TRANSPORTATION SCIENCE
Vol. 39, No. 4, November 2005, pp. 465–476
ISSN 0041-1655 | EISSN 1526-5447 | 05 | 3904 | 0465

DOI 10.1287/trsc.1050.0124
© 2005 INFORMS

A Multiple Pairs Shortest Path Algorithm

I-Lin Wang
Department of Industrial and Information Management, National Cheng Kung University, No. 1, University Road, Tainan, Taiwan 701, ilinwang@mail.ncku.edu.tw

Ellis L. Johnson, Joel S. Sokol
School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0205
{ejohnson@isye.gatech.edu, jsokol@isye.gatech.edu}

The multiple pairs shortest path problem (MPSP) arises in many applications where the shortest paths and distances between only some specific pairs of origin-destination (OD) nodes in a network are desired. The traditional repeated single-source shortest path (SSSP) and all pairs shortest paths (APSP) algorithms often do unnecessary computation to solve the MPSP problem. We propose a new shortest path algorithm to save computational work when solving the MPSP problem. Our method is especially suitable for applications with fixed network topology but changeable arc lengths and desired OD pairs. Preliminary computational experiments demonstrate our algorithm's superiority on airline network problems over other APSP and SSSP algorithms.

Key words: shortest path; multiple pairs; algebraic method; LU decomposition; Carré's algorithm
History: Received: April 2004; revision received: May 2005; accepted: May 2005.

Introduction

The multiple pairs shortest path (MPSP) problem on a network is to compute the shortest paths for q specific origin destination (OD) pairs (s_i, t_i), $i = 1, \ldots, q$. This problem arises often in multicommodity networks (Barnhart et al. 1995) such as telecommunication and transportation networks. In this paper, we propose a new algorithm that saves computational work when compared to the methods currently used to solve MPSP problems. Our algorithm is especially effective when shortest paths between specific sets of OD pairs have to be repeatedly computed using different arc costs.

During the last four decades, many good shortest path algorithms have been developed. We can group shortest path algorithms into three classes: (1) those that employ combinatorial or network traversal techniques such as label-setting methods (Dijkstra 1959, Dantzig 1960, Dial 1965), label-correcting methods (Ford 1956, Moore 1957, Bellman 1958, Pape 1974), and their hybrids (Glover, Glover, and Klingman 1984); (2) those that employ linear programming (LP)-based techniques like the primal network simplex method (Goldfarb, Hao, and Kai 1990; Goldfarb and Jin 1999) and the dual ascent method (Bertsekas, Pallottino, and Scutellà 1995; Pallottino and Scutellà 1997); and (3) those that use algebraic or matrix techniques such as Floyd-Warshall (Floyd 1962, Warshall 1962) and Carré's (1969, 1971) algorithms. The first two groups of shortest path algorithms are mainly designed to solve the single-source (*or* sink) shortest path (SSSP) problem, which is the problem of computing a shortest path tree for a specific source (or sink) node. Algebraic shortest path algorithms, on the other hand, are more suitable for solving the all pairs shortest paths (APSP) problem, which is the problem of computing shortest paths for all the node pairs.

Currently, SSSP and APSP algorithms are used to solve MPSP problems. Obviously, the MPSP problem can be solved by simply applying an SSSP algorithm $\hat{q}$ times, where $\hat{q}$ is the size of a minimum node cover on an appropriately defined bipartite graph. Given the set N of nodes in the MPSP network, the bipartite graph includes two copies of each node: one representing that node's use as an origin and one representing its use as a destination. For each required shortest path, the bipartite graph includes an arc from the node representing the path's origin to the node representing its destination. The minimum node cover on this bipartite graph (i.e., the minimum set of nodes that includes at least one endpoint of each arc) corresponds to the minimum number of SSSP calls necessary to solve the MPSP problem. More specifically, any origin node i included in the node cover corresponds to using SSSP to find a tree of shortest paths out of i, and any destination node j included in the node cover corresponds to using SSSP to find a tree of shortest paths into j. Because this method requires many calls to SSSP, we call such methods *repeated SSSP algorithms*.

It is easy to see that repeated SSSP algorithms are more efficient for MPSP problems with small node covers (i.e., $\hat{q} \ll n$). However, for cases with larger node covers, both the APSP and SSSP methods may involve more computation than necessary. To cite an extreme example, suppose that we want to obtain shortest paths for n OD pairs, (s_i, t_i), $i = 1, \ldots, n$, which correspond to a matching on $N \times N$. That is, each node appears exactly once in the source and sink node set. For this specific example, we must apply an SSSP algorithm exactly n times, which is as hard as solving an APSP problem. Both the APSP and SSSP methods are "overkill" in that they waste computational effort by finding shortest paths for many unwanted OD pairs in the process.

The MPSP problem can also be solved by applying an algebraic APSP algorithm once and extracting the desired OD entries. Algebraic APSP algorithms are closely related to *path algebra*, an algebraic system that is applicable to several path-finding problems (Carré 1971; Backhouse and Carré 1975). The operators ($\oplus$, $\otimes$, *null*, *e*) in path algebra have the following meanings: $a \oplus b$ means $\min\{a, b\}$; $a \otimes b$ means $a + b$; *null* (i.e., 0) means ∞, and *e* (i.e., identity) means 0. In particular, the APSP problem can be interpreted as determining the $n \times n$ shortest distance matrix $X = [x_{ij}]$ that satisfies $X = CX \oplus I_n$ (Carré 1971), where $C = [c_{ij}]$ is the $n \times n$ measure matrix storing the length of arc (i, j) and I_n is the identity matrix. In other words, $X = CX \oplus I_n$ is exactly Bellman's equation: For each node pair (i, j), $x_{ij} = \min_{k \neq i, j}\{c_{ik} + x_{kj}\}$ if $i \neq j$, and $x_{ij} = 0$ if $i = j$. Techniques analogous to Gauss-Jordan and Gaussian elimination (direct method) correspond to the well-known Floyd-Warshall and Carré's algorithms, respectively (see Carré 1971 for proofs of their equivalence). The decomposition algorithm proposed by Mills (1966) (also, see Hu 1968) decomposes a large graph into parts, solves APSP for each part separately, and then reunites the parts. All of these methods have $O(n^3)$ time bounds and are believed to be efficient for dense graphs (Ahuja, Magnanti, and Orlin 1993).

