May 1990 LIDS - P - 1973

Forward Path Search: A New Dual Coordinate Ascent Algorithm for Shortest Paths[1]

by

Dimitri P. Bertsekas[2]

Abstract

We propose a new and simple algorithm for finding a shortest path from a single origin to one or more destinations. 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. The algorithm can be extended for the case of multiple origins, and for this case it is well suited for parallel computation.

[1] Research supported by NSF under Grant No. DDM-8903385 and by the ARO under Grant DAAL03-86-K-0171.

[2] Laboratory for Information and Decision Systems, M.I.T, Cambridge, Mass. 02139.

1. INTRODUCTION

In this paper we consider the problem of finding a shortest path from a single origin to one or more destinations in a directed graph $(\mathcal{N}, \mathcal{A})$. We assume that all arc lengths a_{ij} are nonnegative and all cycles have positive length. Furthermore, to simplify the presentation, we assume that each node has at least one outgoing incident arc; any node not satisfying this condition can be connected to some destination node with a very high length arc without changing materially 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.

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 all the given destinations become the terminal node of the path at least once, the algorithm terminates.

In the process of finding the shortest path to the given destinations, the algorithm discovers a shortest path from the origin to several other nodes; in fact, for the most common initialization of the algorithm, these paths are discovered in order of increasing length order. In this sense, the algorithm resembles Dijkstra's method. However, Dijkstra's method maintains a shortest path tree to all "permanently labeled" nodes, while our method maintains only one path.

In its pure form, the algorithm is pseudopolynomial; its running time 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.

The practical performance of the algorithm remains to be fully investigated. Preliminary experimental results and comparison with the state of the art shortest path codes of Gallo and Pallotino [GaP88] have been encouraging; see Section 7.

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. A similar enhancement does not

appear possible for Dijkstra's method.

The algorithm is also well suited for graphs with few origins and destinations, and with a large number of nodes for which the Bellman-Ford algorithm is unsuitable. For such graphs the algorithm may compete well with heuristic search methods of the type that are popular for artificial intelligence applications (see [Pea84] and [Ber87]), while being more parallelizable than Dijkstra's method.

To place our algorithm in perspective, we note that shortest path methods are traditionally divided in 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. However, our method is different in one apparently fundamental way: prices are monotonically increasing in our method but they are monotonically decreasing in label setting and label correcting methods. This in turn results in a different type of worst case behavior for our method and methods such as the Bellman-Ford algorithm with infinite initial labels; see Fig. 2 and the related discussion.

As we explain in Section 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 mehods can be found in [BeE88] and [BeT89]). If one applies the ϵ-relaxation method for a minimum cost flow formulation of the shortest path problem (see Section 6), but with the important difference that $\epsilon = 0$, then one obtains an algorithm which is very similar to the one provided here.

The paper is organized as follows: In Section 2, we describe the basic algorithm for the single destination case and we prove its basic properties. In Section 3, we develop the polynomial version of the algorithm using arc length scaling. In Section 4, we describe the use of a data structure that accelerates substantially the algorithm. In Section 5, we consider the multiple destination and multiple origin case and we briefly discuss how the algorithm can take advantage of a parallel computing environment. The implementation and performance of parallel versions of the algorithm will be discussed more fully in a separate paper. In Section 6, we derive the connection with duality and we show that the algorithm may be viewed as a coordinate ascent (or Gauss-Seidel relaxation) method for maximizing a certain dual cost function. Finally, Section 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.

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

The algorithm maintains at all times a 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 $(1, i_1, i_2, \ldots, i_k, i_{k+1})$, 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 by the path $(1, i_1, i_2, \ldots, i_{k-1})$.

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

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

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

We denote by p the vector of prices p_i. A path-vector pair (P,p) satisfying the above conditions is said to satisfy *complementary slackness* (or CS for short). This terminology is motivated by a formulation of the shortest path problem as a linear minimum cost flow problem; see Section 6. In this formulation, 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, 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 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 now describe the algorithm. Initially, (P,p) is any pair satisfying CS such as, for example,

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

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 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. Terminate the iteration.

