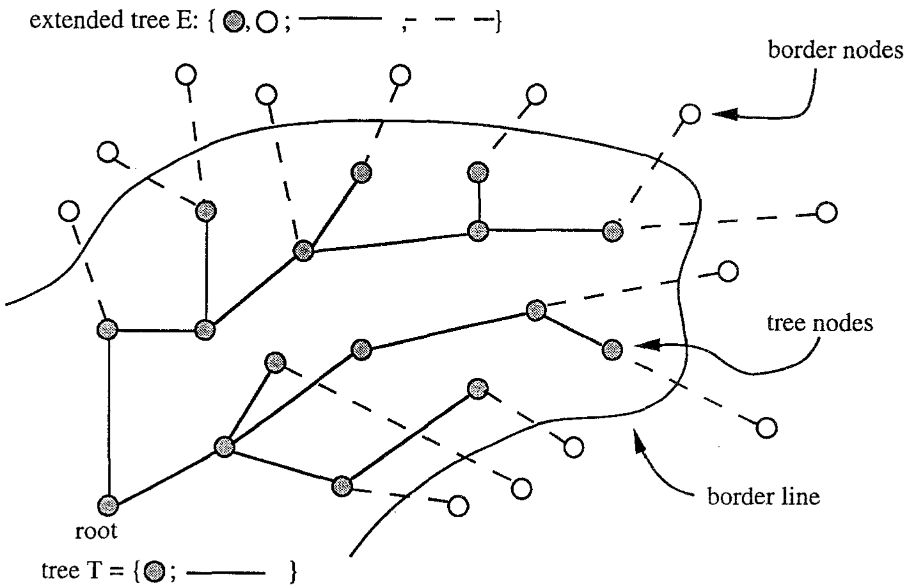


*Figure 4.* Illustration of the structure of the extended tree $E$.

The following proposition establishes some basic facts and identifies some important graph structures underlying the algorithm.

PROPOSITION 1. *The following properties hold at the end of the graph reduction step of each iteration of the auction algorithm with graph reduction*:

*(a)* $(P, p)$ *satisfies CS.*
*(b) The shortest distance of a node from the origin in the reduced graph is the same as its shortest distance in the original graph.*
*(c) Each tree node except the origin has a unique incoming arc in the reduced graph, and this arc is a tree arc. The origin has no incoming arc in the reduced graph.*
*(d) Each border node has a unique incoming border arc in the reduced graph.*
*(e) The shortest path tree* $T$ *is a tree rooted at the origin. For each tree node* $i$*, the unique path of* $T$ *from* 1 *to* $i$ *is a shortest path.*
*(f) The extended tree* $E$ *is a tree rooted at the origin, in which all border nodes are leaf nodes* (*cf. Fig.* 4).
*(g) For all tree nodes and border nodes* $i$, $u_i$ *is equal to the length of the unique path of the tree* $E$ *from* 1 *to* $i$*. Furthermore,* $u_i$ *is equal to the shortest distance from* 1 *to* $i$ *using paths of the original graph that consist of tree nodes, except perhaps for* $i$.
*(h) If* $i$ *is a tree node,* $u_i$ *is equal to the shortest distance from* 1 *to* $i$.

PROOF. We use induction. In the starting iteration, the terminal node of $P$ is the origin 1. The graph reduction step deletes all incoming arcs of 1, and sets $u_j = a_{1j}$ for all outgoing arcs $(1, j)$, which become border arcs, while the corresponding nodes $j$ become border nodes. There is only one tree node (the origin) and there are no tree arcs. It is straightforward to verify (using also our assumption that there is at most one arc between two nodes in each direction), that properties (a)–(h) hold at the end of this first graph reduction step.

Suppose now that we are at the end of the graph reduction step of an iteration. We will show that properties (a)–(h) hold, assuming that they hold at the end of all preceding graph reduction steps. There are four possibilities:

(1) The preceding graph reduction step was followed by a contraction that transformed the path $P' = (1, \ldots, i, j)$ to the path $P = (1, \ldots, i)$ and changed the price $p_j$ to $\min_{(j,k)\in\mathcal{A}_r}\{a_{jk} + p_k\}$. This left the trees $T$ and $E$ unchanged. It is straightforward to verify that properties (a)–(h) were maintained by the contraction. The subsequent graph reduction step changed nothing, since $i$ must have been the terminal node at an earlier iteration, while also $i$ could not be deleted since it has at least one outgoing arc [the tree arc $(i, j)$]. Therefore, properties (a)–(h) hold at the end of the graph reduction step in this case.

(2) The preceding graph reduction step ended with deletion of the terminal node and its incoming arc. In this case, no contraction or extension was performed. Since the new terminal node $i$ must have been the terminal node at some earlier iteration, the subsequent graph reduction step either deleted $i$ (if $i$ had no outgoing arcs), or else changed nothing. In either case it can be seen that properties (a)–(h) were preserved.

(3) The preceding graph reduction step was followed by an extension that transformed the path $P' = (1, \ldots, j)$ to the path $P = (1, \ldots, j, i)$, where the node $i$ was already a tree node (it had been the terminal node of $P$ at an earlier iteration). This left the price vector $p$ and the trees $T$ and $E$ unchanged, so the properties (a)–(h) were maintained by the extension. The subsequent graph reduction step could at most delete $i$ and the tree arc $(j, i)$ in the case where $i$ has no outgoing arc, but this still would have maintained the properties (a)–(h).

(4) The preceding graph reduction step was followed by an extension that transformed the path $P = (1, \ldots, j)$ to the path $P' = (1, \ldots, j, i)$, where the node $i$ had never before been the terminal node. This left the price vector $p$ unchanged, but created a new tree node (the node $i$, which was previously a border node), a new tree arc [the arc $(j, i)$, which was previously the unique border arc incoming to $i$, based on the induction hypothesis], and possibly several new border nodes and border arcs. By property (g) and the induction hypothesis, $u_i$ must have been equal to the length of the path $P'$, which is a shortest path since the pair $(P', p)$ satisfies CS. Thus $u_i$ was equal to the shortest distance of $i$ as required by property (h). It can be also seen that properties (a), (b), and (e) must be satisfied at the end of this extension step, but the remaining properties (c), (d), (f), and (g) may not hold for two reasons: 1) the new tree node $i$ may have some incoming arcs $(k, i)$, where $k$ is not a tree node, and 2) some border nodes may have two incoming border arcs (one that existed prior to the extension and a second one that was created when $i$ became a tree node). It can be seen, however, that the following graph reduction step will delete all incoming arcs of $i$ [except $(j, i)$ in the case where $i$ itself is not deleted because it has no outgoing arcs], so that property (c) will be restored. Furthermore, the graph reduction step will also delete exactly one of the two incoming border arcs for every border node that has two incoming border arcs, thereby restoring properties (d) and (f). In addition, for all outgoing arcs $(i, k)$, the value of $u_k$ will be updated in a way that makes it equal to the shortest distance from 1 to $k$ using paths consisting of tree nodes except for $k$, as required by property

(g). As a result, all the properties (a)–(h) must hold at the end of the graph reduction step. The induction proof is now complete.

□

We now show the validity of the algorithm.