The problem of inverting a matrix is closely related to a series of matrix powers. In particular, the optimal APSP distance matrix $X^* = C^{n-1}$. Aho, Hopcroft, and Ullman (1974, pp. 202–206) showed that computing C^{n-1} is as hard as a single-distance matrix squaring, which takes $O(n^3)$ time. Fredman (1976) proposed an $O(n^{2.5})$ algorithm to compute a single-distance matrix squaring but required a program of exponential size. Its practical implementation, improved by Takaoka (1992), still takes $O(n^3((\log\log n)/\log n)^{1/2})$, which is just slightly better. Recently, much work has been done in using block decomposition and fast matrix multiplication techniques to solve the APSP problem. These new methods, although they have better subcubic time bounds, usually require the arc lengths to be either integers of small absolute value (Zwick 1998) or can only be applied to unweighted, undirected graphs (Seidel 1995; Galil and Margalit 1997). All of these matrix multiplication algorithms seem to be more suitable for dense graphs because they do not exploit sparsity. However, their practical efficiency remains to be evaluated.

Carré's algebraic APSP algorithm (1969, 1971) uses Gaussian elimination to solve $X = CX \oplus I_n$. After a LU decomposition procedure, Carré's algorithm performs n applications of forward elimination and backward substitution procedures. In turn, each forward/backward operation gives an optimal solution to one column of X, which corresponds to an ALL-1 shortest-distance vector. This decomposability of Carré's algorithm makes it more attractive for MPSP problems than the Floyd-Warshall algorithm.

In this paper, we propose an algebraic algorithm designed specifically for the MPSP problem, inspired by Carré's APSP algorithm. When solving MPSP problems, our algorithm avoids unnecessary operations that other algorithms must perform. Preliminary computational experiments show that our algorithm performs well and is faster than state-of-the-art SSSP and APSP algorithms.

This paper contains four sections. Section 1 introduces some definitions and basic concepts. Section 2 presents our MPSP algorithm (*DLU*) and proves its correctness. Section 3 demonstrates classes of MPSP problems in which our algorithm saves computational effort compared with APSP and repeated SSSP algorithms, and contains computational results that demonstrate the superiority of our algorithm for airline network problems. Section 4 concludes our work and proposes future research.

1. Preliminaries

For a *digraph* $G := (N, A)$ with $n = |N|$ nodes and $m = |A|$ arcs, a *measure matrix* $[c_{ij}]$ is the $n \times n$ array in which element c_{ij} denotes the length of arc (i, j) with *tail* i and *head* j. $c_{ij} := \infty$ if $(i, j) \notin A$. A *walk* is a sequence of r nodes $(n_1, n_2, \ldots, n_r)$ composed of $(r - 1)$ arcs, (n_{k-1}, n_k), where $2 \leq k \leq r$. A *path* is a walk without repeated nodes. A *cycle* is a walk without repeated nodes, except that the starting and ending nodes are the same. The length of a path (cycle) is the sum of the lengths of its arcs. When we refer to a shortest path tree with root t, we mean a tree rooted at a sink node t in which all the tree arcs point toward t.

The *distance matrix* $[x_{ij}]$ is an $n \times n$ array in which x_{ij} records the length of a path from i to j. Let $[succ_{ij}]$ denote an $n \times n$ *successor matrix* in which $succ_{ij}$ represents the node that immediately follows i in a path from i to j. We could construct a path from i to j by tracing the successor matrix. In particular, suppose that $i \to k_1 \to k_2 \to \cdots \to k_r \to j$ is a path in G from i to j, then $k_1 = succ_{ij}$, $k_2 = succ_{k_1 j}, \ldots, k_r = succ_{k_{r-1} j}$, and $j = succ_{k_r j}$. Let x^*_{ij} denote the shortest distance from i to j in G, and let $succ^*_{ij}$ denote the successor of i on the shortest path.

A *triple comparison* $s \to k \to t$, which compares $x_{sk} + x_{kt}$ with x_{st}, is a process to update the length of arc (s, t) to be $\min\{x_{st}, x_{sk} + x_{kt}\}$ or to add a *fill-in* arc (s, t) to the original graph with a length equal to $x_{sk} + x_{kt}$, if $(s, t) \notin A$. Because shortest path algorithms operate by performing sequences of triple comparisons (Carré 1971), we can measure the efficiency of algorithms by counting the number of triple comparisons they perform.

We say that node i is *higher* (*lower*) than node j if the indices satisfy $i > j$ ($i < j$). A node i in a set *LIST* is said to be the *highest* (*lowest*) node in *LIST* if $i \geq k$ ($i \leq k$) $\forall k \in LIST$. An arc (i, j) is pointing *downwards* (*upwards*) if $i > j$ ($i < j$) (see Figure 1).

Define an induced subgraph denoted $H(S)$ on the node set S, which contains only arcs (i, j) of G with both ends i and j in S. Let $a < b$ and $[a, b]$ denote the set of nodes $\{a, (a+1), \ldots, (b-1), b\}$. Figure 1 illustrates examples of $H([a, b])$ and $H([1, a] \cup b)$. Thus, $H([1, n]) \equiv G$ and can be decomposed into three subgraphs for any given OD pair (s, t): (1) $H([1, \min\{s, t\}] \cup \max\{s, t\})$; (2) $H([\min\{s, t\}, \max\{s, t\}])$; and (3) $H(\min\{s, t\} \cup [\max\{s, t\}, n])$. Thus, any shortest path in G from s to t is the shortest path among these three induced subgraphs. This paper gives an algebraic algorithm that systematically calculates shortest paths for these cases to obtain a shortest path in G from s to t.

Inspired by Carré's algorithm, we propose Algorithm *DLU*, which further reduces computations required for MPSP problems. We use the name *DLU* for our algorithm because it contains procedures similar to the LU decomposition in Carré's algorithm and is more suitable for dense graphs. Not only can our algorithm decompose a MPSP problem as Carré's algorithm does, it can also compute the requested OD shortest distances without the need of shortest path trees as required by Carré's algorithm. Therefore, our algorithm saves computational work over other APSP algorithms and is advantageous for problems in which only distances (not paths) are required. For problems that require tracing of shortest path for a particular OD pair (s, t), *DLU* traces a shortest path without the need of computing the entire shortest path tree.