Step 2: (Extend path) Extend P by node i_x where

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

If i_x is the destination t, stop; P is the desired shortest path. Otherwise, terminate the iteration.

It can be seen that following the extension Step 2, P is a path from 1 to i_x. Indeed if this were not so, then adding i_x 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.

Proposition 1: The pairs (P,p) generated by the algorithm satisfy CS. Furthermore, for every pair of nodes i and 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 since $a_{ij} \geq 0$ for all $(i,j) \in \mathcal{A}$. 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

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

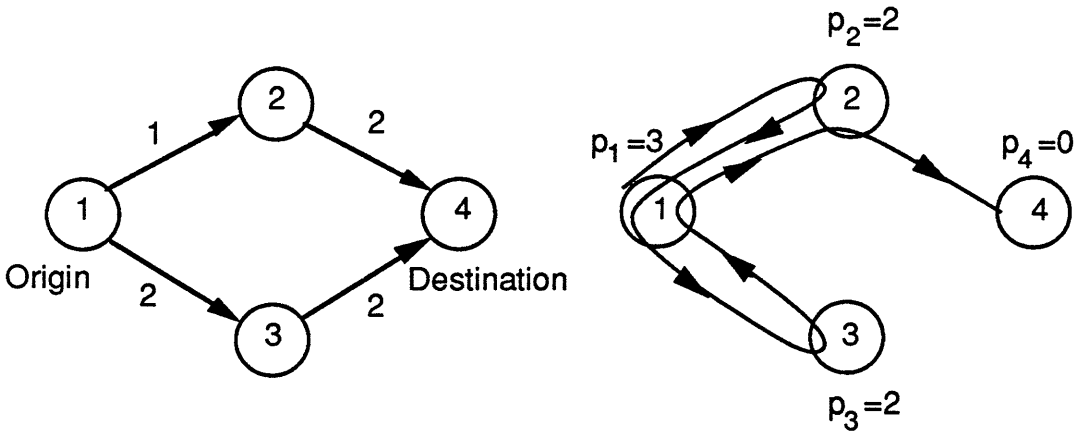

Figure 1: Trajectory of the terminal node and final prices generated by the algorithm starting with $P = (1)$ and $p = 0$.

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

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

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 $\bar{p}_k \leq a_{kj} + \bar{p}_j$ for all arcs (k, j) with $k \notin P$. This completes the induction proof.

Finally, if (P, p) satisfies the CS condition (1a), we see that the length of any path starting at node i and ending at node j is at least $p_i - p_j$, proving the last assertion of the proposition. **Q.E.D.**

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 Prop. 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 $p_1 - p_i$. **Q.E.D.**

Denote for each node i,

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

with $D_1 = 0$ by convention, and denote also

$$p_i^0 = \text{initial price of node } i, \tag{7}$$

$$d_i = D_i + p_i^0. \tag{8}$$

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

$$k_i = \text{the first iteration index at which node } i \text{ becomes a terminal node.} \tag{9}$$

By convention, $k_1 = 0$ and $k_i = \infty$ if the node 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 $d_i \leq d_j$.

Proof: (a) At the end of iteration k_i, P is a shortest path from 1 to i by Prop. 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. **Q.E.D.**

Note that the preceding proposition shows that for the default initialization case where $p_i = 0$ for all i, the nodes become terminal for the first time in the order of their proximity to the origin.

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. Since by Prop. 1, $p_1 - p_t$ is an underestimate of the (finite) shortest distance from 1 to the destination 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 Prop. 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.

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 walk starting at i plus the initial price of the final node of the walk; we call this the *modified length* of the walk. Each time i becomes the terminal node by extension of the path P, p_i is strictly larger over the preceding time i became the terminal node of P, corresponding to a strictly larger modified walk length. Since the number of modified walk lengths within any bounded interval is bounded and p_i stays bounded, 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.