PROPOSITION 2. *Assume that all cycles have nonnegative length. The auction algorithm with graph reduction terminates either with a shortest path tree, or with the deletion of the origin. The latter case occurs if and only if the problem is infeasible.*

PROOF. Suppose that the algorithm does not terminate. Then at least one tree node, say $i$, will become the terminal node of $P$ infinitely many times and its price $p_i$ will also increase infinitely many times. Following each increase of $p_i$, we have $p_i = a_{ij} + p_j$ for some outgoing arc $(i, j)$, so there is at least one tree node $j$ such that $(i, j)$ is an arc and $p_j$ will also increase infinitely many times. Using repeatedly this argument, we conclude that there must be a cycle of tree nodes whose prices will increase infinitely many times. But this is a contradiction since, by Prop. 1, the tree nodes and the corresponding tree arcs belong to the shortest path tree $T$, which cannot contain a cycle. Thus the algorithm must terminate.

By definition, the algorithm can only terminate if either each node of the original graph has become the terminal node of $P$ at least once [in which case, by parts (a) and (b) of Prop. 1, a shortest path tree will be found], or if the tree $T$ collapses into the origin. Just before the latter case occurs, the reduced graph will consist of two disconnected subgraphs, the first being just the origin, and the second containing all nodes which have never become tree nodes. Since by Prop. 1(b) the shortest distance from 1 to each such node is the same in the reduced and in the original graph, it follows that there are no paths from 1 to those nodes in the original graph, and so the algorithm correctly indicates that the problem is infeasible in this case. □

*Estimate of the Running Time of the Algorithm*

It is useful for the purposes of our complexity analysis to divide the iterations into *first scan iterations*, *contraction cycles*, and *extension cycles*. A first scan iteration is an iteration in which the terminal node of $P$ is terminal for the first time (it has just become a tree node). An extension cycle is a sequence of successive iterations involving an extension, such that (a) the iteration immediately preceding the cycle either is a first scan iteration or involves a contraction, and (b) the iteration immediately following the cycle also either is a first scan iteration or involves a contraction. Similarly, a contraction cycle is a sequence of successive iterations involving a contraction, such that the iteration immediately preceding the cycle is either a first scan iteration or else involves an extension, while the iteration immediately following the cycle involves an extension. (Note that a first scan iteration can only follow an iteration involving an extension.) As an example, for the problem of Fig. 1, the algorithm first performs first scan iterations at nodes 1, 2, 3, and 4 in that order, then performs a contraction cycle involving nodes 3, 2, and 1 in that order, and finally performs an extension cycle involving nodes 1, 2, and 3 in that order, and then terminates when node 5 becomes the terminal node of $P$.

To estimate the running time of the algorithm, we first note that the first scan iterations take $O(m)$ total time, since they involve at most one calculation of sums of the form $u_i + a_{ij}$ and $a_{ij} + p_j$ per arc $(i, j)$, and a total of $O(m)$ arc deletions. Next we note that an extension cycle takes $O(n)$ time because it involves examination of a subset of the arcs of the extended tree $E$ existing at the start of the cycle, and there are at most $(n - 1)$ such arcs since $E$ is a tree by Prop. 1(f). Similarly, a contraction cycle takes $O(n)$ time because it involves the examination of a subset of the arcs of the extended tree existing at the start of the cycle. Clearly, there are less than $n$ first scan iterations. As far as the extension and the contraction cycles are concerned, we will show that their number is $O(\min\{m, n \log n\})$, thereby leading to an $O(n \min\{m, n \log n\})$ running time (assuming that $m \geq n$).

The key property is that an iteration that extends $P$ by an arc $(i, j)$ is followed by a (possibly empty) sequence of successive extensions that leads to one of two possible types of nodes, as illustrated in Fig. 3:

(a) A border node $k$, in which case $k$ becomes a tree node, and a first scan iteration follows.
(b) A tree node $k$ (maybe $k = j$), in which case a contraction follows together with either the deletion of node $k$ (if it has no outgoing arcs), or an increase of $p_k$.

The extension cycle that led to $k$ is said to be *successful* in case (a) and *failed* in case (b). The number of successful extension cycles is less than $n$, since each such cycle is followed by a first scan iteration. The number of failed extension cycles is shown in the Appendix to be $O(\min\{m, n \log n\})$. Thus the total number of extension cycles is $O(\min\{m, n \log n\})$. Finally, since a contraction cycle is preceded by either a first scan iteration or a failed extension cycle, the number of contraction cycles is also less than $O(\min\{m, n \log n\})$. We have thus proved the following proposition.

PROPOSITION 3. *The auction algorithm with graph reduction has an* $O(n \min\{m, n \log n\})$ *running time.*

An interesting question is whether the worst-case bound $O(n \min\{m, n \log n\})$ is tight and also whether it is representative of the practical performance of the algorithm. According to the preceding analysis, the running time can be divided into three parts:

(a) The $O(m)$ time needed for first scan operations; this is clearly a tight bound and cannot be improved.
(b) The time needed for the $O(n)$ successful extension cycles and their following contraction cycles. Typically, the number of such cycles is of the order of $n$, but the number of operations per cycle depends on the average number of arcs in the path $P$ during the cycle. For sparse graphs, the time for these contraction and extension cycles typically seems to be of order $n^2$, consistent with our earlier worst-case analysis, and probably dominates the $O(m)$ time needed for first scan iterations. In contrast, for very dense or complete graphs ($m = n^2$), the time for first scan iterations seems to dominate.
(c) The time needed for the failed extension cycles and the following contraction cycles. We derived an $O(n \min\{m, n \log n\})$ estimate for this time, making it the complexity bottleneck. In practice, however, this time seems to be negligible relative to the times for (a) and (b) above. The reason is that the number of failed extension cycles is much smaller than the estimated number (and indeed much smaller than $n$ according to our experimentation).

We have been unable to construct an example showing the tightness of the $O(\min\{m, n \log n\})$ bound for the number of failed extension cycles. It is thus an open question whether this bound can be improved. It is possible, however, to modify the algorithm and guarantee that failed extension cycles never occur, at the expense of $O(n^2)$ extra work. This leads to algorithms with an $O(n^2)$ running time, which are the subject of the next section.

## 4. Auction Algorithms with Improved Running Time

We now consider the possibility of modifying the algorithm to eliminate the failed extension cycles described in the preceding section. To this end, we introduce some definitions.

Following each contraction or extension at the terminal node $i$, there are one or more arcs $(i, j)$ such that $p_i = a_{ij} + p_j$. We arbitrarily select one of these arcs and call it the *candidate arc of* $i$, until the next iteration when a contraction or an extension at $i$ occurs, and a possibly different arc becomes the candidate arc of $i$. We adopt the convention that an extension occurs along the candidate arc, that is, if $(i, j)$ is the candidate arc of $i$ and an extension takes place at $i$, then node $j$ becomes the terminal node of $P$, while $(i, j)$ continues to be the candidate arc of $i$. This guarantees that every arc of the current path $P$ is a candidate arc. Note that, according to our definition, a tree node always has a candidate arc. It is possible, however, that an arc can be deleted while it is the candidate arc of a node, and indeed it can be shown that this is what causes failed extension cycles (see the Appendix for further discussion of this point).

We now introduce a graph, called *multipath* and denoted $M$, which will play a central role in the algorithms of this section. The nodes of $M$ are the tree nodes and the border nodes of the current reduced graph, that is, the tree and the border nodes that have not yet been deleted; the arcs of $M$ are the candidate arcs $(i, j)$ that still belong to the reduced graph and still satisfy $p_i = a_{ij} + p_j$.