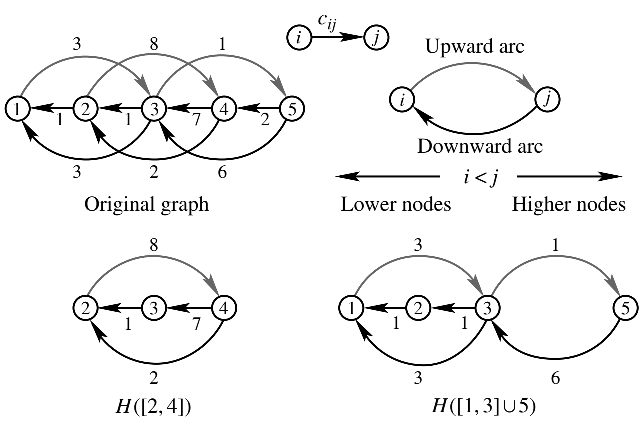

Figure 1 Illustration of Node Ordering and Subgraphs $H([2, 4])$, $H([1, 3] \cup 5)$

2. Algorithm *DLU*

Given a set of q requested OD pairs $Q := \{(s_i, t_i): i = 1, \ldots, q\}$, Algorithm *DLU* first initializes $[x_{ij}] := [c_{ij}]$ and $[succ_{ij}] := [j]$, and then performs two procedures: (1) *A_LU*; and (2) *Get_D*(s_i, t_i) for $i = 1, \ldots, q$. In particular, to find a shortest path in G from s to t, *A_LU* first calculates a shortest path in the subgraph $H([1, \min\{s, t\}] \cup \max\{s, t\})$, and then *Get_D*$(s, t)$ further considers the subgraphs $H([\min\{s, t\}, \max\{s, t\}])$ and $H(\min\{s, t\} \cup [\max\{s, t\}, n])$ to find a shortest path in G. Details about each procedure are discussed in the following sections.

Algorithm 1 *DLU*$(Q := \{(s_i, t_i): i = 1, \ldots, q\})$

```
begin
  Initialize [$x_{ij}$] and [$succ_{ij}$];
  A_LU;
  for $i = 1$ to $q$ do
    Get_D($s_i, t_i$);
    if shortest paths need to be traced then
      if $x_{s_i t_i} \neq \infty$ then
        Get_P($s_i, t_i$);
      else there exists no path from $s_i$ to $t_i$
end
```

2.1. Procedure *A_LU*

The first procedure, *A_LU*, resembles the LU decomposition in Gaussian elimination. In the kth iteration of LU decomposition in Gaussian elimination, we use diagonal entry (k, k) to eliminate entry (k, t) for each $t > k$. This updates the $(n-k) \times (n-k)$ submatrix and creates fill-ins. Similarly, *A_LU* sequentially uses each node $k = 1, \ldots, (n-2)$ as an intermediate node to check whether to update each entry (s, t) of $[x_{ij}]$ and $[succ_{ij}]$ for all $k < s \leq n$ and $k < t \leq n$. An update is performed whenever $x_{sk} < \infty$, $x_{kt} < \infty$, and $x_{st} > x_{sk} + x_{kt}$. Figure 2(a) illustrates the operations of *A_LU* on a 5-node graph. It sequentially uses nodes 1, 2, and 3 as intermediate nodes to update the remaining 4×4, 3×3, and 2×2 submatrix of $[x_{ij}]$ and $[succ_{ij}]$.

Graphically speaking, *A_LU* can be viewed as a process of constructing an *augmented graph* G' obtained by either adding fill-in arcs or changing some arc lengths on the original graph when better paths are identified using intermediate nodes. In *A_LU* only intermediate nodes with indices smaller than both end nodes of the path are considered. For example, in Figure 3, *A_LU* adds fill-in arc $(2, 3)$ because $2 \to 1 \to 3$ is a shorter path than the direct

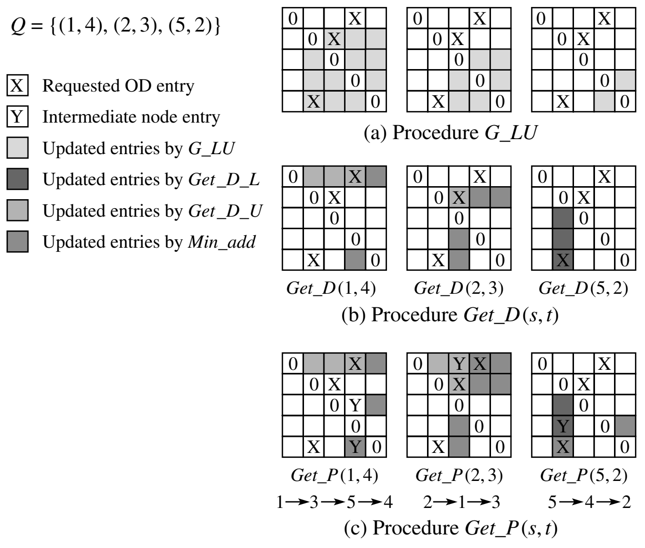

Figure 2 **Solving a Three Pairs Shortest Path Problem on a Five-Node Graph by Algorithm** ***DLU*(*Q*)**

arc from node 2 to node 3 (infinity, in this case). Similarly, the procedure also adds fill-in arcs(3, 4), (4, 5) and modifies the length of original arc (4, 3).

```
Procedure A_LU
begin
  for $k = 1$ to $n-1$ do
    for $s = k+1$ to $n$ do
      for $t = k+1$ to $n$ do
        if $s = t$ and $x_{sk} + x_{kt} < 0$ then
          Found a negative cycle; STOP
        if $s \neq t$ and $x_{st} > x_{sk} + x_{kt}$ then
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end
```

A_LU performs triple comparisons $s \to k \to t$ for each $s \in [2, n]$, $t \in [2, n]$ and for each $k = 1, \ldots, (\min\{s, t\} - 1)$. In particular, for every node pair (s, t), shortest paths in $H([1, \min\{s, t\}] \cup \max\{s, t\})$ will be computed, and thus $x_{n,n-1} = x^*_{n,n-1}$ and $x_{n-1,n} = x^*_{n-1,n}$ because $H([1, n-1] \cup n) = G$ (see Corollary 2).

