SIAM J. OPTIMIZATION
Vol. 1, No. 4, pp. 425-447, November 1991

AN AUCTION ALGORITHM FOR SHORTEST PATHS*

DIMITRI P. BERTSEKAS†

Abstract. A new and simple algorithm for finding shortest paths in a directed graph is proposed. In the single origin-single destination case, the algorithm maintains a single path starting at the origin, which is extended or contracted by a single node at each iteration. Simultaneously, at most one dual variable is adjusted at each iteration so as to either improve or maintain the value of a dual function. For the case of multiple origins, the algorithm is well suited for parallel computation. It maintains multiple paths that can be extended or contracted in parallel by several processors that share the results of their computations. Based on experiments with randomly generated problems on a serial machine, the algorithm substantially outperforms its closest competitors for problems with few origins and a single destination. It also seems better suited for parallel computation than other shortest path algorithms.

Key words. shortest path, network optimization, auction, parallel algorithms

AMS(MOS) subject classifications. primary 90C47; secondary 90C05

1. Introduction. In this paper we propose a new algorithm for finding shortest paths in a directed graph $(\mathcal{N}, \mathcal{A})$. For the single origin and single destination case, our 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 deleting its terminal node. When the destination becomes the terminal node of the path, the algorithm terminates.

To get an intuitive sense of the algorithm, think of a mouse moving in a graphlike maze, trying to reach a destination. The mouse criss-crosses the maze, either advancing or backtracking along its current path. Each time the mouse backtracks from a node, it records a measure of the desirability of revisiting and advancing from that node in the future (this will be represented by a price variable—see § 2). The mouse revisits and proceeds forward from a node when the node's measure of desirability is judged superior to those of other nodes. Our algorithm efficiently emulates this mouse search process using simple data structures.

In a parallel computing environment, the problem of multiple origins with a single destination can be solved by running in parallel a separate version of the algorithm for each origin. However, the different parallel versions can help each other by sharing the interim results of their computations, thereby substantially enhancing the algorithm's performance. The recent Master's thesis [Pol91] discusses a number of parallel asynchronous implementations of our algorithm, and reports on simulations suggesting a significant speedup potential. Generally, our algorithm seems better suited for parallel computation than all of its competitors.

The practical performance of the algorithm and its numerous variations remain to be fully investigated, particularly using parallel machines. Preliminary experimental results with randomly generated problems on a serial machine, and a comparison with the state-of-the-art shortest path codes of Gallo and Pallotino [GaP88] have been very encouraging. In particular, a code implementing one version of our algorithm outperforms by a large margin its closest competitors for the case of few origins and one

* Received by the editors August 30, 1990; accepted for publication (in revised form) March 28, 1991. This research was supported by National Science Foundation grant DDM-8903385 and by Army Research Office grant DAAL03-86-K-0171.

† Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

destination; see § 7. In a parallel computing environment, the relative advantage of our algorithm should increase, but this remains to be verified in future work.

The worst case running time of the algorithm is pseudopolynomial; it depends on the shortest path lengths. This in itself is not necessarily bad. Dial's algorithm (see [Dia69], [DGK79], [AMO89], [GaP88]) is also pseudopolynomial, yet its running time in practice is excellent, particularly for a small range of arc lengths. Another popular method, the D'Esopo–Pape algorithm [Pap74], has exponential worst case running time [Ker81], [ShW81], yet it performs very well in practice [DGK79], [GaP88]. Nonetheless, under mild conditions, our algorithm can be turned into a polynomial one by using the device of arc length scaling. However, in our computational experiments, this scaling device was entirely unnecessary, and, in fact, degraded the algorithm's performance.

To place our algorithm in perspective, we note that shortest path methods are traditionally divided into two categories: label setting (Dijkstra-like) and label correcting (Bellman–Ford-like); see the surveys given in [AMO89], [GaP86], [GaP88], and the references quoted there. Our algorithm shares features from both types of algorithms. It resembles label setting algorithms in that the shortest distance of a node is found at the first time the node is labeled (becomes the terminal node of the path in our case). It resembles label correcting algorithms in that the label of a node may continue to be updated after its shortest distance is found.

As we explain in § 6, our method may be viewed as a dual coordinate ascent or relaxation method. In reality, the inspiration for the algorithm came from the author's auction and ε-relaxation methods [Ber79], [Ber86] (extensive descriptions of these methods can be found in [Ber88], [BeE88], [BeT89], and [Ber90]). If one applies the ε-relaxation method for a minimum cost flow formulation of the shortest path problem (see § 6), but with the important difference that $\varepsilon = 0$, then one obtains an algorithm which is very similar to the one provided here.

Our algorithm may also be viewed as a special case of the so called *naive auction algorithm*, applied to a special type of assignment problem, which is derived from the shortest path problem (see, e.g., [Law76, p. 186]). The naive auction algorithm, first proposed by Bertsekas in [Ber81] as part of the relaxation method for the assignment problem, and also discussed more recently in the tutorial paper [Ber90], is the same as the auction algorithm, except that the parameter ε that controls the accuracy of the solution is set to zero. The naive auction algorithm is not guaranteed to solve general assignment problems, and is primarily useful as an initialization method for other assignment algorithms, such as relaxation (as described in [Ber81]) or sequential shortest path (as described in [JoV87]). Nevertheless, it is guaranteed to solve the special type of assignment problem, which is relevant to the shortest path context of the present paper.

The paper is organized as follows: In § 2, we describe the basic algorithm for the single origin case and we prove its basic properties. In § 3, we develop the polynomial version of the algorithm using arc length scaling. In § 4, we describe various ways to improve the performance of the algorithm. In § 5, we consider the multiple origin case and we discuss how the algorithm can take advantage of a parallel computing environment. In § 6, we derive the connection with duality and we show that the algorithm may be viewed both as a naive auction algorithm and as a coordinate ascent (or Gauss–Seidel relaxation) method for maximizing a certain dual cost function. Finally, § 7 contains computational results.

2. Algorithm description and analysis.

We describe the algorithm in its simplest form for the single origin and single destination case, and we defer the discussion of other and more efficient versions for subsequent sections. Our main assumption is that *all cycles have positive length*, although we will see shortly that the initialization of the algorithm is greatly simplified if, in addition, all arc lengths are nonnegative.

To simplify the presentation, we will also assume that each node except for the destination has at least one outgoing incident arc; any node not satisfying this condition can be connected to the destination with a very high length arc without materially changing the problem and the subsequent algorithm. We also assume that there is at most one arc between two nodes in each direction, so that we can unambiguously refer to an arc (i, j). Again, this assumption is made for notational convenience; our algorithm can be trivially extended to the case where there are multiple arcs connecting a pair of nodes.

Let node 1 be the origin node and let t be the destination node. In the following, by a *path* we mean a sequence of nodes $(i_1, i_2, \cdots, i_k)$ such that (i_m, i_{m+1}) is an arc for all $m = 1, \cdots, k-1$. If, in addition, the nodes $i_1, i_2, \cdots, i_k$ are distinct, the sequence $(i_1, i_2, \cdots, i_k)$ is called a *simple path*. The length of a path is defined to be the sum of its arc lengths.

The algorithm maintains at all times a simple path $P = (1, i_1, i_2, \cdots, i_k)$. 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 $P = (1, i_1, i_2, \cdots, i_k)$ and (i_k, i_{k+1}) is an arc, *extending P by i_{k+1}* means replacing P by the path $(1, i_1, i_2, \cdots, i_k, i_{k+1})$, called the *extension of P by i_{k+1}*. If P does not consist of just the origin node 1, *contracting* P means replacing P with the path $(1, i_1, i_2, \cdots, i_{k-1})$.

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

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

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

We denote by p the vector of prices p_i. A pair (P, p) consisting of a simple path P and a price vector p that satisfies the above conditions is said to satisfy *complementary slackness* (or CS for short). (When we say that a pair (P, p) satisfies the CS conditions, we implicitly assume that P is simple.)

The CS terminology is motivated by a formulation of the shortest path problem as a linear minimum cost flow problem; see § 6. In this formulation, the prices p_i can be viewed as the variables of a problem which is dual in the usual linear programming duality sense. The complementary slackness conditions for optimality of the primal and dual variables can be shown to be equivalent to the conditions (1). For the moment, however, we ignore the linear programming context, and we simply note 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 (1b), $p_1 - p_i$ is the length of the portion of P between 1 and i, and by (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

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

When some arcs have negative lengths, an initial choice of a pair (P, p) satisfying CS may not be obvious or available, but § 4 provides a general method for finding such a pair.