It can be seen that the multipath $M$ is a subgraph of both the reduced graph and the extended tree, and that the path $P$ belongs to $M$. Note that $M$ can change with each iteration. In particular, a new arc $(i, j)$ may enter $M$ by becoming a candidate arc through a contraction or extension at $i$; also a current candidate arc $(i, j)$ that belongs to $M$ may stop satisfying $p_i = a_{ij} + p_j$ because of a contraction at $j$, or may be deleted because of a first scan iteration at a node $k$ with $u_k + a_{kj} < u_j$, in which case it will get out of $M$ [cf. Eqs. (6) and (7)].

Given a node $i$ of the multipath $M$, it can be seen that there is a unique path of $M$, denoted $P_M(i)$, with the property that $P_M(i)$ starts at $i$ and ends at a node $j$ that has no outgoing arc in the multipath [if node $i$ itself has no outgoing arc, as for example when $i$ is a border node, then $P_M(i) = (i)$]. The reason for existence and uniqueness of this path is that each node has at most one outgoing arc in $M$, since at most one of the outgoing arcs of a node can be a candidate arc at any one time, and furthermore $M$ has no cycles, since it is a subgraph of the extended tree. The last node of $P_M(i)$ is denoted by *last*$(i)$ and is either a border node or a tree node. If *last*$(i)$ is a border node for every node $i$ of $M$ that does not belong to $P$, we say that $M$ is *complete*; otherwise we say that $M$ is *incomplete*. Figure 5 illustrates these definitions.

We now make two observations that are important for our purposes:

(a) When the path $P$ extends to a tree node $i$ and *last*$(i)$ is a border node, a sequence of

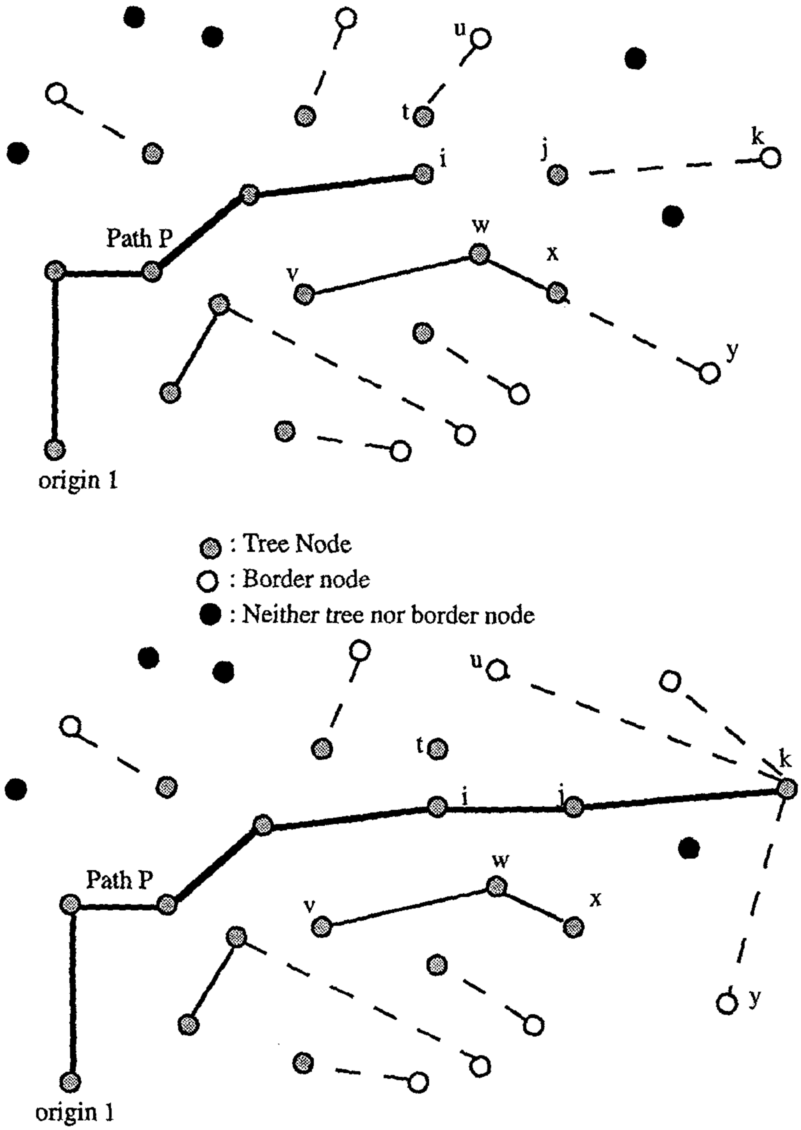


*Figure 5.* Illustration of a multipath. Its arcs are the candidate arcs $(i, j)$ that still belong to the reduced graph and still satisfy $p_i = a_{ij} + p_j$. Tree arcs are shown with solid lines and border arcs are shown with broken lines. The tree nodes are gray shaded. In (a) a complete multipath is shown; the last node $last(i)$ of each path $P_M(i)$ is a border node. In (b), the multipath is shown after two successive extensions (to node $j$ and then to node $k$) and the first scan operation at node $k$, in which arcs $(t, u)$ and $(x, y)$ are deleted. The multipath becomes incomplete because the last node of each of the paths $P_M(t)$, $P_M(v)$, $P_M(w)$, and $P_M(x)$ is not a border node.

extensions follows along the path $P_M(i)$ until the border node *last*($i$) is reached by $P$, and a first scan iteration occurs at that node.

(b) If $M$ is complete at the start of some iteration that is not a first scan iteration, then $M$ will still be complete at the end of the iteration. To prove this we assume that the terminal node $i$ at the given iteration is a tree node, and we show that at the end of the iteration, the last nodes of the paths $P_M(j)$ of all nodes $j \notin P$ are border nodes. Indeed, the paths $P_M(j)$ of all nodes $j \notin P$, except possibly for $i$, have this property because they are not affected by the iteration. There remains to consider the case of a contraction when the terminal node $i$ exits $P$. Then $i$ acquires a candidate arc $(i, j)$ during the iteration, and the path $P_M(i)$ at the end of the iteration consists of arc $(i, j)$ followed by the path $P_M(j)$, which must end at a border node since $M$ is complete at the start of the iteration. This completes the proof that the completeness of $M$ is preserved by an iteration other than a first scan iteration.

Observation (a) above shows that if we could guarantee that $M$ is complete at all times, there would be no failed extension cycles; each first scan iteration would be followed by a (possibly empty) contraction cycle, which would in turn be followed by a successful extension cycle. Unfortunately, in the auction algorithm of the preceding section, the multipath need not be complete, and observation (b) shows that this is due to first scan iterations when some candidate arcs may be deleted. This motivates the idea of occasionally restructuring the multipath to either maintain its completeness or to otherwise ensure that whenever an extension to a tree node $i$ occurs, *last*($i$) is a border node. If this could be done in $O(n)$ time per first scan iteration, the total time required for multipath restructuring would be $O(n^2)$, and the running time of the algorithm would be $O(n^2)$ by the analysis of the preceding section. We will provide two different ways for doing this.

### *Connection Sequences*

Suppose that at the end of an iteration of the auction algorithm with graph reduction we have a multipath $M$ that is not complete. Then there must exist at least one tree node $i \notin P$ such that *last*($i$) $= i$, that is, the current candidate arc $(i, j)$ has been either deleted or else the price $p_j$ was increased at least once since the last time $(i, j)$ became the candidate arc of $i$. For a tree node $i \notin P$ with *last*($i$) $= i$, we define an operation, called a *connection operation at* $i$, which is defined as follows: it deletes $i$ and its unique incoming arc if $i$ has no outgoing arcs, and otherwise sets

$$p_i := \min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\},$$

and selects arbitrarily one of the arcs $(i, j)$ attaining the minimum above and declares it as the new candidate arc of $i$.