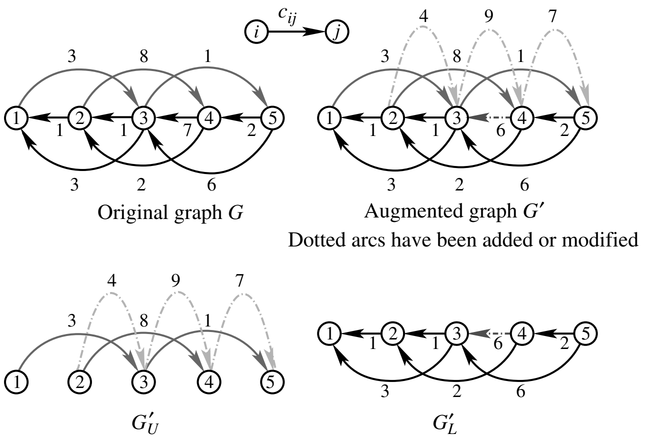

Figure 3 **Augmented Graph After Procedure** ***A_LU***

Theorem 1. *After Procedure A_LU is performed, $[x_{st}]$ represents the length of the shortest path from s to t in $H([1, \min\{s, t\}] \cup \max\{s, t\})$. That is, A_LU calculates the shortest path from s to t using only intermediate nodes with indices less than both s and t.*

Proof. Suppose such a shortest path in G from s to t contains p arcs. In the case of $p = 1$, the result is trivial. Let us consider the case of $p > 1$. That is, $s := v_0 \to v_1 \to v_2 \to \cdots \to v_{p-2} \to v_{p-1} \to v_p := t$ is a shortest path in G from s to t with p arcs and with $(p-1)$ intermediate nodes whose indices are all smaller than $\min\{s, t\}$.

Let $v_\alpha < \min\{s, t\}$ be the lowest node in this shortest path. In the $k = v_\alpha$ iteration, *A_LU* will modify the length of arc $(v_{\alpha-1}, v_{\alpha+1})$ (or add this arc, if it does not exist in G') to the sum of the arc lengths of $(v_{\alpha-1}, v_\alpha)$ and $(v_\alpha, v_{\alpha+1})$. Thus, we obtain another path $s \to v_1 \to \cdots \to v_{\alpha-1} \to v_{\alpha+1} \to \cdots \to v_{p-1} \to t$ with $(p-1)$ arcs that is as short as the previous one. Next, *A_LU* repeats the same procedure that eliminates the new lowest node and constructs another path that is just as short, but contains one fewer arc. By induction, in the $k = \min\{s, t\}$ iteration, *A_LU* eventually modifies (or adds if $(s, t) \notin A$) arc (s, t) with length equal to that of the shortest path from s to t in $H([1, \min\{s, t\}] \cup \max\{s, t\})$.

Therefore, any arc (s, t) in G' corresponds to a shortest path in $H([1, \min\{s, t\}] \cup \max\{s, t\})$ from s to t with length x_{st}. Because any shortest path in G from s to t that passes through only intermediate nodes with indices smaller than $\min\{s, t\}$ corresponds to the same shortest path in $H([1, \min\{s, t\}] \cup \max\{s, t\})$, Procedure *A_LU* correctly computes the length of such a shortest path and stores it as the length of arc (s, t) in G'. ☐

Corollary 2. (a) *Procedure A_LU will correctly compute $x^*_{n,n-1}$ and $x^*_{n-1,n}$.*

(b) *For every node pair (s, t), Procedure A_LU will correctly compute shortest paths in $H([1, \min\{s, t\}] \cup \max\{s, t\})$.*

Proof. (a) This follows immediately from Theorem 1 because all other nodes have indices less than $(n-1)$ and n, so $H([1, n-1] \cup n) = G$.

(b) This follows immediately from Theorem 1. ☐

The next result demonstrates that any negative cycle will also be identified in Procedure *A_LU*.

Theorem 3. *Procedure A_LU will identify the presence of a negative cycle in G if one exists.*

Proof. Suppose that there exists a p-node cycle C_p, $i_1 \to i_2 \to i_3 \to \cdots \to i_p \to i_1$, with negative length. Without loss of generality, let i_1 be the lowest node in C_p, i_r be the second lowest, i_s be the second highest, and i_t be the highest. Let $length(C_p)$ denote the length function of cycle C_p. Because C_p is a negative cycle, $length(C_p) = \sum_{(i,j)\in C_p} c_{ij} < 0$.

In *A_LU*, before we begin iteration $k = i_1$ (using i_1 as the intermediate node), the length of some arcs of C_p might have already been modified, but no arcs of C_p will have been removed nor will $length(C_p)$ have increased. After iteration $k = i_1$, the updated graph contains a cycle C_{p-1} which skips i_1, connects i_p and i_2 by arc (i_p, i_2), and contains one fewer arc than C_p. In particular, C_{p-1} is $i_p \to i_2 \to \cdots \to i_{p-1} \to i_p$, and $length(C_{p-1}) = length(C_p) - (x_{i_1 i_2} + x_{i_p i_1} - x_{i_p i_2})$. Because $x_{i_p i_2} \le x_{i_1 i_2} + x_{i_p i_1}$ by the algorithm, we obtain $length(C_{p-1}) \le length(C_p) < 0$. The lowest-index node in C_{p-1} is now i_r, and we will again reduce the size of C_{p-1} by 1 in iteration $k = i_r$.

We iterate this procedure, each time processing the current lowest node in the cycle and reducing the cycle size by 1, until finally a 2-node cycle C_2, $i_s \to i_t \to i_s$, with $length(C_2) \le length(C_3) \le \cdots \le length(C_p) < 0$ is obtained. Therefore, $x_{ss} < 0$ and a negative cycle in the augmented graph G' is identified with cycle length smaller than or equal to the original negative cycle C_p. □

Thus, *A_LU* identifies the presence of a negative cycle, if one exists. (Note that the specific negative cycle can be recovered using Procedure *Get_P* described in §2.3.) It also computes the shortest distance in $H([1, \min\{s,t\}] \cup \max\{s,t\})$ from each node $s \in N$ to each node $t \in N \backslash \{s\}$. In other words, this procedure computes shortest path lengths for those requested OD pairs (s,t) whose shortest paths have all intermediate nodes with indices lower than $\min\{s,t\}$.