If there is no path from node 1 to the destination, 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 $p_1 \to \infty$. **Q.E.D.**

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

$$I = \{i \mid d_i \leq d_t\}, \tag{10}$$

and the set of arcs

$$\mathcal{R} = \{(i,j) \in \mathcal{A} \mid i \in I\}. \tag{11}$$

We denote by R the number of arcs in $\mathcal{R}$.

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 running time of the algorithm is $O\left(R(D_t + p_t^0 - p_1^0)\right)$.

Proof: Each time a node i becomes the terminal node of the path, we have $p_i = p_1 - D_i$ (cf. Prop. 2). Since at all times we have $p_1 \leq D_t + p_t^0$ (cf. Prop. 3), it follows that

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

and using the definitions $d_t = D_t + p_t^0$ and $d_i = D_i + p_i^0$, we see that

$$p_i - p_i^0 \leq d_t - d_i$$

at the end of all iterations for which node i becomes the terminal node of the path. Therefore, since prices increase by integer amounts, $d_t - d_i + 1$ bounds the number of times that p_i increases (with an attendant path contraction if $i \neq 1$). The number of path extensions with i being the terminal node of the path is bounded by the corresponding number of price increases. Furthermore, for each iteration where i is the terminal node of the path, the computation is proportional to the number of outgoing incident arcs of i, call it n_i. We conclude that the computation time at iterations where i is the terminal node of the path is bounded by

$$M(d_t - d_i + 1)n_i, \tag{12}$$

where M is a constant which is independent of the problem data. Adding over the set I of all nodes with $d_i \leq d_t$, we can bound the running time of the algorithm by

$$M\sum_{i\in I}(d_t - d_i + 1)n_i. \tag{13}$$

Since by Prop. 3, we have $d_i \geq d_1$, the upper bound (13) is less or equal to

$$M(d_t - d_1 + 1)\sum_{i\in I} n_i = M(d_t - d_1 + 1)R = M(D_t + p_t^0 - p_1^0)R,$$

and the result follows. **Q.E.D.**

Let us denote

$$L = \max_{(i,j)\in\mathcal{A}} a_{ij}, \tag{14}$$

$$h = \text{minimum number of arcs in a shortest path from 1 to } t. \tag{15}$$

Then we have $D_t \leq hL$, and for the default price vector $p = 0$, Prop. 5 yields the running time estimate

$$O(RhL). \tag{16}$$

As the preceding estimate suggests, the running time can depend on L. This is 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). In the next section we modify the algorithm to improve its complexity.

3. ARC LENGTH SCALING

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

$$K = \lfloor \log L \rfloor + 1 \tag{17}$$

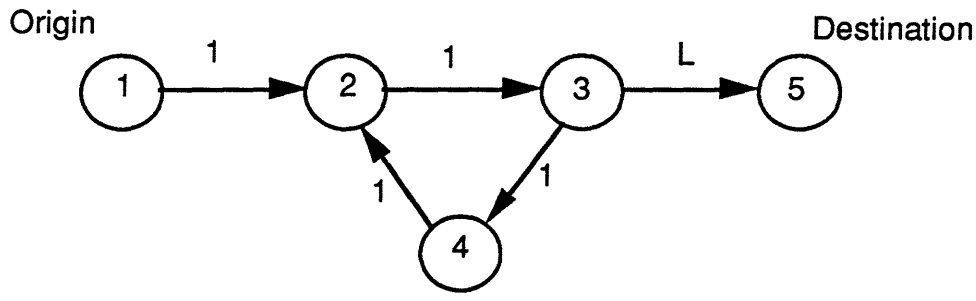

Figure 2: Example graph for which the number of iterations of the algorithm is not polynomially bounded. The lengths are shown next to the arcs and $L > 1$. By tracing the steps of the algorithm starting with $P = (1)$ and $p = 0$, we see that the price of node 3 will be first increased by 1 and then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