Note that when a connection operation at $i$ is performed and $i$ is not deleted, exactly one arc will be added to the multipath (this is the new candidate arc of $i$), but it is possible that another candidate arc will get out of the multipath; this will happen if for some tree node $v$, the arc $(v, i)$ was a candidate arc, and $p_i$ was increased through the connection operation, in which case $(v, i)$ will not continue to qualify for membership in the multipath, so that *last*($v$) $= v$.

Our intent is to restore the completeness of portions of the multipath through a sequence of connection operations at tree nodes $i \notin P$ with *last*$(i) = i$. Such a sequence is called a *connection sequence* and is specified by the corresponding sequence of tree nodes at which the connection operations are performed. We would like a connection sequence to contain at most one connection operation per node, so we impose the requirement that once a connection operation is performed at a node $i$, the condition *last*$(i) = i$ does not arise at the end of a subsequent connection operation in the same sequence. It will be shown below (Prop. 4) that this is guaranteed if the sequence of nodes

$$\{i_1, i_2, \ldots, i_k\},$$

corresponding to the sequence of connection operations, has the property that for $q = 2, \ldots, k$, the unique path from 1 to $i_q$ on the shortest path tree does not pass through node $i_s$ for all $s < q$ (this will be true in particular if, for $q = 2, \ldots, k$, node $i_q$ became a tree node before node $i_{q-1}$). A connection sequence with this property is called *properly ordered.* In particular, we have the following proposition.

PROPOSITION 4. *Consider a properly ordered connection sequence corresponding to a set of nodes*

$$I = \{i_1, i_2, \ldots, i_k\}.$$

*Then for $q = 1, \ldots, k$, at the end of the connection operation at $i_q$ we have last$(i_s) \neq i_s$ for all $s \leq q$ such that $i_s$ was not deleted during the connection operation at $i_s$.*

PROOF. We use induction. Suppose that for some $q < k$, at the end of the connection operation at $i_q$, we have *last*$(i_s) \neq i_s$ for all $s \leq q$ such that $i_s$ was not deleted. Consider the multipath $M$ at the end of the connection operation at $i_{q+1}$. Then we have *last*$(i_{q+1}) \neq i_{q+1}$, since $i_{q+1}$ just acquired a new candidate arc that qualifies for membership in $M$. The only node of $M$ whose candidate arc can exit $M$ during the connection operation at $i_{q+1}$ is a node $i$ such that $(i, i_{q+1})$ was the candidate arc of $i$ at the start of this connection operation. Such a node cannot be one of the nodes $i_s$ with $s \leq q$, because the path of the shortest path tree from 1 to $i_{q+1}$ cannot pass through such a node (by the definition of a properly ordered connection sequence). Therefore, by the induction hypothesis, we have *last*$(i_s) \neq i_s$ for all $s \leq q + 1$, completing the induction proof. □

The main consequence of Prop. 4 is that a properly ordered connection sequence can contain at most one connection operation per node.

### *Multipath Restructuring*

In what follows we assume that some additional data structures are maintained by the algorithm. These are:

(a) A data structure that maintains the portion of the shortest path tree $T$ that consists of the undeleted nodes. This data structure must have the property that it allows the enumeration of the nodes of any subtree $T_j$ of $T$ that is rooted at some node $j$ in a way that is consistent with the topological order of the subtree, and in time proportional to the number of nodes of the subtree. In particular if the subtree $T_j$ consists of $k$ nodes, the data structure should allow in time $O(k)$ the ordering of the nodes as

$$\{i_1, i_2, \ldots, i_k\},$$

where for $q = 2, \ldots, k$, the unique path from $j$ to $i_q$ on $T_j$ does not pass through node $i_s$ for all $s < q$. Furthermore, the total time for maintaining the data structure during the entire duration of the algorithm should be $O(n)$; there are several tree data structures that can be used for this purpose.

(b) Two arrays that maintain the candidate arc and the node *last*($i$) for each node $i$ of the multipath. Maintaining these arrays can be done with $O(1)$ computation per contraction, extension, or connection operation, so the additional overhead will be lumped into the time for contractions, extensions, and connection operations. The reason for maintaining *last*($i$) is to be able to check quickly whether *last*($i$) is a border node, and whether *last*($i$) $= i$. There is, however, an additional computational benefit: if an extension occurs at a tree node $i$ such that *last*($i$) $\neq i$, we know that it will be followed by several extensions along the corresponding candidate arcs, culminating with *last*($i$) becoming the terminal node of $i$. Thus we can jump directly to *last*($i$), and save the computation for the extensions.

Suppose now that we are given the subtree $T_j$ of the shortest path tree $T$ that is rooted at a node $j$ and we construct a properly ordered connection sequence as follows: we start by performing a connection operation at a node $i_1 \notin P$ of $T_j$ with *last*($i_1$) $= i_1$, which is such that the path of $T_j$ from $j$ to any node $i$ of $T_j$ with *last*($i$) $= i$ does not pass through $i_1$; then we perform a connection operation at a node $i_2 \notin P$ of $T_j$ with *last*($i_2$) $= i_2$, which is such that the path of $T_j$ from $j$ to nodes $i$ of $T_j$ with *last*($i$) $= i$ in the current multipath does not pass through $i_2$. We similarly continue with nodes $i_3$, $i_4$, etc. (The overhead for selecting the nodes $i_q$ in this order is $O(k)$, where $k$ is the number of nodes in $T_j$, by using the data structure that maintains the tree $T$ mentioned earlier.) The resulting connection sequence will be properly ordered and by Prop. 4, eventually (within at most $k$ connection operations), there will be no more nodes $i \in T_j$ left with $i \notin P$ and *last*($i$) $= i$. We call this connection sequence, a *complete connection sequence at* $j$. Since the number of arithmetic operations for a connection operation at a node $i$ is proportional to the number of outgoing arcs of $i$, and there are at most $k - 1$ arcs in $T_j$, it is seen that the running time of the complete connection sequence at $j$ is $O(k)$.

The special case where $j = 1$ and the tree $T_j$ is the entire shortest path tree, is of particular interest. It is seen that a complete connection sequence at the origin 1 has running time $O(n)$ and ends with a complete multipath.

Consider now the auction algorithm with graph reduction, modified so that at the end of each first scan iteration, we perform a complete connection sequence at the origin 1 that restores the completeness of the multipath. We call this, the *auction algorithm with multipath restructuring*. There are no failed extension cycles in this algorithm because the multipath is complete at the end of all iterations, so the required computation for contraction and extension cycles is $O(n^2)$, as argued in the preceding section. The computation for first scan iterations and for multipath restructuring is also $O(n^2)$. We have thus proved the following proposition.

PROPOSITION 5. *The running time of the auction algorithm with multipath restructuring is $O(n^2)$.*

Instead of performing a complete connection sequence at the origin after each first scan iteration, one may consider implementations of the multipath restructuring process that are more practically efficient. For example, by using various data structures one can calculate the list of tree nodes $i \notin P$ such that *last*($i$) is not a border node following a first scan operation and use this list to order appropriately the ensuing connection operations.