2.2. Procedure *Get_D*(s_i, t_i)

Given an OD pair (s_i, t_i), this procedure contains three subprocedures: *Get_D_L*(t_i), *Get_D_U*(s_i), and *Min_add*(s_i, t_i).

```
Procedure Get_D($s_i, t_i$)
begin
  Get_D_L($t_i$);
  Get_D_U($s_i$);
  Min_add($s_i, t_i$);
end
Subprocedure Get_D_L($t_i$)
begin
  for $s = t_i + 2$ to $n$ do
    for $k = t_i + 1$ to $s - 1$ do
      if $x_{st_i} > x_{sk} + x_{kt_i}$ then
        $x_{st_i} := x_{sk} + x_{kt_i}$; $succ_{st_i} := succ_{sk}$;
end
Subprocedure Get_D_U($s_i$)
begin
  for $t = s_i + 2$ to $n$ do
    for $k = s_i + 1$ to $t - 1$ do
      if $x_{s_i t} > x_{s_i k} + x_{kt}$ then
        $x_{s_i t} := x_{s_i k} + x_{kt}$; $succ_{s_i t} := succ_{s_i k}$;
end
Subprocedure Min_add($s_i, t_i$)
begin
  $r_i := \max\{s_i, t_i\}$;
  for $k = r_i + 1$ to $n$ do
    if $x_{s_i t_i} > x_{s_i k} + x_{kt_i}$ then
      $x_{s_i t_i} := x_{s_i k} + x_{kt_i}$; $succ_{s_i t_i} := succ_{s_i k}$;
end
```

The lower and upper triangular parts of $[x_{ij}]$ induce two acyclic subgraphs, G'_L and G'_U, of augmented graph G'. G'_L (G'_U) contains all the downward (upward) arcs of G'. The arcs can be easily identified by drawing the nodes in ascending order of their indices from left to right as illustrated in Figure 3. Graphically, *Get_D_L*(t_i) and *Get_D_U*(s_i) compute the shortest path tree to t_i in G'_L and from s_i in G'_U, respectively. *Min_add*(s_i, t_i) then merges the resulting two shortest trees and computes $x^*_{s_i t_i}$ in the original graph G.

Get_D_L(t_i) resembles the forward step in Gaussian elimination. It performs triple comparisons to update $x_{st_i} := \min\{x_{st_i}, x_{sk} + x_{kt_i}\}$ for each $k = (t_i + 1), \ldots, (s-1)$, and for each $s = (t_i + 2), \ldots, n$. Because G'_L is acyclic, the updated x_{st_i} for each $s = (t_i+2), \ldots, n$ corresponds to the shortest distance in G'_L from each node $s > t_i$ to t_i, which in fact corresponds to the shortest distance in $H([1,s])$ from s to t_i (see Corollary 5(a)).

Get_D_U(s_i) is similar to *Get_D_L*(t_i), except it is applied to the upper-triangular part of $[x_{ij}]$ and $[succ_{ij}]$. Thus, it is applied to the induced subgraph G'_U. *Get_D_U*(s_i) updates $x_{s_i t}$ for each $t = (s_i + 2), \ldots, n$. The updated $x_{s_i t}$ corresponds to the shortest distance in G'_U from node s_i to each node $t > s_i$, which in fact corresponds to the shortest distance in $H([1,t])$ from s_i to t (see Corollary 5(b)).

Let $r_i = \max\{s_i, t_i\}$. After running *Get_D_U*(s_i) and *Get_D_L*(t_i), the shortest distance in $H([1, r_i])$ from s_i to t_i is computed. *Min_add*(s_i, t_i) continues the remaining triple comparisons necessary to compute $x^*_{s_i t_i}$ in G. In particular, it computes the length of the shortest paths in $H([1, r_i] \cup k)$ that must pass through an intermediate node k by adding $x_{s_i k}$ and x_{kt_i} for each $k = (r_i + 1)$ to n, and then computes $x^*_{s_i t_i} = \min_{k > r_i}\{x_{s_i t_i}, x_{s_i k} + x_{kt_i}\}$ (see Corollary 7).

Theorem 4. (a) *A shortest path in* $H([1,s])$ *from node* $s > t$ *to node* t *corresponds to a shortest path in* G'_L *from* s *to* t.

(b) *A shortest path in* $H([1,t])$ *from node* $s < t$ *to node* t *corresponds to a shortest path in* G'_U *from* s *to* t.

Proof. (a) Suppose that a shortest path in G from node $s > t$ to node t contains p arcs. In the case where $p = 1$, the result is trivial. Let us consider the case where $p > 1$. That is, $s \to v_1 \to v_2 \to \cdots \to v_{p-2} \to v_{p-1} \to t$ is a shortest path in G from node $s > t$ to node t with $(p-1)$ intermediate nodes whose indices are smaller than $\max\{s, t\} = s$.

In the case where every intermediate node has an index smaller than $\min\{s, t\} = t < s$, Theorem 1 already shows that *A_LU* will compute such a shortest path and store it as arc (s, t) in G'_L. So, we only need to discuss the case where there exists some intermediate node with an index in the range $[t+1, s-1]$.

Suppose that the shortest path contains two intermediate nodes i and j such that all nodes k in the path between i and j have smaller indices than i and j (i.e., $k < i$ and $k < j$). Then, *A_LU* will have already updated the distance between i and j to reflect this segment of the path. Therefore, without loss of generality, we can look at just the r intermediate nodes $\{u_i\colon i = 1, \ldots, r\}$ in the shortest path in G from s to t such that $s := u_0 > u_1 > u_2 > \cdots > u_{r-1} > u_r > u_{r+1} := t$. In essence, we break the shortest path into $(r+1)$ segments u_0 to u_1, u_1 to $u_2, \ldots,$ and u_r to u_{r+1}. Each shortest path segment $u_{k-1} \to u_k$ in G contains intermediate nodes that all have lower indices than u_k. Because Theorem 1 guarantees that *A_LU* will produce an arc (u_{k-1}, u_k) for any shortest path segment $u_{k-1} \to u_k$ and G'_L is acyclic, the original shortest path $s \to v_1 \to v_2 \to \cdots \to v_{p-2} \to v_{p-1} \to t$ in G will be reduced to the shortest path $s \to u_1 \to u_2 \to \cdots \to u_{r-1} \to u_r \to j$ in G'_L.