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

$$a_{ij}(k) = \left\lfloor \frac{a_{ij}}{2^{K-k}} \right\rfloor \qquad \forall\ (i,j) \in \mathcal{A}. \tag{18}$$

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

$$\bar{k} = \min\{k \geq 1 \mid \text{each cycle has at least one arc } (i,j) \text{ with } a_{ij}(k) > 0\}. \tag{19}$$

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, $k = \bar{k}, \ldots, K$, are $a_{ij}(k)$ and the starting prices are obtained by doubling the final prices $p_i^*(k)$ of the previous subproblem

$$p_i^0(k+1) = 2p_i^*(k), \qquad \forall\ i \in \mathcal{N}, \tag{20}$$

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

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

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 (18), we have

$$0 \leq a_{ij}(k+1) - 2a_{ij}(k) \leq 1, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{21}$$

it follows that cycles have positive length for each subproblem. Furthermore, in view of Eq. (21), and the doubling of the prices at the end of each subproblem (cf. Eq. (20)), the CS condition

$$p_i^0(k+1) \leq p_j^0(k+1) + a_{ij}(k+1), \qquad \forall\ (i,j) \in \mathcal{A} \tag{22}$$

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

$$h(k) = \text{the number of arcs in the final path from 1 to } t \text{ in subproblem } k. \tag{23}$$

It can be seen using Eq. (21) that

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

and in view of Eq. (20), we obtain

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

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

$$O\big(R(k)h(k)\big), \tag{24}$$

where $R(k)$ is the number of arcs in the set of the form (11) corresponding to subproblem k. The running time of the algorithm for subproblem $\bar{k}$ is

$$O\big(R(\bar{k})D_t(\bar{k})\big), \tag{25}$$

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

$$D_t(\bar{k}) < 2^{\bar{k}} h(\bar{k}). \tag{26}$$

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

$$O\left(2^{\bar{k}} R(\bar{k})h(\bar{k}) + \sum_{k=\bar{k}+1}^{K} R(k)h(k)\right). \tag{27}$$

Assuming that $\bar{k}$ is bounded as L increases, the above expression is bounded by $O(A\bar{h}\log L)$, where $\bar{h} = \max_{k=\bar{k},\ldots,K} h(k)$, and A is the number of arcs. 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 N is the number of nodes. The estimate (27) compares favorably with the estimate $O(Ah)$ for the Bellman-Ford algorithm when $2^{\bar{k}} \max_k R(k)$ is much smaller than A; this may occur if the destination is close to the origin relative to other nodes in which case $\max_k R(k)$ 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 – 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 arc"

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

we also compute the "second best arc" $\bar{j}_i$ given by

$$\bar{j}_i = \arg \min_{(i,j)\in calA,\ j\neq j_i} \{a_{ij} + p_j\},$$

and the corresponding "second best level"

$$w_i = a_{i\bar{j}_i} + p_{\bar{j}_i}.$$

Then, at the next time node i becomes the terminal node of the path, we can check whether the condition $a_{ij_i} + p_{j_i} \leq w_i$ 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 $w_i = a_{i\bar{j}_i} + p_{\bar{j}_i}$; if this is so, then $\bar{j}_i$ becomes the "best arc",

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

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.

Preprocessing

Another important way to reduce the computation time of the algorithm is 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. A related situation arises in a parallel computing context; see the next section.

One difficulty here is that the "favorable" initial condition p may not satisfy the CS conditions (1), that is, for some node i we may have

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

In this case, we can obtain a vector p satisfying the CS conditions (except on the immaterial outgoing arcs from the destination t) by a *preprocessing algorithm*, which is reminiscent of some implementations of label correcting methods.