*Path Scanning*

An alternative approach to ensure that each extension cycle will be successful, is to allow the multipath $M$ to be incomplete at the end of an iteration, but whenever an impending failed extension cycle is detected, to appropriately restructure the multipath by means of complete connection sequences at certain nodes. In particular, we consider an algorithm which is the same as the auction algorithm with graph reduction except that if for the terminal node $i$ of $P$ we find that *last*($j$) is not a border node for all nodes $j$ attaining the minimum in the expression

$$\min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\},$$

we do not perform the contraction or extension as per Step 3 or Step 4, respectively, of the auction algorithm with graph reduction. Instead, we perform a complete connection sequence at each of the nodes $j$ such that $(i, j)$ belongs to the reduced graph and *last*($j$) is not a border node, and then decide via Step 2 of the auction algorithm with graph reduction whether a contraction or an extension will be performed. Thus Steps 2, 3, and 4 of the auction algorithm with graph reduction are replaced by the following modified versions.

*Modifications for the Path Scanning Algorithm*

1. **Modified Step 2: (Decide on Contraction, Extension, or Connection Sequence)** If *last*($j_i$) is not a border node for all nodes $j_i$ such that

$$j_i = \arg \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\},$$

perform a complete connection sequence at each node $j$ such that $(i, j)$ belongs to the reduced graph and *last*($j$) is not a border node. If

$$p_i < \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\},$$

go to Step 3; else go to Step 4.

2. **Modified Step 3: (Contraction)** Set

$$p_i := \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\},$$

and if $i \neq 1$, contract $P$. Go to the next iteration.

3. **Modified Step 4: (Extension)** Extend $P$ by one of the nodes $j_i$ such that

$$j_i = \arg \min_{(i,j)\in\mathcal{A}_r} \{a_{ij} + p_j\}.$$

If the number of the tree nodes is equal to $n$, then stop: the set of the tree arcs defines a shortest path tree. Otherwise, go to the next iteration.

The algorithm that uses the above modified Steps 2, 3, and 4 in place of Steps 2, 3, and 4 of the auction algorithm with graph reduction, is called the *auction algorithm with path scanning* (since prior to a contraction or extension, it scans the possible extension paths checking whether the last node on these paths is a border node). We have the following proposition.

PROPOSITION 6. *The running time of the auction algorithm with path scanning is* $O(n^2)$.

PROOF. Consider the subtrees $T_j$ that are involved in the complete connection sequences that occur between two successive first scan iterations. We prove by contradiction that these subtrees are all disjoint. Indeed, assume that there is a node $i$ that belongs to two subtrees $T_j$ and $T_{j'}$ such that complete connection sequences were performed at nodes $j$ and $j'$ at iterations $\tau$ and $\tau'$, respectively, where $\tau < \tau'$, and no first scan iteration occurred between iterations $\tau$ and $\tau'$. Since by construction of the algorithm, each extension is followed by other extensions culminating in a first scan iteration, it follows that only contractions can occur between iterations $\tau$ and $\tau'$. Thus the path from $j'$ to $i$ on the shortest path tree passes through $j$, and if $\{j', j_1, j_2, \ldots, j_k, j\}$ is the portion of this path that connects $j'$ to $j$, it can be seen that the complete connection sequence at $j$ was followed by contractions at $j_k, j_{k-1}, \ldots, j_1, j'$ in that order. By construction of the algorithm, following a contraction at any node $k$, the node *last*$(k)$ is a border node, so following the contraction at $j'$, the node *last*$(j')$ is a border node. This is a contradiction since in order for a complete connection sequence at $j'$ to occur, *last*$(j')$ must not be a border node.

Having proved that the subtrees $T_j$ involved in the complete connection sequences that occur between two successive first scan iterations are disjoint, it can be seen that the time required for all these complete connection sequences is $O(n)$, showing that the total overhead for multipath restructuring is $O(n^2)$. Since as before, the time for successful extension sequences, contraction sequences, and first scan iterations is $O(n^2)$, the result follows. □

By comparing the path scanning and the multipath restructuring approaches, we observe that both approaches guarantee a running time of $O(n^2)$ when combined with the auction algorithm with graph reduction. However, the version based on path scanning differs from the version using multipath restructuring since only partial restructuring of the multipath is performed to the extent needed.

## 5. Implementation Issues and Computational Results

In this section we describe briefly the implementation of some of the auction algorithms with graph reduction and we give some experimental results. All of our implemented forward auction codes use the following data structures:

(a) A linked list that stores the arcs of the graph and allows one to scan sequentially the "forward star" (set of outgoing arcs) of each node. This list is modified as arcs are getting deleted. (Actually, in our implementations, we postpone the removal of an arc from the linked list until the first subsequent time that its start node becomes the terminal node of $P$. It turns out that it is more convenient to update the linked list at that time, while the outgoing arcs of the start node are scanned.)
(b) An array that stores the start node and an array that stores the end node of each arc.
(c) An array that stores the candidate arc of each tree node.

We have selectively employed two techniques for improving performance. The first is to maintain, in addition to the candidate arc, the "second best" outgoing arc of a tree node; this technique has been used in all the forward codes and can save computation by executing many of the contraction steps without scanning all the outgoing arcs of the terminal node (see [4] and [5]). The second technique is to maintain for each node $i$ of the multipath, the node $last(i)$ as described in Section 4; this allows us to extend quickly the path $P$ from $i$ to $last(i)$, thereby effectively compressing a whole extension cycle into a single extension.

We have also implemented a forward/reverse version of the auction algorithm with graph reduction. This implementation is identical to the forward/reverse auction code published in [5], except that its forward portion uses the error bounds to delete nodes and arcs exactly as in the algorithm of Section 3. In the reverse portion of the algorithm there is no graph reduction. The switch from forward to reverse and back is controlled in a way that polynomial complexity is preserved. Regarding data structures, we also use in addition a linked list that stores the "backward star" of each node, that is, the set of incoming arcs to the node. The "second best" data structure was not used in the forward/reverse code, because it requires that prices increase monotonically, which is not the case in forward/reverse methods.

*Summary of Results*

We have experimented with two random problem generators: the public domain NETGEN generator [9], and a generator called COMPLGEN, which simply introduces each of the $n(n-1)$ possible arcs with a specified probability and assigns to it an integer length from a given range according to a uniform distribution. We have experimented with two types of problems:

(1) All destination problems with varying degrees of density. For such problems we found that forward auction algorithms with graph reduction perform better than their closest competitors for dense problems but worse for relatively sparse problems.
(2) Few destination problems. For such problems we found that forward/reverse auction algorithms with graph reduction perform much better than their closest competitors, without suffering from the pseudopolynomial behavior of their counterparts that do not use graph reduction.

*Experiments with Dense Single Origin/All Destinations Problems*

We have tested four different FORTRAN auction/shortest path codes for the single origin/all destinations problem:

*Table 1.* Average solution times of 20 runs in hundredths of a second on a Mac IIsi for randomly generated fully dense single origin/all destinations shortest path problems.

| $N$ | S-HEAP | ASP-R | ASP-R2 | ASP-R2-Last | ASP-R-PScan |
|---|---|---|---|---|---|
| 100 | 14.5 | 13.8 | 13.0 | 13.3 | 13.6 |
| 200 | 53.5 | 48.7 | 47.3 | 47.8 | 48.6 |
| 300 | 117.3 | 102.9 | 101.5 | 102.8 | 102.8 |
| 400 | 205.1 | 176.8 | 176.2 | 177.7 | 176.9 |

(1) **ASP-R:** This is the auction algorithm with graph reduction as described in Section 3.
(2) **ASP-R2:** This is the same as ASP-R except that it uses the "second best" outgoing arc data structure.
(3) **ASP-R2-Last:** This is the same as ASP-R2 except that it uses in addition the $last(i)$ array.
(4) **ASP-R-PScan:** This is the path scanning algorithm of Section 4.