(b) Using an argument similar to (a) above, the result follows immediately. □

Corollary 5. (a) *After Procedure A_LU has run, subprocedure Get_D_L(t_i) will correctly compute shortest paths in $H([1, s])$ for all node pairs (s, t_i) such that $s > t_i$.*

(b) *After Procedure A_LU has run, subprocedure Get_D_U(s_i) will correctly compute shortest paths in $H([1, t])$ for all node pairs (s_i, t) such that $t > s_i$.*

Proof. (a) Because G'_L is acyclic, subprocedure *Get_D_L*(t_i) computes the shortest path tree in G'_L rooted at node t_i. By Theorem 4(a), a shortest path in G'_L from node $s > t_i$ to node t_i corresponds to a shortest path in G from s to t_i. So, s must be the highest node because all other nodes in this path in G'_L have a lower index than s. In other words, a shortest path in G'_L corresponds to the same shortest path in $H([1, s])$.

Including the case of $t_i = (n-1)$ and $s = n$, as discussed in Corollary 2(a), the result follows directly.

(b) Using a similar argument as in part (a), the result again follows directly. □

Lemma 6. (a) *Every shortest path in G from s to t that has a highest $h > \max\{s, t\}$ can be decomposed into two segments: a shortest path from s to h in G'_U and a shortest path from h to t in G'_L.*

(b) *Given a node r where $1 \le r \le n$, every shortest path in G from s to t can be determined as the shortest of the following two paths:* (i) *the shortest path from s to t in G that passes through only nodes $v \le r$, and* (ii) *the shortest path from s to t in G that must pass through some node $v > r$.*

Proof. (a) This follows immediately by combining Corollary 5(a) and 5(b).

(b) It is easy to see that every path from s to t either must pass through some node $v > r$ or not. Therefore, the shortest path from s to t must be the shorter of the minimum-length paths of each type. □

Corollary 7. *After conducting A_LU, Get_D_L(t_i), and Get_D_U(s_i), subprocedure Min_add(s_i, t_i) will correctly compute a shortest path in G for a requested OD pair (s_i, t_i).*

Proof. By Corollary 5, before conducting

$$Min\_add(s_i, t_i),$$

we will have obtained shortest paths in $H([1, r_i])$ from s_i to t_i, where $r_i = \max\{s_i, t_i\}$. To obtain the shortest path in G from s_i to t_i, we only need to compare the results of *Get_D_L*(t_i) and *Get_D_U*(s_i) with the shortest paths that pass through node k for each $k = (r_i+1), \ldots, n$. By Lemma 6(a), a shortest path can be decomposed into two segments: from s_i to k in G'_U and from k to t_i in G'_L. Note that the shortest distances of the segments x_{s_ik} and x_{kt_i} will have been calculated by *Get_D_U*(s_i) and *Get_D_L*(t_i), respectively. Thus, $x_{s_ik} + x_{kt_i}$ corresponds to the length of the shortest path that must pass through node k in $H([1, k])$ from s_i to t_i. Lemma 6(b) (with $r = r_i$) shows that by computing $\min_{k > r_i}\{x_{s_it_i}, x_{s_ik} + x_{kt_i}\}$, *Min_add*$(s_i, t_i)$ correctly computes $x^*_{s_it_i}$. □

Theorem 8. *Procedure Get_D(s_i, t_i) will correctly compute $x^*_{s_it_i}$ and succ$^*_{s_it_i}$ for a given OD pair (s_i, t_i).*

Proof. This follows immediately by combining Corollary 2(b), Corollary 5(a) and 5(b), and Corollary 7. □

Figure 2(b) illustrates how *Get_D*(s_i, t_i) individually solves $x^*_{s_it_i}$ for each requested OD pair (s_i, t_i). For example, to obtain x^*_{23}, it first applies subprocedure *Get_D_U*(2) to update x_{23}, x_{24}, and x_{25}, then updates x_{43} and x_{53} using subprocedure *Get_D_L*(3). Finally, *Get_D*(2, 3) computes $\min\{x_{23}, (x_{24} + x_{43}), (x_{25} + x_{53})\}$ which gives x^*_{23}.

Note that the correctness of *DLU* depends only on the order in which triple comparisons are conducted, and not on path-tracing operations. Therefore, the algorithm is still correct even if we do not conduct the successor updating operations. This is similar to other algebraic algorithms such as Floyd-Warshall's algorithm, but is very different from the conventional

SSSP algorithms. The consequence is that we can compute a shortest path length without knowing how the path is constructed. This is advantageous for applications that require only shortest distances and not the specific shortest paths.

If, on the other hand, an entire shortest path from s to t needs to be traced, the following procedure $Get\_P(s, t)$ will iteratively compute all the intermediate nodes in a shortest path from s to t.

2.3. Procedure $Get\_P(s_i, t_i)$

DLU does only the necessary computations to get the shortest distance for each requested OD pair (s_i, t_i). Procedure $Get\_P(s_i, t_i)$ traces the shortest paths calculated by the rest of the algorithm. Note that if only the distances (not the paths themselves) are required, this procedure may be skipped.

```
Procedure $Get\_P(s_i, t_i)$
begin
  let $k := succ_{s_i t_i}$;
  while $k \neq t_i$ do
    $Get\_D(k, t_i)$;
    let $k := succ_{k t_i}$
end
```

Procedure $Get\_P(s_i, t_i)$ iteratively calls procedure $Get\_D(k, t_i)$ to update x_{kt_i} and $succ_{kt_i}$ for every node k that lies on the shortest path from s_i to t_i. In particular, starting from the successor of the origin node s_i, we check whether it coincides with the destination t_i. If not, we update its shortest distance and successor, and then visit the successor. We iterate this procedure until, eventually, the destination t_i is encountered. Because each intermediate node on this path has correct shortest distance and successor (by the correctness of procedure Get_D (see Theorem 8)), an entire shortest path is obtained.

For example, suppose that $1 \to 3 \to 5 \to 4$ is a shortest path from node 1 to node 4 in Figure 2(c). *DLU* first computes x^*_{14} and $succ^*_{14}$. Because $succ^*_{14} = 3$, which means node 3 is the successor of node 1 in this shortest path, the next values to be computed are x^*_{34} and $succ^*_{34}$. Finally, because $succ^*_{34} = 5$, it computes x^*_{54} and $succ^*_{54}$. Node 5 is the last intermediate node in the shortest path because $succ^*_{54} = 4$. Thus, procedure $Get\_P(1, 4)$ gives all the intermediate nodes and their shortest distances to the sink node 4.