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 strictly increased. A degenerate case occurs when the path consists by 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

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: (Contract path). Set

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

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

Step 2: (Extend path). Extend P by node j_i where

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

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

It can be seen that, 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.

Figure 1 provides an example of the operation of the algorithm. In this example, the terminal node traces the tree of shortest paths from the origin to the nodes that are closer to the origin than the given destination. We will see that this behavior is typical when the initial prices are all zero.

PROPOSITION 1. *The pairs (P, p) generated by the algorithm satisfy* CS. *Furthermore, for every pair of nodes i and j, and at all iterations, $p_i - p_j$ is an underestimate of the shortest distance from i to j.*

Proof. We first show by induction that (P, p) satisfies CS. Indeed, the initial pair satisfies CS by assumption. Consider an iteration that starts with a pair (P, p) satisfying CS and produces a pair $(\bar{P}, \bar{p})$. Let i be the terminal node of P. If

$$p_i = \min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}, \tag{5}$$

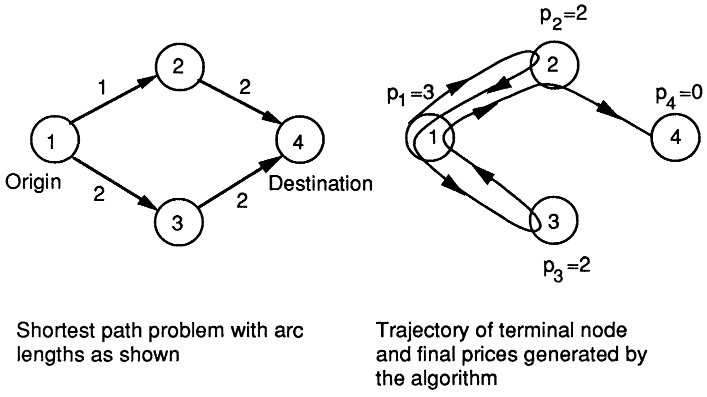
then $\bar{P}$ is the extension of P by a node j_i and $\bar{p} = p$, implying that the CS condition (1b) holds for all arcs of P as well as arc (i, j_i) (since j_i attains the minimum in (5); cf. condition (4)).

Suppose next that

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

Then if P is the degenerate path (1), the CS condition holds vacuously. Otherwise, $\bar{P}$ is obtained by contracting P, and for all nodes $j \in \bar{P}$, we have $\bar{p}_j = p_j$, implying conditions (1a) and (1b) for arcs outgoing from nodes of $\bar{P}$. Also, for the terminal node i, we have

$$\bar{p}_i = \min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\},$$


1
2
2
2
Origin
Destination
Shortest path problem with arc lengths as shown
p2=2
p1=3
p4=0
p3=2
Trajectory of terminal node and final prices generated by the algorithm

Iteration #	Path P prior to the iteration	Price vector p prior to the iteration	Type of action during the iteration
1	(1)	(0, 0, 0, 0)	contraction at 1
2	(1)	(1, 0, 0, 0)	extension to 2
3	(1, 2)	(1, 0, 0, 0)	contraction at 2
4	(1)	(1, 2, 0, 0)	contraction at 1
5	(1)	(2, 2, 0, 0)	extension to 3
6	(1, 3)	(2, 2, 0, 0)	contraction at 3
7	(1)	(2, 2, 2, 0)	contraction at 1
8	(1)	(3, 2, 2, 0)	extension to 2
9	(1, 2)	(3, 2, 2, 0)	extension to 4
10	(1, 2, 4)	(3, 2, 2, 0)	stop

FIG. 1. *An example illustrating the algorithm starting with* $P = (1)$ *and* $p = 0$.

implying condition (1a) for arcs outgoing from that node as well. Finally, since $\bar{p}_i > p_i$ and $\bar{p}_k = p_k$ for all $k \neq i$, we have $\bar{p}_k \leqq a_{kj} + \bar{p}_j$ for all arcs (k, j) outgoing from nodes $k \notin P$. This completes the induction proof.

Finally, consider any path from a node i to a node j. By adding the CS condition (1a) along the path, we see that the length of the path is at least $p_i - p_j$, proving the last assertion of the proposition. □

PROPOSITION 2. *If P is a path generated by the algorithm, then P is a shortest path from the origin to the terminal node of P.*

Proof. This follows from the CS property of the pair (P, p) shown in Proposition 1; see the remarks following the CS conditions (1). Furthermore, by the CS condition (1a), every path connecting 1 and i must have length at least equal to $p_1 - p_i$. □

2.1. Interpretation of the algorithm. The preceding propositions can be used to provide an intuitive interpretation of the algorithm. Denote for each node i

(6) $$D_i = \text{shortest distance from the origin 1 to node } i,$$

with $D_1 = 0$ by convention. By Proposition 1, we have, throughout the course of the algorithm,

$$p_1 - p_j \leqq D_j \quad \forall j \in \mathcal{N},$$

while by Proposition 2, we have

$$p_1 - p_i = D_i \quad \text{for all } i \text{ in } P.$$

It follows that

$$D_i+p_i-p_t \leqq D_j+p_j-p_t \quad \forall i \in P \quad \text{and} \quad j \in \mathcal{N}.$$

Since by Proposition 1, p_i-p_t is an estimate of the shortest distance from i to t, we may view the quantity

$$D_j+p_j-p_t$$

as an estimate of the shortest distance from 1 to t using only paths passing through j. Thus, intuitively, it makes sense to consider a node j as "eligible" for inclusion in the algorithm's path only if $D_j+p_j-p_t$ is minimal.

Based on the preceding interpretation, it can be seen that:

(a) The algorithm maintains a path consisting of "eligible" candidates for participation in a shortest path from 1 to t.

(b) The algorithm extends P by a node j if and only if j is an "eligible" candidate.

(c) The algorithm contracts P if the terminal node i has no neighbor which is "eligible." Then, the estimate of i's shortest distance to t is improved (i.e., is increased), and i becomes "ineligible" (since $D_i+p_i-p_t$ is not minimal anymore), thus justifying its deletion from P. Node i will be revisited only after $D_i+p_i-p_t$ becomes minimal again, following sufficiently large increases of the prices of the currently "eligible" nodes.

The preceding interpretation suggests also that the nodes become terminal for the first time in the order of the initial values $D_j+p_j^0-p_t^0$, where

(7) $$p_i^0 = \text{initial price of node } i.$$

To formulate this property, denote for every node i

(8) $$d_i = D_i + p_i^0.$$

Let us index the iterations by $1, 2, \cdots$, and let

(9) k_i = the first iteration index at which node i becomes a terminal node,

where, by convention, $k_1=0$ and $k_i=\infty$ if i never becomes a terminal node.

PROPOSITION 3. (a) *At the end of iteration k_i we have $p_1=d_i$.*

(b) *If $k_i<k_j$, then $d_i \leqq d_j$.*

Proof. (a) At the end of iteration k_i, P is a shortest path from 1 to i by Proposition 2, while the length of P is $p_1-p_i^0$.

(b) By part (a), at the end of iteration k_i, we have $p_1=d_i$, while at the end of iteration k_j, we have $p_1=d_j$. Since p_1 is monotonically nondecreasing during the algorithm and $k_i<k_j$, the result follows. □

Note that the preceding proposition shows that when all arc lengths are nonnegative, and the default initialization $p=0$ is used, the nodes become terminal for the first time in the order of their proximity to the origin.

2.2. Termination and running time of the algorithm. The following proposition establishes the validity of the algorithm.

PROPOSITION 4. *If there exists at least one path from the origin to the destination, the algorithm terminates with a shortest path from the origin to the destination. Otherwise the algorithm never terminates and $p_1 \to \infty$.*

Proof. Assume first that there is a path from node 1 to the destination t. Since by Proposition 1, p_1-p_t is an underestimate of the (finite) shortest distance from 1 to t, p_1 is monotonically nondecreasing, and p_t is fixed throughout the algorithm, p_1 must stay bounded. We next claim that p_i must stay bounded for all i. Indeed, in order to have $p_i \to \infty$, node i must become the terminal node of P infinitely often, implying (by Proposition 1) that $p_1 - p_i$ must be equal to the shortest distance from 1 to i infinitely often, which is a contradiction since p_1 is bounded.