We have compared these codes with the state-of-the-art FORTRAN code S-HEAP from [8]. This is a label setting code that uses a binary heap, and has outperformed all other label setting and label correcting codes in the tests of [8] for fully dense randomly generated graphs. All the codes were run on a Macintosh IIsi under System 7.0 using the Absoft compiler. We have found that for the single origin/all destinations problem, all of the above codes are superior to auction codes that do not use graph reduction (forward or combined forward/reverse).

We have generally found that for all-destination problems, the auction algorithms with graph reduction outperform by a small margin S-HEAP for relatively dense problems. We have also found that for all-destination problems, the relative performance of forward auction deteriorates as the graph density decreases, and the above auction codes are slower (by as much as two times) than the S-HEAP code (see the subsequent Tables 2 and 3).

We present in Table 1 some results with fully dense graphs. We have generated such graphs by independently selecting each arc length as an integer from the range [1, 1000] according to a uniform distribution. We have found that the arc length range does not affect materially the running times of various algorithms; for example if the cost range is [1, 10000], the running times grow by no more than 3%.

### *Experiments with Single Origin/Few Destinations Problems*

For problems with few destinations, forward/reverse auction codes without graph reduction have proved to be extremely efficient for many types of problems [4], but are susceptible to pseudopolynomial behavior in the presence of cycles with small positive lengths as illustrated in Fig. 1. Our experiments show that when graph reduction is introduced in these codes, it eliminates the difficulties due to small length cycles and results in superior performance for a much broader range of problems with few destinations. Thus, for few destination problems, forward/reverse auction algorithms with graph reduction perform much better than their closest competitors, without suffering from the pseudopolynomial behavior of their counterparts that do not use graph reduction.

*Table 2.* Solution times in seconds on a Mac IIci for NETGEN problems without any extra bidirectional arcs. Each entry gives the times of the corresponding method for One destination/10 destinations/All destinations.

| $N$ | $A$ | S-HEAP | ASPFR-NR | ASPFR-R | ASP-R2 |
|---|---|---|---|---|---|
| 2000 | 8000 | .317/.350/.367 | .050/.067/.917 | .055/.075/.925 | .567/.667/.667 |
| 2000 | 20000 | .450/.483/.500 | .117/.150/1.68 | .133/.217/1.30 | .683/.750/.816 |
| 3000 | 12000 | .050/.550/.550 | .033/.083/1.43 | .033/.117/1.43 | .800/.983/1.02 |
| 3000 | 30000 | .583/.717/.767 | .017/.183/2.45 | .033/.267/2.03 | .800/1.08/1.30 |
| 4000 | 16000 | .617/.767/.767 | .050/.100/1.88 | .050/.135/1.95 | 1.00/1.40/1.43 |
| 4000 | 40000 | .200/1.00/1.05 | .017/.183/3.33 | .033/.267/2.82 | .183/1.55/1.80 |
| 5000 | 20000 | .700/.983/.983 | .033/.133/2.45 | .033/.167/2.45 | 1.08/1.83/1.83 |
| 5000 | 50000 | .283/1.15/1.35 | .017/.200/4.63 | .017/.267/3.57 | .283/1.68/2.28 |

To support this assessment, we present some results using problems obtained using the NETGEN program. Problems were generated by specifying the number of nodes $N$, the number of arcs $A$, the length range (chosen to be [1, 1000] in all our experiments), and a single source and sink (automatically chosen by NETGEN to be nodes 1 and $N$). For each problem generated by NETGEN, an additional modified version was generated by adding 100 bidirectional arcs of length 1, thus creating a problem with many small-length cycles that are likely to induce pseudopolynomial behavior for auction algorithms without graph reduction.

We tested three auction codes, in addition to S-HEAP:

(1) **ASPFR-NR:** This is the forward/reverse auction algorithm without graph reduction given in [5].
(2) **ASPFR-R:** This is the forward/reverse auction algorithm with graph reduction as described above.
(3) **ASP-R2:** This is the forward auction with graph reduction that uses the "second best" outgoing arc data structure.

All the codes were compiled with the Absoft compiler for the Apple Macintosh and were run on a Macintosh IIci under System 6.0.8. The times are shown in Tables 2 and 3, for the cases of one destination (node $N$ in all problems), 10 destinations (nodes $N - 100i$ and $N/2 - 100i$ for $i = 0, 1, 2, 3, 4$ in all problems), and all destinations.

In summary the results are as follows:

For the problems without the extra bidirectional unit length arcs (cf. Table 2), forward/reverse auction with and without graph reduction run very close. The forward auction code ASP-R2 is uniformly worse than S-HEAP for few and many destinations (by roughly a factor of 2). S-HEAP is far worse than the forward/reverse codes for few destinations, but its relative performance improves as the number of destinations increases. For all-destination problems S-HEAP becomes better than the forward/reverse codes (by a factor up to 3).

For the problems with the extra bidirectional unit length arcs (cf. Table 3), forward/reverse auction without graph reduction runs much slower than auction with graph reduction for any number of destinations, and sometimes even slower than both S-HEAP and ASP-R2 for few destinations. The forward auction code ASP-R2 continues to be uniformly worse than S-HEAP for few and many destinations. Both ASP-R2 and HEAP seem unaffected by the small length cycles. Generally, we have found that the performance of the forward/reverse code with graph reduction with few destinations deteriorates as the number of small length

*Table 3.* Solution times in seconds on a Mac IIci for NETGEN problems with 100 extra bidirectional arcs of unit length. Each entry gives the times of the corresponding method for One destination/10 destinations/All destinations. The number of arcs given in the column under $A$ includes the 200 extra unit length arcs.

| $N$ | $A$ | S-HEAP | ASPFR-NR | ASPFR-R | ASP-R2 |
|---|---|---|---|---|---|
| 2000 | 8200 | .267/.367/.367 | .417/.683/9.47 | .133/.200/.983 | .367/.617/.700 |
| 2000 | 20200 | .500/.500/.533 | .317/.983/11.27 | .167/.375/1.53 | .783/.783/.850 |
| 3000 | 12200 | .517/.567/.567 | .150/.400/11.65 | .083/.183/1.57 | .867/1.02/1.03 |
| 3000 | 30200 | .633/.750/.783 | .017/.767/12.87 | .017/.450/2.20 | .883/1.13/1.32 |
| 4000 | 16200 | .667/.767/.785 | .217/1.17/14.45 | .117/.450/2.13 | 1.08/1.45/1.45 |
| 4000 | 40200 | .250/1.05/1.08 | .167/1.18/19.85 | .099/.551/3.05 | .233/1.65/1.83 |
| 5000 | 20200 | .767/1.00/1.00 | .117/.717/15.50 | .117/.367/2.63 | 1.18/1.90/2.65 |
| 5000 | 50200 | .367/1.25/1.38 | .050/1.53/24.12 | .050/.683/3.85 | .367/1.87/2.32 |

cycles increases (basically, graph reduction protects the forward portion from pseudopolynomial behavior but not the reverse portion). However, forward/reverse auction with graph reduction continues to dominate substantially S-HEAP for few destinations.