To be precise, suppose that we have a vector $\bar{p}$ which satisfies the CS conditions together with a set of arc lengths $\{\bar{a}_{ij}\}$, and we are given a new set of arc lengths $\{a_{ij}\}$. We describe a preprocessing algorithm that maintains a subset of arcs $\mathcal{E}$ and a price vector p. Initially

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

The typical iteration is as follows:

Typical Preprocessing Iteration

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

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

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

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

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

$$p_i \leq a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } i \neq t. \tag{28}$$

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

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

As a result, when $\mathcal{E}$ becomes empty, the condition (28) 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 walk starting at i (see the proof of Prop. 4). Thus, termination of the algorithm will follow as in the proof of Prop. 4 (using the fact that walk lengths are nonnegative 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

$$\bar{p}_k \geq p_k^* - r, \qquad \forall\ k.$$

We show by induction that throughout the algorithm we have

$$p_k \geq p_k^* - r, \qquad \forall\ k \neq t. \tag{29}$$

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 $\mathcal{E}$. We then have

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

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 (29) and the prices p_i remain bounded throughout the preprocessing algorithm. This completes the proof. **Q.E.D.**

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

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

and operates as follows:

Alternative Preprocessing Iteration

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

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

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

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

The following proposition admits a similar proof as Prop. 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

$$p_i \leq a_{ij} + p_j, \qquad \forall\ (i,j) \in \mathcal{A} \text{ with } j \neq 1.$$

The preprocessing idea can also be used in conjunction with arc length scaling in the case where the integer $\bar{k}$ of Eq. (19) is large or unknown. We can then use in place of the scaled arc lengths $a_{ij}(k)$ of Eq. (18), the arc lengths

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

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, however, now is that Eqs. (21) and (22) may not be satisfied. In particular, we will have instead

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

and

$$p_i^0(k+1) \leq p_j^0(k+1) + \tilde{a}_{ij}(k+1) + 1, \qquad \forall\ (i,j) \in \mathcal{A}, \tag{30}$$

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 Eq. (30) 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 the ones of the previous section.

5. MULTIPLE DESTINATIONS AND ORIGINS; PARALLEL IMPLEMENTATION

We first 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 Prop. 3.

The case of a single destination and multiple origins is particularly interesting because it is well suited for parallel computation in a way that Dijkstra's algorithm is not. The idea is to maintain a common price vector p but a different path P_i for each origin i.

Briefly, the algorithm maintains CS of all the pairs (P_i, p), and the typical iteration is executed simultaneously for all origins i. At the end of an iteration, the results corresponding to the different origins must be 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 origins, i.e., a path extension or a path contraction and corresponding price change will occur simultaneously for all 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 must be 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. For this reason, 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.

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]. Synchronous and asynchronous implementations of the algorithm will be discussed in a different paper.

6. RELATION TO DUAL COORDINATE ASCENT

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

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

subject to

$$\sum_{\{j|(i,j)\in\mathcal{A}\}} x_{ij} - \sum_{\{j|(j,i)\in\mathcal{A}\}} x_{ji} = s_i, \qquad \forall\, i \in \mathcal{N}, \tag{31}$$

$$0 \le x_{ij} \le 1, \qquad \forall\, (i,j) \in \mathcal{A}, \tag{32}$$

where

$$s_1 = 1, \qquad s_m = -1$$

$$s_i = 0, \qquad \forall\, i \neq 1, t,$$

and t is the given destination.