We next show that the algorithm terminates finitely. Indeed, it can be seen with a straightforward induction argument that for every node i, p_i is either equal to its initial value, or else it is the length of some path starting at i plus the initial price of the final node of the path; we call this the *modified length* of the path. Every path starting at i can be decomposed into a simple path together with a finite number of cycles, each having positive length by assumption, so the number of distinct modified path lengths within any bounded interval is bounded. Now p_i was shown earlier to be bounded, and each time i becomes the terminal node by extension of the path P, p_i is strictly larger over the preceding time that i became the terminal node of P, corresponding to a strictly larger modified path length. It follows that the number of times i can become a terminal node by extension of the path P is bounded. Since the number of path contractions between two consecutive path extensions is bounded by the number of nodes in the graph, the number of iterations of the algorithm is bounded, implying that the algorithm terminates finitely.

Assume now that there is no path from node 1 to the destination. Then, the algorithm will never terminate, so by the preceding argument, some node i will become the terminal node by extension of the path P infinitely often and $p_i \to \infty$. At the end of iterations where this happens, $p_1 - p_i$ must be equal to the shortest distance from 1 to i, implying that $p_1 \to \infty$. $\square$

We will now estimate the running time of the algorithm, assuming that all the arc lengths and initial prices are integer. Our estimate involves the set of nodes

$$\mathcal{I} = \{i \mid d_i \leqq d_t\}; \tag{10}$$

by Proposition 3, these are the only nodes that ever become terminal nodes of the paths generated by the algorithm. Let us denote

$$I = \text{number of nodes in } \mathcal{I}, \tag{11}$$

$$G = \text{maximum out-degree (number of outgoing arcs) over the nodes in } \mathcal{I}, \tag{12}$$

and let us also denote by E the product

$$E = I \cdot G. \tag{13}$$

PROPOSITION 5. *Assume that there exists at least one path from the origin* 1 *to the destination t, and that the arc lengths and initial prices are all integer. The worst case running time of the algorithm is $O(E(D_t + p_t^0 - p_1^0))$.*

Proof. Each time a node i becomes the terminal node of the path, we have $p_i = p_1 - D_i$ (cf. Proposition 2). Since at all times we have $p_1 \leqq D_t + p_t^0$ (cf. Proposition 1), it follows that

$$p_i = p_1 - D_i \leqq D_t + p_t^0 - D_i,$$

and using the definitions $d_t = D_t + p_t^0$ and $d_i = D_i + p_i^0$, and the fact $d_i \geqq d_1$ (cf. Proposition 3), we see that throughout the algorithm, we have

$$p_i - p_i^0 \leqq d_t - d_i \leqq d_t - d_1 = D_t + p_t^0 - p_1^0 \quad \forall i \in \mathcal{I}. \tag{14}$$

Therefore, since prices increase by integer amounts, $D_t + p_t^0 - p_1^0 + 1$ bounds the number of times that p_i increases (with an attendant path contraction if $i \neq 1$). Since the computation per iteration is bounded by a constant multiple of the out-degree of the terminal node of the path, we see that the computation corresponding to contractions and price increases is $O(E(D_t + p_t^0 - p_1^0))$.

The number of path extensions with $i \in \mathcal{I}$ becoming the terminal node of the path is bounded by the number of increases of p_i, which in turn is bounded by $D_t + p_t^0 - p_1^0 + 1$. Thus the computation corresponding to extensions is also $O(E(D_t + p_t^0 - p_1^0))$. □

Note that we have $D_t \leqq hL$, where

$$L = \max_{(i,j) \in \mathcal{A}} a_{ij}, \tag{15}$$

$$h = \text{minimum number of arcs in a shortest path from 1 to } t. \tag{16}$$

Then in the special case where all arc lengths are nonnegative, and for the default price vector $p = 0$, Proposition 5 yields the running time estimate

$$O(EhL). \tag{17}$$

As the preceding estimate suggests, the running time can depend on L, as illustrated in Fig. 2 for a graph involving a cycle with relatively small length. This is the same type of graph for which the Bellman-Ford method starting with the zero initial conditions performs poorly (see [BeT89, p. 298]).

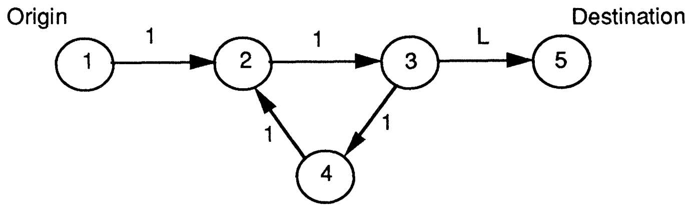


FIG. 2. *Example graph for which the number of iterations of the algorithm is not polynomially bounded. The lengths are shown next to the arcs and $L > 1$. By tracing the steps of the algorithm starting with $P = (1)$ and $p = 0$, we see that the price of node 3 will be first increased by 1 and then it will be increased by increments of 3 (the length of the cycle) as many times as necessary for p_3 to reach L.*

In the next section we will modify the algorithm to improve its complexity. However, we believe that the estimate of Proposition 5 is far from representative of the algorithm's "average" performance. For randomly generated problems, it appears that the number of iterations can be estimated quite reliably (within a constant factor roughly equal to two) by

$$n_t - 1 + \sum_{i \in \mathcal{I}, i \neq t} (2n_i - 1),$$

where n_i is the number of nodes in a shortest path from 1 to i; for example, for the problem of Fig. 1, the above estimate is exact.

2.3. The case of multiple destinations. We finally note that when there is a single origin and multiple destinations, the algorithm can be applied with virtually no change. We simply stop the algorithm when all destinations have become the terminal node of the path P at least once. If, initially, we choose $p_i = 0$ for all i, the destinations will be reached in the order of their proximity to the origin, as shown by Proposition 3. We also note that the algorithm can be similarly applied to a problem with multiple origins and a single destination, by first reversing the roles of origins and destinations, and the direction of each arc.

3. Arc length scaling. Throughout this section (and only this section) we will assume that all arc lengths are nonnegative. We introduce a version of the algorithm where the shortest path problem is solved several times, each time with different arc lengths and starting prices. Let

$$K = \lfloor \log L \rfloor + 1 \tag{18}$$

and for $k = 1, \cdots, K$, define

$$a_{ij}(k) = \left\lfloor \frac{a^{ij}}{2^{K-k}} \right\rfloor \quad \forall (i,j) \in \mathcal{A}. \tag{19}$$

Note that $a_{ij}(k)$ is the integer consisting of the k most significant bits in the K-bit binary representation of a_{ij}. Define

$$\bar{k} = \min \{k \geqq 1 \mid \text{each cycle has positive length}\}. \tag{20}$$

The following algorithm is predicated on the assumption that $\bar{k}$ is a small integer that does not grow beyond a certain bound as K increases. This is true for many problem types; for example, when the graph is acyclic, in which case $\bar{k} = 1$. For the case where this is not so, a slightly different arc length scaling procedure can be used; see the next section.

The scaled version of the algorithm solves $K - \bar{k} + 1$ shortest path problems, called *subproblems.* The arc lengths for subproblem k, $k = \bar{k}, \cdots, K$, are $a_{ij}(k)$ and the starting prices are obtained by doubling the final prices $p_i^*(k)$ of the previous subproblem

$$p_i^0(k+1) = 2p_i^*(k) \quad \forall i \in \mathcal{N}, \tag{21}$$

except for the first subproblem ($k = \bar{k}$), where we take

$$p_i^0(\bar{k}) = 0 \quad \forall i \in \mathcal{N}.$$

Note that we have $a_{ij}(K) = a_{ij}$ for all (i, j), and the last subproblem is equivalent to the original. Since the length of a cycle with respect to arc lengths $a_{ij}(\bar{k})$ is positive (by the definition of $\bar{k}$) and from the definition (19), we have

$$0 \leqq a_{ij}(k+1) - 2a_{ij}(k) \leqq 1 \quad \forall (i,j) \in \mathcal{A}, \tag{22}$$

it follows that cycles have positive length for each subproblem. Furthermore, in view of (22) and the doubling of the prices at the end of each subproblem (cf. (19)), the CS condition

$$p_i^0(k+1) \leqq p_j^0(k+1) + a_{ij}(k+1) \quad \forall (i,j) \in \mathcal{A} \tag{23}$$

is satisfied at the start of subproblem $k+1$, since it is satisfied by $p_i^*(k)$ at the end of subproblem k. Therefore, the algorithm of the preceding section can be used to solve all the subproblems.