## Appendix

In this appendix we focus on a sample run of the algorithm and we show that the number of failed extension cycles of the auction algorithm with graph reduction is $O(\min\{m, n \log n\})$. We first recall the definition of candidate arc that was introduced in Section 4.

Following each contraction or extension at the terminal node $i$, there are one or more arcs $(i, j)$ such that $p_i = a_{ij} + p_j$. We arbitrarily select one of these arcs and call it the *candidate arc of* $i$, until the next iteration when a contraction or an extension at $i$ occurs, and a possibly different arc becomes the candidate arc of $i$. If an arc $(i, j)$ is deleted while it is simultaneously a candidate arc and a border arc, it is called *candidate-deleted.*

Candidate-deleted arcs are interesting for our purposes because their deletion can cause failed extension cycles. In particular, a failed extension cycle is a sequence of successive extensions at the end of which the terminal node of the path $P$ is a tree node $i$ satisfying $p_i < a_{ij} + p_j$ for all arcs $(i, j)$ of the reduced graph. Consider the preceding time, say iteration $\tau$, when $i$ was the terminal node of the path $P$ (prior to the given failed extension cycle). At that iteration, a contraction at $i$ occurred, and $p_i$ was set to satisfy $p_i = a_{ij} + p_j$ for some candidate arc $(i, j)$. We call this candidate arc and the corresponding iteration $\tau$, the *critical arc* and the *critical iteration* of the failed extension cycle, respectively. We have the following lemma.

LEMMA A.1. *Let $(i, j)$ be the critical arc of a failed extension cycle. Then $(i, j)$ is candidate-deleted. Furthermore, $j$ became a tree node at some iteration between the critical iteration of $(i, j)$ and the final iteration of the failed extension cycle.*

PROOF. Let iterations $\tau$ and $\overline{\tau}$ be the critical and the final iteration of the failed extension cycle, respectively. At the end of iteration $\tau$ we have $p_i = a_{ij} + p_j$ for the critical arc $(i, j)$, while at the start of iteration $\overline{\tau}$ we had $p_i < a_{ij} + p_j$, so $p_j$ was increased prior to iteration $\overline{\tau}$, implying that $j$ was a tree node at iteration $\overline{\tau}$. If $(i, j)$ belonged to the reduced graph at the start of iteration $\overline{\tau}$, then $(i, j)$ must have been the unique tree arc incoming to $j$. Hence immediately following the price increase of $p_j$ and the associated contraction, node $i$ became the terminal node of $P$. This, however, is a contradiction since $\overline{\tau}$ was the first iteration when $i$ became the terminal node of $P$ subsequent to the critical iteration $\tau$. Therefore $(i, j)$ must have been deleted prior to iteration $\overline{\tau}$. Let $\tilde{\tau}$ be the iteration at which $(i, j)$ was deleted. At that iteration, $i$ was a tree node (since $\tau < \tilde{\tau}$), while $j$ was not a tree node (since no incoming tree arc of a tree node can be deleted). Hence $(i, j)$ was a border arc at iteration $\tilde{\tau}$, and it was also a candidate arc since it was a candidate arc at iteration $\tau$ and $i$ did not again become the terminal node of $P$ until iteration $\overline{\tau}$. Thus, the arc $(i, j)$ is candidate-deleted. □

Lemma A.1 also shows that an arc cannot become critical in connection with more than one failed extension cycle. Thus there is a distinct critical arc associated with each failed extension cycle, thereby showing that the number of failed extension cycles is at most $m$. We state this formally.

COROLLARY A.1. *The number of failed extension cycles is at most m.*

Since by Lemma A.1, a critical arc is candidate-deleted, it will suffice for our purposes to show that the number of candidate-deleted arcs is $O(n \log n)$; we will do this in Lemma A.6 below, after some preparatory analysis.

For an arc $(i, j)$ to get deleted while it is a border arc, a first scan iteration must occur at a node $k$ with $u_k + a_{kj} < u_j$ [cf. Eqs. (6) and (7)]. We call node $k$ the *dominating node* of this arc. We also write $i \prec j$ if node $i$ became a tree node before node $j$. We have:

LEMMA A.2. *If $k$ is the dominating node of an arc $(i, j)$ that was deleted while it was a border arc, then $i \prec k \prec j$.*

PROOF. Arc $(i, j)$ was deleted when a first scan iteration occured at the dominating node $k$. At that iteration, $(i, j)$ was a border arc, so $i$ was a tree node, while $j$ was not a tree node. This implies that $i \prec k \prec j$. □

The following lemma gives a basic property.

LEMMA A.3. *Consider a candidate-deleted arc $(i, j)$ that was deleted at iteration $\tau$, and suppose that, after the deletion of $(i, j)$, $i$ became the terminal node of $P$ for the first time at iteration $\overline{\tau}$. Then the price of $j$ increased at some iteration between $\tau$ and $\overline{\tau}$, and hence $j$ became a tree node prior to $\overline{\tau}$.*

PROOF. Let $p$ be the price vector at the end of iteration $\tau$ and let $\overline{p}$ be the price vector at the start of iteration $\overline{\tau}$. Let also $d_i$ denote the shortest distance from 1 to $i$. We argue by contradiction. If $p_j = \overline{p}_j$, then since $p_i = \overline{p}_i$ and $p_i = a_{ij} + p_j$, we must have $\overline{p}_i = a_{ij} + \overline{p}_j$. Since $d_i = \overline{p}_1 - \overline{p}_i$, it follows that $d_i + a_{ij} = \overline{p}_1 - \overline{p}_j \leq d_j$, where the last inequality follows from the CS property (1a) and (1b) (the price differential between two nodes is a lower bound to the shortest distance between the two nodes). Hence the path $P_j$ that consists of a shortest path from 1 to $i$ followed by arc $(i, j)$ is a shortest path from 1 to

$j$. But this is a contradiction, since for arc $(i, j)$ to be deleted, the path $\overline{P}_j$ consisting of a shortest path from 1 to the dominating node $k$ of arc $(i, j)$ followed by arc $(k, j)$ must be shorter than $P_j$. □

Let $(i, j)$ be a candidate-deleted arc. Following the deletion of $(i, j)$, node $i$ may have become again the terminal node of $P$ and have acquired a new candidate arc $(i, l)$, which is also candidate-deleted. If $(i, l)$ is the first such arc to be deleted, we say that $(i, l)$ is the *candidate-deleted arc of node i next to* $(i, j)$. The following lemma establishes an important ordering property.

LEMMA A.4. *Let $(i, j)$ and $(i, l)$ be candidate-deleted arcs of node $i$ with dominating nodes $k$ and $r$, respectively. Suppose that $(i, l)$ is the candidate-deleted arc of node $i$ next to $(i, j)$. Then $i \prec k \prec j \prec r \prec l$.*

PROOF. From Lemma A.2, we have $i \prec k \prec j$ and $i \prec r \prec l$. Also from Lemma A.3, we have that $j$ was a tree node when $(i, l)$ became a candidate and border arc, which in turn occurred prior to the time the dominating node $r$ was first scanned and $(i, l)$ was deleted. Hence $j \prec r$. □

In the course of the algorithm, as long as a node $j$ is a border node, it has a unique incoming border arc. When this arc, call it $(i, j)$, is deleted, it is replaced by arc $(k, j)$, where $k$ is the dominating node of $(i, j)$. Consider the sequence of all arcs that become incoming border arcs of $j$

$$(i_1, j), (i_2, j), \ldots, (i_p, j), \tag{A.1}$$

where for $q = 2, \ldots, p$, $i_q$ is the dominating node of arc $(i_{q-1}, j)$, and $(i_p, j)$ is the arc that eventually becomes the unique tree arc that is incoming to $j$. Given a candidate-deleted arc $(i_q, j)$ from this sequence, we will define its *succession pair*. This is the pair $[i_q, i_{\overline{q}}]$, where $\overline{q} = p$ if $(i_q, j)$ is the last candidate-deleted arc in the sequence, and otherwise $\overline{q}$ is the first integer $q'$ with $q < q' < p$ such that $(i_{q'}, j)$ is candidate-deleted.