We formulate a dual problem to (LNF) by associating a Lagrange multiplier p_i with each conservation of flow constraint (31). Letting p be the vector with elements p_i, $i \in \mathcal{N}$, we can write the

corresponding Lagrangian function as

$$L(x,p) = \sum_{(i,j)\in\mathcal{A}} (a_{ij} + p_j - p_i)x_{ij} + p_1 - p_t.$$

One obtains the dual function value $q(p)$ at a vector p by minimizing $L(x,p)$ over all flows x satisfying the capacity constraints (10). This leads to the dual problem

$$\begin{aligned} &\text{maximize } q(p) \\ &\text{subject to no constraint on } p, \end{aligned} \tag{33}$$

with the dual functional q given by

$$q(p) = \min_x \{L(x,p) \mid 0 \le x_{ij} \le 1, (i,j) \in \mathcal{A}\} = \sum_{(i,j)\in\mathcal{A}} q_{ij}(p_i - p_j) + p_1 - p_t, \tag{34}$$

where for all $(i,j) \in \mathcal{A}$,

$$q_{ij}(p_i - p_j) = \begin{cases} a_{ij} + p_j - p_i & \text{if } a_{ij} + p_j < p_i, \\ 0 & \text{otherwise.} \end{cases} \tag{35}$$

This formulation of the dual problem has been used in many prior works dealing with relaxation methods. We note that standard duality results imply that the optimal primal cost equals the optimal dual cost.

Let us associate with a given path $P = (1, i_1, i_2, \ldots, i_k)$ 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 network complementary slackness conditions

$$x_{ij} < 1 \quad \Rightarrow \quad p_i \le a_{ij} + p_j \qquad \forall\, (i,j) \in \mathcal{A}, \tag{36}$$

$$0 < x_{ij} \quad \Rightarrow \quad p_i \ge a_{ij} + p_j \qquad \forall\, (i,j) \in \mathcal{A}. \tag{37}$$

Equation (36) is equivalent to the condition $p_i \le a_{ij} + p_j$ for all $(i,j) \in \mathcal{A}$, since the constraint $x_{ij} \le N$ is not active for any arc flow. For a pair (x,p), the above conditions together with primal feasibility (the conservation of flow constraint (31) 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.

We finally provide an interpretation of the algorithm as a dual ascent method. From Eqs. (34) and (35) it can be seen that since the CS condition (1a) holds throughout the algorithm, we have $q_{ij}(p_i - p_j) = 0$ for all (i,j) throughout the algorithm, and $q(p) = p_1 - p_t$. Thus 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 unchanged. 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 single origin-destination pair version of the algorithm without arc length scaling and with the default initialization ($P = (1)$, $p = 0$) was implemented in a code called FORPATH. Two other codes, due to Gallo and Pallotino [GaP88], were used for comparison. They are implementations of Dijkstra's method, called SDKSTR and SHEAP, that use a queue and a binary heap, respectively, to store the nodes which are not yet permanently labeled. We made a single line modification to these codes so that they terminate when the unique destination is 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. SDKSTR is typically less efficient than SHEAP, being a less sophisticated implementation. We did not compare our code with label correcting methods, since these methods are at a disadvantage when there is only one origin-destination pair.

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 three codes on a Macintosh Plus equiped with a Radius16 accelerator are shown in Tables 1 and 2 (length ranges [1,10] and [1,1000], respectively). The tables also show the *proximity rank* of the destination, defined as the number of nodes that are at no larger distance from the origin than the destination; this is also the total number of permanently labeled nodes by the Dijkstra codes.

The tables show that FORPATH is not quite as fast as SHEAP. However, the difference in the running times is not large for the randomly generated problems tested, and it is hoped that in a suitable parallel computing environment FORPATH will gain an advantage. The reader is warned that the computational results of the tables are very preliminary. Clearly one can find problems where FORPATH is vastly inferior to the Dijkstra codes in view of its inferior computational complexity. The important question is whether there are types of practical problems for which our algorithm is

well suited. Given the novelty of the algorithm, we find the results of Tables 1 and 2 encouraging. However, more solid conclusions on the computational merits of our algorithm must await further research and testing with both serial and parallel machines.