Let $D_t(k)$ be the shortest distance from 1 to t for subproblem k and let

(24) $h(k) =$ the number of arcs in the final path from 1 to t in subproblem k.

It can be seen using (22) that

$$D_t(k+1) \leqq 2D_t(k) + h(k),$$

and in view of (21), we obtain

$$D_t(k+1) \leqq 2(p_1^*(k) - p_t^*(k)) + h(k) = p_1^0(k+1) - p_t^0(k+1) + h(k).$$

Using Proposition 5, it follows that the running time of the algorithm for subproblem k, $k = \bar{k}+1, \cdots, K$, is

$$O(E(k)h(k)), \tag{25}$$

where $E(k)$ is the number of the form (12) corresponding to subproblem k. The running time of the algorithm for subproblem $\bar{k}$ is

$$O(E(\bar{k})D_t(\bar{k})), \tag{26}$$

where $D_t(\bar{k})$ is the shortest distance from 1 to t corresponding to the lengths $a_{ij}(\bar{k})$. Since

$$a_{ij}(\bar{k}) < 2^{\bar{k}},$$

we have

$$D_t(\bar{k}) < 2^{\bar{k}} h(\bar{k}). \tag{27}$$

Adding over all $k = \bar{k}, \cdots, K$, we see that the running time of the scaled version of the algorithm is

$$O\left(2^{\bar{k}} E(\bar{k}) h(\bar{k}) + \sum_{k=\bar{k}+1}^{K} E(k)h(k)\right). \tag{28}$$

Assuming that $\bar{k}$ is bounded as L increases, the above expression is bounded by $O(NG\bar{h} \log L)$, where $\bar{h} = \max_{k=\bar{k},\cdots,K} h(k)$, N is the number of nodes, and G is the maximum out-degree of a node. These worst-case estimates of running time are still inferior to the sharpest estimate $O(A + N \log N)$ available for implementations of Dijkstra's method, where A is the number of arcs. The estimate (28) compares favorably with the estimate $O(Ah)$ for the Bellman-Ford algorithm when $2^{\bar{k}} \max_k E(k)$ is much smaller than A; this may occur if the destination is close to the origin relative to other nodes, in which case $\max_k E(k)$ may be much smaller than A.

We finally note that we can implement arc length scaling without knowing the value of $\bar{k}$. We can simply guess an initial value of $\bar{k}$, say $\bar{k} = 1$, apply the algorithm for lengths $a_{ij}(\bar{k})$, and at each path extension, check whether a cycle is formed. If so, we increment $\bar{k}$, we double the current prices, we reset the path to $P = (1)$, and we restart the algorithm with the new data and initial conditions. Eventually, after a finite number of restarts, we will obtain a value of $\bar{k}$ which is large enough for cycles never to form during the rest of the algorithm. The computation done up to that point, however, will not be entirely wasted; it will serve to provide a better set of initial prices.

4. Efficient implementation, two-sided algorithm, and preprocessing. The main computational bottleneck of the algorithm is the calculation of $\min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}$, which is done every time node i becomes the terminal node of the path. We can reduce the number of these calculations using the following observation. Since the CS condition (1a) is maintained at all times, if some arc (i, j_i) satisfies

$$p_i = a_{ij_i} + p_{j_i},$$

it follows that

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

so the path can be extended by j_i if i is the terminal node of the path. This suggests the following implementation strategy: each time a path contraction occurs with i being the terminal node, we calculate

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

together with an arc (i, j_i) such that

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

At the next time node i becomes the terminal node of the path, we check whether the condition $p_i = a_{ij_i} + p_{j_i}$ is satisfied, and if so, we extend the path by node j_i without going through the calculation of $\min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}$. In practice, this device is very effective, typically saving from a third to a half of the calculations of the preceding expression. The reason is that the test $p_i = a_{ij_i} + p_{j_i}$ is rarely failed; the only way it can fail is when the price p_{j_i} is increased between the two successive times i became the terminal node of the path.

The preceding idea can be strengthened further. Suppose that whenever we compute the "best neighbor"

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

we also compute the "second best neighbor" k_i given by

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

and the corresponding "second best level"

$$w_i = a_{ik_i} + p_{k_i}.$$

Then, at the next time node i becomes the terminal node of the path, we can check whether the condition $a_{ij_i} + p_{j_i} \leqq w_i$ is satisfied, and if so, we know that j_i still attains the minimum in the expression

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

thereby obviating the calculation of this minimum. If on the other hand we have $a_{ij_i} + p_{j_i} > w_i$ (due to an increase of p_{j_i} subsequent to the calculation of w_i), we can check to see whether we still have $w_i = a_{ik_i} + p_{k_i}$; if this is so, then k_i becomes the "best neighbor,"

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

thus obviating again the calculation of the minimum.

With proper implementation, the devices outlined above can typically reduce the number of calculations of the expression $\min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}$ by a factor in the order of three to five, thereby dramatically reducing the total computation time.

4.1. The two-sided algorithm. In shortest path problems, one can exchange the role of origins and destinations by reversing the direction of all arcs. It is therefore possible to use a destination-oriented version of our algorithm which maintains a path R that *ends* at the destination and changes at each iteration by means of a contraction or an extension. This algorithm, presented below and called the *reverse algorithm*, is equivalent to the algorithm in § 2, which will henceforth be referred to as the *forward algorithm*. The CS conditions for the problem with arc directions reversed are

$$\bar{p}_j \leqq a_{ij} + \bar{p}_i \quad \forall (i,j) \in \mathcal{A},$$

$$\bar{p}_j = a_{ij} + \bar{p}_i \quad \text{for all pairs of successive nodes } i \text{ and } j \text{ of } R,$$

where $\bar{p}$ is the price vector. By replacing $\bar{p}$ by $-p$, we obtain the CS conditions in the form of (1), thus maintaining a common CS condition for both the forward and the reverse algorithm. The following description of the reverse algorithm also replaces $\bar{p}$ by $-p$, with the result that the prices are *decreasing* instead of increasing. To be consistent with the assumptions made regarding the forward algorithm, we assume that each node except for the origin has at least one incoming arc.

In the reverse algorithm, initially, R is any path ending at the destination and p is any price vector satisfying the CS conditions (1) together with R; for example,

$$R=(t), \qquad p_i=0 \quad \forall i,$$

if all arc lengths are nonnegative.

TYPICAL ITERATION OF THE REVERSE ALGORITHM

Let j be the starting node of R. If

$$p_j > \max_{(i,j)\in\mathcal{A}} \{p_i - a_{ij}\},$$

go to Step 1; else go to Step 2.

Step 1: (Contract path). Set

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

and if $j \neq t$, contract R (that is, delete the starting node j of R). Go to the next iteration.

Step 2: (Extend path). Extend R by node i_j, (that is, make i_j the starting node of R, preceding j), where

$$i_j = \arg \max_{(i,j)\in\mathcal{A}} \{p_i - a_{ij}\}.$$

If i_j is the origin 1, stop; R is the desired shortest path. Otherwise, go to the next iteration.

The reverse algorithm is really the forward algorithm applied to a reverse shortest path problem, so by the results of § 2, it is valid and obtains a shortest path in a finite number of iterations, assuming that at least one path exists from 1 to t.

We now consider combining the forward and the reverse algorithms into one. In this combined algorithm, we initially have a price vector p and two paths P and R satisfying CS together with p, where P starts at the origin and R ends at the destination. The paths P and R are extended and contracted according to the rules of the forward and the reverse algorithms, respectively, and the combined algorithm terminates when P and R have a common node. Both P and R satisfy CS together with p throughout the algorithm, so when P and R meet, say at node i, the composite path consisting of the portion of P from 1 to i and the portion of R from i to t will be shortest.

COMBINED ALGORITHM

Step 1: (Run forward algorithm). Execute several iterations of the forward algorithm (subject to the termination condition), at least one of which leads to an increase of the origin price p_1. Go to Step 2.

Step 2: (Run reverse algorithm). Execute several iterations of the reverse algorithm (subject to the termination condition), at least one of which leads to a decrease of the destination price p_t. Go to Step 1.