LEMMA A.5. *The following hold true:*

*(a) For the succession pair $[i_q, i_{\overline{q}}]$ of a candidate-deleted arc $(i_q, j)$ we have $i_q \prec i_{\overline{q}} \prec j$.*
*(b) Let $(i, j)$ and $(i, l)$ be candidate-deleted arcs such that $(i, l)$ is the candidate-deleted arc of node $i$ next to $(i, j)$. If $[i, \overline{k}]$ and $[i, \overline{r}]$ are the succession pairs corresponding to $(i, j)$ and $(i, l)$, respectively, then $i \prec \overline{k} \prec j \prec \overline{r} \prec l$.*

PROOF. (a) Consider the sequence (A.1) of border arcs that are incoming to node $j$. By Lemma A.2 we have $i_1 \prec i_2 \prec \cdots \prec i_p \prec j$, and by using the definition of succession pair the result follows.
(b) Let $k$ and $r$ be the dominating nodes of $(i, j)$ and $(i, l)$, respectively. By Lemma A.4, we have $i \prec k \prec j \prec r \prec l$, while by the definition of a succession pair, we have $\overline{k} \prec j$ and $k = \overline{k}$ or $k \prec \overline{k}$. Therefore, $i \prec \overline{k} \prec j$. Similarly, we have $\overline{r} \prec l$ and $r = \overline{r}$ or $r \prec \overline{r}$, implying that $j \prec \overline{r} \prec l$. □

We now use the following lemma, which is stated in [6], p. 208.

**Combinatorial Lemma**: Let $N$ and $M$ be positive integers. Consider a process that modifies a set $S$ of integer pairs $[p, q]$, which is a subset of the set $\{[p, q] \mid 1 \le p < q \le N\}$. Initially $S$ is either empty or else contains at most one pair $[p, q]$ for every $p$. The process consists of $M$ stages. During a stage some pairs may leave $S$ and at its end some pairs may enter $S$. The leaving pairs constitute a chain of the form

$$[p_1, p_2], [p_2, p_3], \ldots$$

Each entering pair $[p, r]$ must satisfy the following two requirements:

(1) No pair of the form $[p, q]$ is currently in $S$.
(2) If a pair of the form $[p, q]$ was earlier in $S$ and left at the same stage as a pair $[p', q']$, then $q' < r$.

Then the number of pairs that left $S$ during the $M$ stages is

$$O\left(\frac{(M+N)\log(M+N)}{\log\left(\lceil \frac{M+N}{N} \rceil\right)}\right).$$

We apply the Combinatorial Lemma using the following associations. Each considered pair $[i, k]$ is a succession pair, so $N = n$. In addition, the $j$th stage corresponds to the first scanning of node $j$, so $M = n$ too. Initially, the set $S$ contains one successor pair $[i, k]$ for each node $i$ for which a succession pair exists; this is the pair for which $k$ is smallest. Then, during the $j$th stage, for each candidate-deleted arc of the form $(i, j)$, the associated succession pair $[i, k]$ exits $S$ and, at the end of the stage, the succession pair corresponding to the candidate-deleted arc of $i$ next to $(i, j)$ enters $S$ (if such an arc exists).

LEMMA A.6. *The number of candidate-deleted arcs is $O(n \log n)$.*

PROOF. Without loss of generality, we will assume that the nodes of the graph became tree nodes in the order corresponding to their values, that is $i \prec j$ if the integer $i$ is less than the integer $j$. The proof consists of applying the Combinatorial Lemma using the associations described above. By Lemma A.5(a), the succession pairs leaving $S$ during the $j$th stage form a chain of the form $[p_1, p_2], [p_2, p_3], \ldots$, and all nodes $p$ involved in the chain satisfy $p < j$. Since, by Lemma A.5(b), for all pairs $[p, q]$ entering $S$ during the $j$th stage we have $j < q$, condition (2) of the Combinatorial Lemma is satisfied. For each node $i$, let $[i, k_1], [i, k_2], \ldots, [i, k_p]$ be the succession pairs with first node $i$, ordered so that $k_1 < k_2 < \cdots < k_p$. Then the corresponding candidate-deleted arcs $(i, j_1), (i, j_2), \ldots, (i, j_p)$ satisfy $j_1 < j_2 < \cdots < j_p$, by Lemma A.5(b). By assumption, $[i, k_1]$ is initially in $S$, and for $q = 1, \ldots, p-1$, the succession pair $[i, k_{q+1}]$ will enter $S$ at stage $j_q$ when its predecessor pair $[i, k_q]$ exits $S$. It follows that, for each $i$, only one pair of the form $[i, k]$ can be in $S$ at any stage, and condition (1) of the Combinatorial Lemma is satisfied. Thus all the hypotheses of the Combinatorial Lemma hold and, since $M = N = n$, the conclusion is that the number of pairs leaving S is $O(n \log n)$. Furthermore, by construction, every succession pair will exit $S$ exactly once, so that the number of candidate-deleted arcs is also $O(n \log n)$. □

From Corollary A.1 and Lemma A.6, it can be seen that the number of failed extension cycles is $O(\min\{m, n \log n\})$, as claimed.

## References


1. D.P. Bertsekas, "A Distributed Algorithm for the Assignment Problem," Lab. for Information and Decision Systems Working Paper, M.I.T., Cambridge, MA, March 1979.
2. D.P. Bertsekas, "A New Algorithm for the Assignment Problem," Math. Programming, vol. 21, pp. 152–171, 1981.
3. D.P. Bertsekas, "The Auction Algorithm for Assignment and Other Network Flow Problems: A Tutorial," Interfaces, vol. 20, pp. 133–149, 1990.
4. D.P. Bertsekas, "The Auction Algorithm for Shortest Paths," SIAM J. on Optimization, vol. 1, pp. 425–447, 1991.
5. D.P. Bertsekas, "Linear Network Optimization: Algorithms and Codes," M.I.T. Press, Cambridge, MA, 1991.
6. Z. Galil and A. Naamad, "$O(VE \log^2 V)$ Algorithm for the Maximum Flow Problem," J. of Comput. Sys. Sci., vol. 21, pp. 203–217, 1980.
7. G. Gallo and S. Pallottino, "Shortest Path Methods: A Unified Approach," Math. Programming Study, vol. 26, pp. 38–64, 1986.
8. G. Gallo and S. Pallottino, "Shortest Path Algorithms," Annals of Operations Research, vol. 7, pp. 3–79, 1988.
9. D. Klingman, A. Napier, and J. Stutz, "NETGEN—A Program for Generating Large Scale (Un) Capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems," Management Science, vol. 20, pp. 814–822, 1974.
10. C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, N.J., 1982.
11. S. Pallottino and M.G. Scutellà, "Strongly Polynomial Algorithms for Shortest Paths," Ricerca Operativa, vol. 60, pp. 33–53, 1991.