N	A	Proximity Rank	FORPATH	SDKSTR	SHEAP
500	2000	422	0.1997	0.8164	0.1328
500	5000	181	0.0669	0.5332	0.0840
1000	4000	962	0.6167	3.500	0.3340
1000	10000	509	0.1997	2.684	0.2676
2000	8000	1221	0.5332	11.02	0.5332
2000	20000	1762	1.117	18.85	0.9023
3000	12000	2651	1.467	28.46	1.016
3000	30000	2443	1.434	41.04	1.230

Table 1: Solution times of shortest path codes on a Mac+ using problems generated by NETGEN. The lengths of all arcs were randomly generated from the range [1,10].

N	A	Proximity Rank	FORPATH	SDKSTR	SHEAP
500	2000	130	0.0669	0.2334	0.0501
500	5000	115	0.0832	0.3833	0.0835
1000	4000	646	0.4834	2.717	0.2666
1000	10000	302	0.2332	1.933	0.2000
2000	8000	44	0.0166	0.0664	0.0332
2000	20000	1720	2.217	18.87	0.9336
3000	12000	2461	2.317	27.88	1.034
3000	30000	1950	2.050	37.23	1.199

Table 2: Solution times of shortest path codes on a Mac+ using problems generated by NETGEN. The lengths of all arcs were randomly generated from the range [1,1000].

REFERENCES

[AMO89] Ahuja, R. K., Magnanti, T. L., and Orlin, J. B., "Network Flows", Sloan W. P. No. 2059-88, M.I.T., 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] Bertsekas, D. P., "A Distributed Algorithm for the Assignment Problem", Lab. for Information and Decision Systems Working Paper, M.I.T., March 1979.

[Ber82] Bertsekas, D. P., "Distributed Dynamic Programming", IEEE Trans. on Aut. Control, Vol. AC-27, 1982, pp. 610-616.

[Ber86] Bertsekas, D. P., "Distributed Relaxation Methods for Linear Network Flow Problems", Proceedings of 25th IEEE Conference on Decision and Control, 1986, pp. 2101-2106.

[Ber87] Bertsekas, D. P., Dynamic Programming: Deterministic and Stochastic Models, Prentice-Hall, Englewood Cliffs, N. J., 1987.

[BeE88] Bertsekas, D. P., and Eckstein, J., "Dual Coordinate Step Methods for Linear Network Flow Problems", Math. Progr., Series B, Vol. 42, 1988, pp. 203-243.

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

[DGK79] Dial, R., Glover, F., Karney, D., and Klingman, D., "A Computational Analysis of Alternative Algorithms and Labeling Techniques for Finding Shortest Path Trees", Networks, Vol. 9, 1979, pp. 215-248.

[Dia69] Dial, R. B., "Algorithm 360: Shortest Path Forest with Topological Ordering", Commun. A.C.M., Vol. 12, 1969, pp. 632.

[GaP86] Gallo, G., and Pallotino, S., "Shortest Path Methods: A Unified Approach", Math. Programming Study, Vol. 26, 1986, p. 38.

[GaP88] Gallo, G., and Pallotino, S., "Shortest Path Algorithms", Annals of Operations Research, Vol. 7, 1988.

[KNS74] Klingman, D., Napier, A., and Stutz, J., "NETGEN - A Program for Generating Large Scale (Un) Capacitated Assignment, Transportation, and Minimum Cost Flow Network Problems", Management Science, Vol. 20, 1974, pp. 814-822.

[Ker81] Kershenbaum, A., "A Note on Finding Shortest Path Trees", Networks, Vol. 11, 1981, p. 399.

[Pap74] Pape, U., "Implementation and Efficiency of Moore - Algorithms for the Shortest Path Problem", Math. Programming, Vol. 7, 1974, pp. 212-222.

[Pea84] Pearl, J., Heuristics, Addison-Wesley, Reading, Mass., 1984.

[ShW81] Shier, D. R., and Witzgall, C., "Properties of Labeling Methods for Determining Shortest Path Trees", J. Res. Natl. Bureau of Standards, Vol. 86, 1981, p. 317.