To justify the combined algorithm, note that p_1 can only increase and p_t can only decrease during its course, while the difference $p_1 - p_t$ can be no more than the shortest distance between 1 and t. Assume that the arc lengths and the initial prices are integer, and that there is at least one path from 1 to t. Then, p_1 and p_t can only change by integer amounts and $p_1 - p_t$ is bounded. Hence, p_1 and p_t can change only a finite number of times, guaranteeing that there will be only a finite number of executions of Steps 1 and 2 of the combined algorithm. By the results of § 2, each Step 1 and Step 2 must contain only a finite number of iterations of the forward and the reverse algorithms, respectively. It follows that the algorithm must terminate in a finite number of iterations. Note that this argument relies on the requirement that p_1 increases at least once in Step 1 and p_t decreases at least once in Step 2. Without this requirement, one can construct examples showing that the combined algorithm may never terminate. Note also that our termination proof depends on the problem data being integer. For real problem data, we have been unable to prove termination or to disprove it with a counterexample.

One motivation for the combined algorithm is that two processors can be used in parallel to maintain the forward and the reverse paths while sharing the same price vector. However, there is another motivation. Based on our computational results, the combined algorithm is much faster than both the forward and the reverse algorithms.

4.2. Initialization and preprocessing. In order to initialize the algorithm, one should have a price vector p satisfying $p_i \leqq a_{ij} + p_j$ for all arcs (i, j). When some arc lengths are negative, the default choice $p = 0$ does not satisfy this condition, and there may be no obvious initial choice for p. In other situations, even when all arc lengths are nonnegative, it may be preferable to use a "favorable" initial price vector in place of the default choice $p = 0$. This possibility arises in a reoptimization context with slightly different arc length data, or with a different origin and/or destination. However, the "favorable" initial price vector may not satisfy the preceding condition.

To cope with situations such as the above, we provide a *preprocessing algorithm* for obtaining an appropriate initial vector p satisfying the condition $p_i \leqq a_{ij} + p_j$ for all arcs (i, j) (except for the immaterial outgoing arcs from the destination t).

To be precise, suppose that we have a vector $\bar{p}$, which, together with a set of arc lengths $\{\bar{a}_{ij}\}$, satisfies $\bar{p}_i \leqq \bar{a}_{ij} + \bar{p}_j$ for all arcs (i, j), and that we are given a new set of arc lengths $\{a_{ij}\}$. We describe a preprocessing algorithm for obtaining a vector p satisfying $p_i \leqq a_{ij} + p_j$ for all arcs (i, j). (Thus, to deal with the case where some arc lengths are negative and no appropriate initial vector is known, one can take $\bar{p} = 0$ and $\bar{a}_{ij} = \max\{0, a_{ij}\}$.) The algorithm maintains a subset of arcs $\mathscr{E}$ and a price vector p. Initially,

$$\mathscr{E} = \{(i, j) \in \mathscr{A} \mid a_{ij} < \bar{a}_{ij},\ i \neq t\}, \qquad p = \bar{p}.$$

The typical iteration is as follows:

Typical Preprocessing Iteration

Step 1: (Select arc to scan). If $\mathscr{E}$ is empty, stop; otherwise, remove an arc (i, j) from $\mathscr{E}$ and go to Step 2.

Step 2: (Add affected arcs to $\mathscr{E}$). If $p_i > a_{ij} + p_j$, set

$$p_i := a_{ij} + p_j$$

and add to $\mathscr{E}$ every arc (k, i) with $k \neq t$ that does not already belong to $\mathscr{E}$.

We have the following proposition.

Proposition 6. *Assume that each node i is connected to the destination t with at least one path. Then the preprocessing algorithm terminates in a finite number of iterations with a price vector p satisfying*

$$p_i \leqq a_{ij} + p_j \quad \forall (i, j) \in \mathscr{A} \quad \text{with } i \neq t. \tag{29}$$

Proof. We first note that by induction we can prove that throughout the algorithm we have

$$\mathscr{E} \supset \{(i, j) \in \mathscr{A} \mid p_i > a_{ij} + p_j,\ i \neq t\}.$$

As a result, when $\mathscr{E}$ becomes empty, the condition (29) is satisfied. Next, observe that by induction it can be seen that throughout the algorithm, p_i is equal to the modified length of some path starting at i (the length of the path plus the initial price of the final node of the path; see the proof of Proposition 4). Thus, termination of the algorithm will follow as in the proof of Proposition 4 (using the fact that cycle lengths are positive and prices are monotonically nonincreasing throughout the algorithm), provided we can show that the prices are bounded from below. Indeed, let

$$p_k^* = \begin{cases} \bar{p}_t + \text{shortest distance from } k \text{ to } t & \text{if } k \neq t, \\ \bar{p}_t & \text{if } k = t, \end{cases}$$

and let r be a sufficiently large scalar so that

$$\bar{p}_k \geqq p_k^* - r \quad \forall k.$$

We show by induction that throughout the algorithm we have

$$p_k \geqq p_k^* - r \quad \forall k \neq t. \tag{30}$$

Indeed, this condition holds initially by the choice of r. Suppose that the condition holds at the start of an iteration where arc (i, j) with $i \neq t$ is removed from $\mathscr{E}$. We then have

$$a_{ij} + p_j \geqq a_{ij} + p_j^* - r \geqq \min_{(i,m) \in \mathscr{A}} \{a_{im} + p_m^*\} - r = p_i^* - r,$$

where the last equality holds in view of the definition of p_k^* as a constant plus the shortest distance from k to t. Therefore, the iteration preserves the condition (30) and the prices p_i remain bounded throughout the preprocessing algorithm. This completes the proof. □

If the new arc lengths differ from the old ones by "small" amounts, it can be reasonably expected that the preprocessing algorithm will terminate quickly. This hypothesis, however, must be tested empirically on a problem-by-problem basis.

In the preceding preprocessing iteration, node prices can only decrease. An alternative iteration where node prices can only increase starts with

$$\mathscr{E} = \{(i, j) \in \mathscr{A} \mid a_{ij} < \bar{a}_{ij}, j \neq 1\}, \qquad p = \bar{p}.$$

and operates as follows:

Alternative Preprocessing Iteration

Step 1: (Select arc to scan). If $\mathscr{E}$ is empty, stop; otherwise, remove an arc (i, j) from $\mathscr{E}$ and go to Step 2.

Step 2: (Add affected arcs to $\mathscr{E}$). If $p_i > a_{ij} + p_j$, set

$$p_j := p_i - a_{ij}$$

and add to $\mathscr{E}$ every arc (j, k) with $k \neq 1$ that does not already belong to $\mathscr{E}$.

This algorithm is the preceding preprocessing algorithm (where prices decrease monotonically), but is applied to the reverse shortest path problem, where the arc directions have been reversed and the roles of origin and destination have been exchanged (cf. the two-sided algorithm given earlier). The following proposition therefore follows from Proposition 6.

Proposition 7. *Assume that the origin node* 1 *is connected to each node i with at least one path. Then the alternative preprocessing algorithm terminates in a finite number of iterations with a price vector p satisfying*

$$p_i \leqq a_{ij} + p_j \quad \forall (i, j) \in \mathscr{A} \quad \textit{with } j \neq 1.$$

The preprocessing idea can also be used in conjunction with arc length scaling in the case where the integer $\bar{k}$ of (20) is large or unknown. We can then use, in place of the scaled arc lengths $a_{ij}(k)$ of (19), the arc lengths

$$\tilde{a}_{ij}(k) = \left\lceil \frac{a_{ij}}{2^{K-k}} \right\rceil \quad \forall (i,j) \in \mathcal{A},$$

in which case we will have $\tilde{a}_{ij}(k) > 0$ if $a_{ij} > 0$. As a result, every cycle will have positive length with respect to arc lengths $\{\tilde{a}_{ij}(k)\}$ for all k. The difficulty now, however, is that (22) and (23) may not be satisfied. In particular, we will have instead

$$-1 \leqq \tilde{a}_{ij}(k+1) - 2\tilde{a}_{ij}(k) \leqq 0 \quad \forall (i,j) \in \mathcal{A},$$

and

$$p_i^0(k+1) \leqq p_j^0(k+1) + \tilde{a}_{ij}(k+1) + 1 \quad \forall (i,j) \in \mathcal{A}, \tag{31}$$

and the vector $p^0(k+1)$ may not satisfy the CS conditions with respect to arc lengths $\{\tilde{a}_{ij}(k+1)\}$. The small violation of the CS conditions indicated in (31) can be rectified by applying the preprocessing algorithm at the beginning of each subproblem. It is then possible to prove a polynomial complexity bound for the corresponding arc length scaling algorithm, by proving a polynomial complexity bound for the preprocessing algorithm and by using very similar arguments to those used in the previous section.