To obtain a shortest path tree rooted at sink node t, we set $Q := \{(i, t)\colon i \neq t,\ i = 1, \dots, n\}$. Setting $Q := \{(i, j)\colon i \neq j,\ i = 1, \dots, n,\ j = 1, \dots, n\}$ is sufficient to solve an APSP problem.

2.4. Complexity and Implementation of Algorithm *DLU*

For an instance of MPSP where $Q = \{(s_i, t_i)\colon i = 1, \dots, q\}$, let $|Q_s|$ denote the size of the requested origin node set $Q_s := \{s_i\colon i = 1, \dots, q\}$ and let $|Q_t|$ denote the size of the requested destination node set $Q_t := \{t_i\colon i = 1, \dots, q\}$. *DLU* performs one iteration of Procedure *A_LU*, q iterations of procedure *Get_D*, which includes $|Q_t|$ iterations of subprocedure *Get_D_L* and $|Q_s|$ iterations of subprocedure *Get_D_U*, and q iterations of subprocedure *Min_add*.

In particular, Procedure *A_LU* performs

$$\sum_{k=1}^{n-2} \sum_{s=k+1}^{n} \sum_{t=k+1,\, s\neq t}^{n} (1) = \tfrac{1}{3} n(n-1)(n-2)$$

triple comparisons, if we skip the triple comparisons for self-loops. $|Q_t|$ iterations of *Get_D_L* require

$$\sum_{t\in Q_t} \sum_{s=t+2}^{n} \sum_{k=t+1}^{s-1} (1) = \tfrac{1}{2} \sum_{t\in Q_t} (n-t_i)(n-t_i-1)$$

triple comparisons. $|Q_s|$ iterations of *Get_D_U* require

$$\sum_{s\in Q_s} \sum_{t=s+2}^{n} \sum_{k=s+1}^{t-1} (1) = \tfrac{1}{2} \sum_{s\in Q_s} (n-s_i)(n-s_i-1)$$

triple comparisons. Finally, q iterations of *Min_add* require

$$\sum_{(s_i, t_i)\in Q} \sum_{k=r_i+1}^{n} (1) = \sum_{(s_i, t_i)\in Q} (n-r_i)$$

triple comparisons, where $r_i := \max\{s_i, t_i\}$.

Thus, *DLU* has an $O(n^3)$ worst-case complexity. When solving an APSP problem on a complete graph K_n, *DLU* performs $n(n-1)(n-2)$ triple comparisons, which Nakamori (1972) has shown to be the minimum. Of these $n(n-1)(n-2)$ triple comparisons, 1/3 is contributed by Procedure *A_LU*, and 2/3 by *Get_D* (1/6 by *Get_D_U*, 1/6 by *Get_D_L*, and 1/3 by *Min_add*). Floyd-Warshall and Carré's algorithms also perform the same amount of triple comparisons, and are better than most SSSP algorithms; label-setting algorithms require $O(n^3)$ and label-correcting algorithms require $O(n^4)$. For problems on acyclic graphs, we can reorder the nodes so that the upper (or lower) triangular part of $[x_{ij}]$ becomes empty and only Procedure *A_LU* and either subprocedure *Get_D_L* or *Get_D_U* is required.

For sparse graphs, node ordering plays an important role in the efficiency of the algorithm. A bad node ordering will incur more fill-in arcs, similar to the fill-ins required in Gaussian elimination. Computing an ordering that minimizes the fill-ins is *NP*-complete (Rose and Tarjan 1978). Nevertheless, many fill-in reducing techniques such as Markowitz's (1957) rule, minimum degree method, and nested dissection method (see Duff, Erisman, and Reid 1989, Chapter 8) used in solving systems of linear equations can be exploited here to speed up *DLU*. Because our algorithm does computations on higher nodes before lower nodes, optimal distances can be obtained for higher nodes earlier than lower nodes. Thus, reordering the nodes so that the endpoints of the requested OD pairs have higher indices may also shorten the computational time, although such an ordering might incur more fill-in arcs. More details about the impact of node ordering will be discussed in a forthcoming paper (Wang 2005). Here, we use a predefined node ordering to start our algorithm.

Although *DLU* is an algebraic algorithm, its "graphical" implementation might greatly improve its practical efficiency. In particular, *A_LU* constructs an augmented graph G' (see Figure 3). We can use arc-adjacency lists to record the nontrivial entries (i.e., finite entries). If G' is sparse (i.e., with few fill-in arcs), then the shortest path computations of *Get_D_L* and *Get_D_U* on its acyclic subgraphs G'_L and G'_U can be efficiently implemented to avoid many trivial triple comparisons the algebraic algorithms must perform. Note that the efficiency of subprocedures *Get_D_L* and *Get_D_U* depends on the sparsity of augmented graph G'. Therefore, any fill-in reduction techniques discussed in the previous paragraph will not only reduce the running time of *A_LU*, but also make *Get_D_L* and *Get_D_U* faster.

Note that we may avoid repeated computation in *Get_D_L*(t_i) and *Get_D_U*(s_i) if some OD pairs in Q share the same origin node s_i or destination node t_i. Similarly, we may avoid repeated computations for some intermediate nodes when tracing a shortest path from s_i to t_i with *Get_P*. Thus, when solving an APSP problem, the complexity bound on *Get_P* remains $O(n^3)$ because it applies *Get_D_L* and *Get_D_U* (both take $O(n^2)$ time) at most n times. Note that *Min_add*(s, t) for each $s = 1, \ldots, n$ and $t = 1, \ldots, n$ takes $O(n^3)$ time as well.

In general, when solving an MPSP problem with $q < n^2$ OD pairs, *DLU* saves computational work compared to other algebraic algorithms. Unlike Carré's algorithm and label-correcting algorithms, which have to compute an entire shortest path tree rooted at t to trace a shortest path for a specific OD pair (s, t), *DLU* can retrieve such a path by successively traversing each intermediate node on that path. Thus, it is more efficient.

Next, we will give examples, including both dense and sparse graphs, to show the superiority of our algorithm over the APSP and SSSP algorithms.