5. Parallelization issues.

When there is a single destination and multiple origins, several interesting parallel computation possibilities arise. The idea is to maintain a different path P^i for each origin i, and possibly, a reverse path R for the destination. Different paths may be handled by different processors, and price information can be shared by the processors in some way. There are several possible implementations of this idea. We will describe two of these implementations, motivated by the architectures of shared memory and message passing machines, respectively. For simplicity, we will not consider the possibility of using the reverse path R. In [Pol91], Polymenakos discusses parallel two-sided algorithms.

5.1. Shared memory implementation.

Here, there is a common price vector p stored in memory that is accessible by all processors. For each origin i, there is a path P^i satisfying CS together with p. In a synchronous implementation of the algorithm, an iteration is executed simultaneously for some origins (possibly all origins, depending on the availability of processors). At the end of an iteration, the results corresponding to the different origins are coordinated. To this end, we note that if a node is the terminal node of the path of several origins, the result of the iteration will be the same for all these origins, i.e., a path extension or a path contraction and corresponding price change will occur simultaneously for all these origins. The only potential conflict arises when a node i is the terminal path node for some origin and the path of a different origin is extended by i as a result of the iteration. Then, if p_i is increased due to a path contraction for the former origin, the path extension of the latter origin is cancelled. An additional important detail is that an origin i can stop its computation once the terminal node of its path P^i is an origin that has already found its shortest path to the destination. Thus, the processor handling this origin may be diverted to handle the path of another origin.

It is reasonable to speculate that the parallel time to solve the multiple origins problem is closer to the smallest time over all origins to find a single origin shortest path, rather than to the longest time. However, this conjecture needs to be tested experimentally on a shared memory machine.

The parallel implementation outlined above is synchronous, that is, all origins iterate simultaneously, and the results are communicated and coordinated at the end of the iteration to the extent necessary for the next iteration. An asynchronous implementation is also possible, principally because of the monotonicity of the mapping

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

see [Ber82] and [BeT89]. We refer to [Pol91] for a discussion of such an asynchronous implementation.

5.2. Message passing implementation. Here, for each origin i, there is a separate processor that executes the forward algorithm and keeps in local memory a price vector p^i and a corresponding path P^i satisfying CS together with p^i. The price vectors are communicated at various times to other processors, perhaps irregularly. A processor operating on (P^i, p^i), upon reception of a price vector p^j from another processor j, adopts as the price of each node n the *maximum* of the prices of n according to the existing and the received price vectors, that is,

$$p_n^i := \max\{p_n^i, p_n^j\} \quad \forall n \in \mathcal{N}. \tag{32}$$

The processor also uses the updated price vector p^i to delete successively, starting with the terminal node, the arcs (m, n) of P^i for which the equality $p_m = a_{mn} + p_n$ is violated. The CS property is maintained in this way because it can be shown that the updated price vector p^i satisfies the condition

$$p_m^i \leqq a_{mn} + p_n^i \quad \forall (m, n) \in \mathcal{A}.$$

This is the subject of the following proposition.

PROPOSITION 8. *Let p^i and p^j be two price vectors satisfying*

$$p_m^i \leqq a_{mn} + p_n^i, \qquad p_m^j \leqq a_{mn} + p_n^j \quad \forall (m, n) \in \mathcal{A}. \tag{33}$$

Then,

$$\max\{p_m^i, p_m^j\} \leqq a_{mn} + \max\{p_n^i, p_n^j\} \quad \forall (m, n) \in \mathcal{A}, \tag{34}$$

and

$$\min\{p_m^i, p_m^j\} \leqq a_{mn} + \min\{p_n^i, p_n^j\} \quad \forall (m, n) \in \mathcal{A}. \tag{35}$$

Proof. From (33), we have

$$p_m^i \leqq a_{mn} + \max\{p_n^i, p_n^j\} \quad \forall (m, n) \in \mathcal{A},$$

and

$$p_m^j \leqq a_{mn} + \max\{p_n^i, p_n^j\} \quad \forall (m, n) \in \mathcal{A}.$$

Combining these two relations, we obtain (34). The proof of (35) is similar. □

Note that even with no communication between the processors, the algorithm would still involve considerable parallelism, since a multiple origin problem would be solved in the time needed to solve a single origin problem. Combining the price vectors of several processors, however, tends to speed up the termination of the algorithm for all origins. In fact, if there are more processors than origins, it may still be beneficial to create some additional artificial origins in order to obtain additional price vectors. The drawback of this implementation is that communication of the price vectors may be relatively slow, and that combining two price vectors according to (32) may be time-consuming if no vector processing hardware is available at the processors.

6. Relation to naive auction and dual coordinate ascent. We now explain how our (forward) single origin-single destination algorithm can be viewed as an instance of the application of the naive auction algorithm to a special type of assignment problem.

The naive auction algorithm is applicable to assignment problems where we have to match n persons and n objects on a one-to-one basis. There is a cost c_{ij} for matching person i with object j and we want to assign persons to objects so as to minimize the total cost. There is also a restriction that person i can be assigned to object j only if (i, j) belongs to a set of given pairs $\mathcal{A}$. Mathematically, we want to find a feasible assignment that minimizes the total cost $\sum_{i=1}^{n} c_{ij_i}$, where by a feasible assignment we mean a set of person-object pairs $(1, j_1), \cdots, (n, j_n)$, such that the objects $j_1, \cdots, j_n$ are all distinct and $(i, j_i) \in \mathcal{A}$ for all i. (Auction algorithms are usually described in terms of maximization of the total "benefit" of the assignment; see, for example, [Ber90]. It is, however, convenient here to reformulate the problem and the algorithm in terms of minimization; this amounts to reversing the signs of the cost coefficients and the prices, and replacing maximization by minimization.)

The naive auction algorithm proceeds in iterations and generates a sequence of price vectors p and partial assignments (that is, assignments where only a subset of the persons have been matched with objects). At the beginning of each iteration, the condition

$$c_{ij_i} + p_{j_i} = \min_{(i,j) \in \mathcal{A}} \{c_{ij} + p_j\} \tag{36}$$

is satisfied for all pairs (i, j_i) of the partial assignment. The initial price vector–partial assignment pair is required to satisfy this condition, but is otherwise arbitrary. If all persons are assigned, the algorithm terminates. If not, some person who is unassigned, say i, is selected. This person finds an object j_i, which is best in the following sense:

$$j_i \in \arg \min_{(i,j) \in \mathcal{A}} \{c_{ij} + p_j\};$$

and then:

(a) Gets assigned to the best object j_i; the person that was assigned to j_i at the beginning of the iteration (if any) becomes unassigned.

(b) Sets the price of j_i to the level at which he/she is indifferent between j_i and the second best object, that is, he/she sets p_{j_i} to

$$p_{j_i} + w_i - v_i,$$

where v_i is the cost for acquiring the best object (including payment of the corresponding price),

$$v_i = \min_{(i,j) \in \mathcal{A}} \{c_{ij} + p_j\},$$

and w_i is the cost for acquiring the second best object

$$w_i = \min_{(i,j) \in \mathcal{A}, j \neq j_i} \{c_{ij} + p_j\}.$$

This process is repeated in a sequence of iterations until each person is assigned to an object.

The naive auction algorithm differs from the auction algorithm in the choice of the price increase increment. In the auction algorithm the price p_{j_i} is increased by $w_i - v_i + \varepsilon$, where ε is a small positive constant. Thus the naive auction algorithm is the same as the auction algorithm, except that $\varepsilon = 0$. This is, however, a significant difference; while the auction algorithm is guaranteed to terminate in a finite number of iterations if at least one feasible assignment exists, the naive auction algorithm may cycle indefinitely, with some objects remaining unassigned. If, however, the naive auction algorithm terminates, the feasible assignment obtained upon termination is optimal. The reason is that (36) may be viewed as a complementary slackness condition for the linear programming problem associated with the assignment problem, and by a classical linear programming result, this condition, together with feasibility, guarantees optimality of the final assignment.

6.1. Formulation of the shortest path problem as an assignment problem. Now, given the shortest path problem described in § 2, with node 1 as origin and node t as destination, we formulate the following assignment problem.

Let $2, \cdots, N$ be the "object" nodes, and for each node $i \neq t$, introduce a "person" node i'. For every arc (i, j) of the shortest path problem with $i \neq t$ and $j \neq 1$, introduce the arc (i', j) with cost a_{ij} in the assignment problem. Also introduce the zero cost arc (i', i) for each $i \neq 1, t$. Figure 3 illustrates the assignment problem.

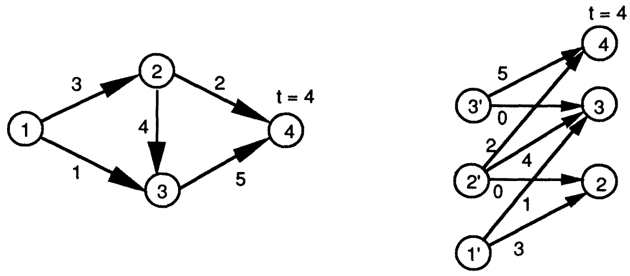


FIG. 3. *A shortest path problem and its corresponding assignment problem. The arc lengths and the assignment costs are shown next to the arcs.*

Now consider applying the naive auction algorithm starting from a price vector p satisfying the CS condition (1a), i.e.,

$$p_i \leqq a_{ij} + p_j \quad \forall (i, j) \in \mathcal{A}, \tag{37}$$

and the partial assignment

$$(i', i) \quad \forall i \neq 1, t.$$

This initial pair satisfies the corresponding condition (36), because the cost of the assigned arcs (i', i) is zero.

We impose an additional rule for breaking ties in the naive auction algorithm: if at some iteration involving the unassigned person i', the arc (i', i) is the best arc and is equally desirable with some other arc (i', j_i) (i.e., $p_i = a_{ij_i} + p_{j_i} = \min_{(i,j)\in\mathcal{A}} \{a_{ij} + p_j\}$), then the latter arc is preferred; that is, (i', j_i) is added to the assignment rather than (i', i). Furthermore, we introduce an inconsequential modification of the naive auction iteration involving a bid of person $1'$, in order to account for the special way of handling a contraction at the origin in the shortest path algorithm. In particular, the bid of $1'$ will consist of finding an object j_1 attaining the minimum in

$$\min_{(1,j)\in\mathcal{A}} \{a_{1j} + p_j\},$$

assigning j_1 to $1'$, and deassigning the person assigned to j_1 (in the case $j_1 \neq t$), but *not* changing the price p_{j_1}.

It can now be shown that the naive auction algorithm under the preceding conditions is equivalent to the (forward) shortest path algorithm of § 2. In particular, the following can be verified by induction:

(a) The CS condition (37) is preserved by the naive auction algorithm.

(b) Each assignment generated by the algorithm consists of a sequence of the form

$$(1', i_1), (i'_1, i_2), \cdots, (i'_{k-1}, i_k), \tag{38}$$

together with the additional arcs

$$(i', i) \quad \text{for } i \neq i_1, \cdots, i_k, t,$$

and corresponds to a path $P = (1, i_1, \cdots, i_k)$ generated by the shortest path algorithm. As long as $i_k \neq t$, the (unique) unassigned person in the naive auction algorithm is person i'_k, corresponding to the terminal node of the path. When $i_k = t$, a feasible assignment results, in which case the naive auction algorithm terminates, consistently with the termination criterion for the shortest path algorithm.

(c) In an iteration corresponding to an unassigned person i' with $i \neq 1$, the arc (i', i) is always a best arc; this is a consequence of the complementary slackness condition (37). Furthermore, there are three possibilities: (1) (i', i) is the unique best arc, in which case (i', i) is added to the assignment, and the price p_i is increased by

$$\min_{(i,j)\in\mathcal{A}} \{c_{ij} + p_j\} - p_i;$$

this corresponds to contracting the current path by the terminal node i. (2) There is an arc (i', j_i) with $j_i \neq t$, which is equally preferred to (i', i), that is,

$$p_i = a_{ij_i} + p_{j_i},$$

in which case, in view of the tie-breaking rule specified earlier, (i', j_i) is added to the assignment and the price p_{j_i} remains the same. Furthermore, the object j_i must have been assigned to j'_i at the start of the iteration, so adding (i', j_i) to the assignment (and removing (j'_i, j_i)) corresponds to extending the current path by node j_i. (The positivity assumption on the cycle lengths is crucial for this property to hold.) (3) The arc (i', t) is equally preferred to (i', i), in which case the heretofore unassigned object t is assigned to i', thereby terminating the naive auction algorithm; this corresponds to the destination t becoming the terminal node of the current path, thereby terminating the shortest path algorithm.

We have thus seen that the shortest path algorithm may be viewed as an instance of the naive auction algorithm. However, the properties of the former algorithm do not follow from generic properties of the latter. As mentioned earlier, the naive auction algorithm need not terminate, in general. In the present context, it does terminate thanks to the special structure of the corresponding assignment problem, and also thanks to the positivity assumption on all cycle lengths.

6.2. Relation to dual coordinate ascent. We next explain how the single origin-single destination algorithm can be viewed as a dual coordinate ascent method. The shortest path problem can be written in the minimum cost flow format

$$\text{(LNF)} \qquad \text{minimize} \sum_{(i,j)\in\mathcal{A}} a_{ij}x_{ij}$$

$$\text{subject to} \sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i \quad \forall i \in \mathcal{N}, \tag{39}$$

$$0 \leqq x_{ij} \quad \forall (i,j) \in \mathcal{A}, \tag{40}$$

where

$$s_1 = 1, \qquad s_t = -1,$$
$$s_i = 0 \quad \forall i \neq 1, t,$$

and t is the given destination.

The standard linear programming dual problem is

$$\text{(41)} \qquad \begin{aligned} &\text{maximize} \quad p_1 - p_t \\ &\text{subject to} \quad p_i - p_j \leqq a_{ij} \quad \forall (i,j) \in \mathcal{A}, \end{aligned}$$

and by a classical duality theorem [Chv83], [Dan63], [PaS82], [Roc84], the optimal primal cost is equal to the optimal dual cost.

Let us associate with a given path $P = (1, i_1, i_2, \cdots, i_k)$ the flow

$$x_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are successive nodes in } P, \\ 0 & \text{otherwise.} \end{cases}$$

Then, the CS conditions (1a) and (1b) are equivalent to the usual linear programming complementary slackness conditions

$$p_i \leqq a_{ij} + p_j \quad \forall (i,j) \in \mathcal{A},$$
$$0 < x_{ij} \Rightarrow p_i = a_{ij} + p_j \quad \forall (i,j) \in \mathcal{A}.$$

For a pair (x, p), the above conditions, together with primal feasibility (the conservation of flow constraint (39) for all $i \in \mathcal{N}$, which in our case translates to the terminal node of the path P being the destination node) are the necessary and sufficient conditions for x to be primal-optimal and p to be dual-optimal. Thus, upon termination of our shortest path algorithm, the price vector p is an optimal-dual solution.

To interpret the algorithm as a dual ascent method, note that a path contraction and an attendant price increase of the terminal node i of P, corresponds to a step along the price coordinate p_i that leaves the dual cost $p_1 - p_t$ unchanged if $i \neq 1$. Furthermore, an increase of the origin price p_1 by an increment δ improves the dual cost by δ. Thus the algorithm may be viewed as a finitely terminating dual coordinate ascent algorithm, except that true ascent steps occur only when the origin price increases; all other ascent steps are "degenerate," producing a price increase but no change in dual cost.

7. Computational results.

The combined (forward and reverse) version of the algorithm without arc length scaling was implemented in a code called AUCTION_SP. This code solves the problem with a single origin and a selected set of destinations. It operates in cycles of iterations, alternating between the origin and one of the destinations. In particular, the algorithm first performs a group of (forward) iterations starting with the origin and proceeding up to the point where the origin again becomes the terminal node of the forward path; then the algorithm performs a group of (reverse) iterations starting at some destination, call it t, and proceeding up to the point where t becomes again the terminal node of the reverse path. The process is then repeated, starting again at the origin and then starting at another destination, and so on. The destinations are taken up cyclically, except that once the reverse path of some destination meets the forward path (in which case a shortest path for the given destination has been found), this destination is not iterated upon any further. Naturally, the same price vector p is used for the forward and all the reverse paths. The algorithm uses the default initialization ($p = 0$, $P = (1)$, $R = (t)$, for all destinations t), and terminates when each of the reverse paths have met the forward path.

We compared our code with the shortest path code SHEAP, due to Gallo and Pallotino [GaP88]. This is an implementation of Dijkstra's method that uses a binary heap to store the nodes which are not yet permanently labeled. We made a simple modification to this code so that it terminates when all the destinations (rather than all the nodes) become permanently labeled. Our informal comparison with other shortest path codes agrees with the conclusion of [GaP88] that SHEAP is a very efficient state-of-the-art code for a broad variety of types of shortest path problems. While other shortest path codes may produce faster solution times than SHEAP, we believe that the differences are not sufficiently large to invalidate the qualitative nature of our comparisons. We did not test our code against label correcting methods such as the threshold algorithm [GKP85], [GaP88], since these methods are at a disadvantage in the case of only a few origin-destination pairs.

We restricted our experiments to randomly generated shortest path problems obtained using the widely available NETGEN program [KNS74]. Problems were generated by specifying the number of nodes N, the number of arcs A, the length range $[1, L]$, and a single source and sink (automatically chosen by NETGEN to be nodes 1 and N). The times required by the two codes on a Macintosh II are shown in Tables 1 and 2, for the cases of one destination and four destinations, respectively. The tables show that AUCTION_SP is much faster than SHEAP on NETGEN problems; this was confirmed by extensive additional testing.

For the case of a single destination, we have also experimented with a version of SHEAP, called TWO_TREE_SHEAP, that builds a shortest path tree from the origin and another shortest path tree from the destination. Recent computational research [HKS88], [HKS89] has confirmed that using two trees in Dijkstra's method, as originally suggested in [Nic66], typically accelerates convergence, and our experience agrees with this conclusion. Still, however, AUCTION_SP was substantially faster than TWO_TREE_SHEAP, as shown in Table 1.

For multiple destination problems, we know of no Dijkstra-like algorithm that uses multiple trees; one has to run a two-sided algorithm separately for each origin-destination pair. Thus, in contrast with our algorithm, the advantage of a two-sided Dijkstra algorithm is dissipated quickly as the number of destinations increases from one. Therefore, based on our computational experience, we conclude that AUCTION_SP is by far the fastest code for random problems of the type generated by NETGEN and for few destinations (more than one, but much less than the maximum possible).

We note that for "one-to-all" problems, where there is a single origin and all other nodes are destinations, AUCTION_SP has been running slower than the best label correcting methods, including SHEAP. However, the differences in performance were not overwhelming (a factor of the order of two to three), and it will be interesting to make the corresponding comparison in a parallel computing environment.

The reader is warned that the computational results of the table are far from conclusive. Clearly, one can find problems where AUCTION_SP is vastly inferior to SHEAP in view of its inferior computational complexity, cf. Fig. 2 (although such a problem was never encountered in our experiments with randomly generated problems). An important issue is to delineate, through average complexity analysis and computational experimentation, the types of practical problems for which our algorithm is substantially better than the best label setting and label correcting methods. We find our computational results very encouraging, but further research and testing with both serial and parallel machines must be done before we can reach solid conclusions on the merits of our algorithm. We also note that the ideas in this paper are new and

TABLE 1
Solution times in secs of shortest path codes on a Mac II using problems generated by NETGEN *with one destination (node N). The lengths of all arcs were randomly generated from the range* [1, 1000].

N	A	AUCTION_SP	SHEAP	TWO_TREE_SHEAP
1,000	4,000	0.033	0.250	0.033
1,000	10,000	0.050	0.200	0.133
2,000	8,000	0.017	0.017	0.017
2,000	20,000	0.067	0.867	0.150
3,000	12,000	0.067	0.983	0.100
3,000	30,000	0.033	1.117	0.100
4,000	16,000	0.067	1.233	0.100
4,000	40,000	0.033	0.383	0.100
5,000	20,000	0.050	1.383	0.083
5,000	50,000	0.033	0.550	0.100

TABLE 2
Solution times in secs of shortest path codes on a Mac II using problems generated by NETGEN *with four destinations (nodes N, $N-100$, $N-200$, $N-300$). The lengths of all arcs were randomly generated from the range* [1, 1000].

N	A	AUCTION_SP	SHEAP
1,000	4,000	0.050	0.250
1,000	10,000	0.080	0.383
2,000	8,000	0.100	0.667
2,000	20,000	0.233	0.883
3,000	12,000	0.117	1.100
3,000	30,000	0.167	1.117
4,000	16,000	0.100	1.233
4,000	40,000	0.117	1.883
5,000	20,000	0.150	1.533
5,000	50,000	0.183	1.833

their potential is not yet fully developed. It is likely that as these ideas are better understood, more efficient codes will become available.

REFERENCES

[AMO89] R. K. AHUJA, T. L. MAGNANTI, AND J. B. ORLIN, *Network flows*, Sloan Working Paper No. 2059-88, Sloan School of Management, Cambridge, MA, March 1989; also in Handbooks in Operations Research and Management Science, Vol. 1, Optimization, G. L. Nemhauser, A. H. G. Rinnooy-Kan, and M. J. Todd, eds., North-Holland, Amsterdam, 1989.

[Ber79] D. P. BERTSEKAS, *A distributed algorithm for the assignment problem*, Laboratory for Information and Decision Systems Working Paper, Massachusetts Institute of Technology, Cambridge, MA, March 1979.

[Ber81] ———, *A new algorithm for the assignment problem*, Math. Programming, 21 (1981), pp. 152-171.

[Ber82] ———, *Distributed dynamic programming*, IEEE Trans. Automat. Control, 27(1982), pp. 610-616.

[Ber86] ———, *Distributed relaxation methods for linear network flow problems*, in Proc. 25th IEEE Conference on Decision and Control, 1986, pp. 2101-2106.

[Ber88] ———, *The auction algorithm: A distributed relaxation method for the assignment problem*, Ann. Oper. Res. 14 (1988), pp. 105-123.

[Ber90] ———, *The auction algorithm for assignment and other network flow problems: A tutorial*, Interfaces, 20 (1990), pp. 133–149.

[BeE88] D. P. Bertsekas and J. Eckstein, *Dual coordinate step methods for linear network flow problems*, Math. Programming Ser. B, 42 (1988), pp. 203–243.

[BeT89] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[Chv83] V. Chvatal, *Linear Programming*, W. H. Freeman, New York, 1983.

[Dan63] G. B. Dantzig, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[DGK79] R. Dial, F. Glover, D. Karney, and D. Klingman, *A computational analysis of alternative algorithms and labeling techniques for finding shortest path trees*, Networks, 9 (1979), pp. 215–248.

[Dia69] R. B. Dial, *Algorithm* 360: *Shortest path forest with topological ordering*, Comm. ACM, 12 (1969), pp. 632–633.

[GaP86] G. Gallo and S. Pallotino, *Shortest path methods: A unified approach*, Math. Programming Stud., 26 (1986), pp. 38–64.

[GaP88] G. Gallo and S. Pallotino, *Shortest path algorithms*, Ann. Oper. Res., 7 (1988), pp. 3–79.

[GKP85] F. Glover, D. Klingman, N. Phillips, and R. F. Schneider, *New polynomial shortest path algorithms and their computational attributes*, Management Science, 31 (1985), pp. 1106–1128.

[HKS88] R. V. Helgason, J. L. Kennington, and B. D. Stewart, *Dijkstra's two-tree shortest path algorithm*, Tech. Report 89-CSE-32, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX, 1988.

[HKS89] R. V. Helgason, J. L. Kennington, and B. D. Stewart, *Computational comparison of sequential and parallel algorithms for the one-to-one shortest-path problem*, Tech. Report 89-CSE-32, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX, 1989.

[JoV87] R. Jonker and A. Volegnant, *A shortest augmenting path algorithm for dense and sparse linear assignment problems*, Computing, 38 (1987), pp. 325–340.

[KNS74] 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, 20 (1974), pp. 814–822.

[Ker81] A. Kershenbaum, *A note on finding shortest path trees*, Networks, 11 (1981), pp. 399–400.

[Law76] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, and Winston, New York, 1976.

[Nic66] T. Nicholson, *Finding the shortest route between two points in a network*, Comput. J., 9 (1966), pp. 275–280.

[Pap74] U. Pape, *Implementation and efficiency of Moore-algorithms for the shortest path problem*, Math. Programming, 7 (1974), pp. 212–222.

[PaS82] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Pol91] L. Polymenakos, *Analysis of parallel asynchronous schemes for the auction shortest path algorithm*, Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1991.

[Roc84] R. T. Rockafellar, *Network flows and monotropic programming*, Wiley-Interscience, New York, 1984.

[ShW81] D. R. Shier and C. Witzgall, *Properties of labeling methods for determining shortest path trees*, J. Res. Nat. Bur. Standards, 86 (1981), p. 317.