3. Preliminary Computational Experiments

In this section, we show that our algorithm requires less computational effort than APSP or SSSP algorithms for many instances of MPSP. In addition to showing that our algorithm performs better on a class of dense graphs for which we can explicitly count triple comparisons, we also show that our algorithm is empirically superior by testing it on artificial grid networks and real airline flight networks. Our algorithm requires fewer triple comparisons and (consequently) less running time than the APSP and SSSP algorithms.

First, we present a class of graphs where our algorithm dominates the others. Consider a complete graph K_n, $n \geq 4$, which contains no negative cycle but may have negative arc lengths. Suppose that we want to compute the shortest distance for n requested OD pairs $\{(1, n), (2, n-1), (3, n-2), \ldots, (n/2-1, n/2+2), (n/2, n/2+1), (n/2+1, n/2), (n/2+2, n/2-1), \ldots, (n-1, 2), (n, 1)\}$. The Floyd-Warshall algorithm requires $(n-1)^2(n-2) + (n-2)$ triple comparisons; label-correcting SSSP algorithms also solve this MPSP as an APSP that takes $O(n^4)$. On the other hand, *DLU* requires $(2/3)n(n-1)(n-2)$ triple comparisons in *A_LU*, *Get_D_L*, and *Get_D_U*, and only $(1/4)n(n-2)$ triple comparisons in *Min_add*.

Compared with the Floyd-Warshall algorithm, *DLU* saves $(1/12)(4n^3 - 27n^2 + 62n - 24)$ triple comparisons when $n \geq 4$. *DLU* is also more efficient than the $O(n^4)$ label-correcting SSSP algorithms.

We also compare *DLU* with implementations of other shortest path algorithms in Cherkassky, Goldberg, and Radzik (1996) to study *DLU*'s practical efficiency on several classes of artificially generated grid networks and real flight networks that are both airline specific and region specific.

We use nine SSSP C codes (five label-correcting and four label-setting codes) written by Cherkassky, Goldberg, and Radzik (1996) with slight modification so that they can read the requested destination node set, Q_t, and then calculate shortest path trees rooted at each requested destination node in Q_t. Table 1 summarizes these SSSP codes.

We first evaluate the performance of *DLU* and other SSSP algorithms for solving MPSP problems with $|Q_t| = |Q_s| = 75\%|N|$ on two families of artificial grid networks (SPGRID-SQ and SPGRID-WL) generated by SPGRID, an artificial network generator written by Cherkassky, Goldberg, and Radzik (1996). SPGRID generates grid-like networks with $X \times Y$ grid nodes plus a super node. By changing X and Y we can specify the grid shape to be square (SPGRID-SQ), or wide or long (SPGRID-WL). We specify the degree to be 3 and arc lengths to range from 10^3 to 10^4, and then generate 8 square, 4 wide, and 4 long random grid networks. Each entry in the tables shows the performance of the algorithm as a ratio of its running time, or number of triple comparisons, to that of the fastest algorithm. Table 2 shows that label-correcting codes *TWOQ*, *PAPE*, and *BFP* perform the best on

Table 1 Summary of the 10 Algorithms Tested

Algorithm	Implementation description	Complexity*	References
Label-correcting codes			
*GOR*1	Topological ordering with distance updates	$O(nm)$	Goldberg and Radzik (1993)
BFP	Queue implementation with parent checking	$O(nm)$	Cherkassky, Goldberg, and Radzik (1996)
THRESH	Hybrid of Bellman-Ford and Dijkstra's algorithm	$O(nm)$	Glover, Glover, and Klingman (1984)
PAPE	Maintaining candidate lists as a stack and a queue	$O(n2^n)$	Pape (1974)
TWOQ	Maintaining candidate lists as two queues	$O(n^2m)$	Pallottino (1984)
Label-setting codes	Dijkstra's algorithm		
DIKH	k-ary heap implementation with $k = 3$	$O(m \log n)$	Johnson (1972)
DIKBD	Double buckets implementation	$O(m + n(\Delta + C/\Delta))$	Cherkassky, Goldberg, and Radzik (1996)
DIKR	Radix-heap implementation	$O(m + n \log C)$	Ahuja et al. (1990)
DIKBA	Approximate buckets implementation	$O(m\Delta + n(\Delta + C/\Delta))$	Cherkassky, Goldberg, and Radzik (1996)

*$C = \max_{(i,j)\in A}\{|c_{ij}|\}$; Δ is a fixed parameter.

this SPGRID-SQ family. Dijkstra-based codes perform relatively worse for smaller networks. *DIKBD* performs slightly worse than *THRESH*, but is the fastest of Dijkstra's codes. *DLU* performs similarly to *GOR*1, but is faster than *DIKH* and *DIKR* most of the time.

Table 3 shows that label-correcting codes *TWOQ*, *PAPE*, and *BFP* perform the best on this SPGRID-WL family. *THRESH* is slightly worse than *BFP*. *DIKBD* is the fastest Dijkstra's code, but *DIKBA* catches up for larger LONG cases. *DLU* is faster in the WIDE cases, and is slightly better than *GOR*1. *DLU* also beats *DIKH* and *DIKR*. *DIKR* performs the worst for the WIDE cases, but *DIKH* performs the worst for the LONG cases.

On random grid networks with dense demands *DLU* is not the fastest MPSP algorithm. However, on real-life airline networks *DLU* performs much better.

To determine how *DLU* performs when solving MPSP problems on real transportation networks, we used data based on annual worldwide-flight schedules to create networks for six international airlines (denoted as A_1, A_2, A_3, A_4, A_5, and A_6). We also create networks for six geographic regions (denoted as R_1, R_2, R_3, R_4, R_5, and R_6), incorporating all flights over all airlines within each region. The number of nodes and arcs for these 12 graphs are listed in Table 4. The networks are sparse because their average degree ($|N|/|A|$) is between 3 and 6. For each graph, we randomly generate two sets of requested OD pairs which contain $|Q_t| = |Q_s| = 100\%|N|$ and $|Q_t| = |Q_s| = 50\%|N|$ distinct destinations, respectively. In other words, to solve a $|Q_t| = |Q_s| = 50\%|N|$ MPSP problem, all SSSP algorithms have to perform $50\%|N|$ shortest path tree computations.

We use both running time and number of triple comparisons to measure the algorithmic efficiency. We state the normalized results for running time (see Tables 5 and 6) and the number of triple comparisons (see Tables 7 and 8) on these 12 graphs.

These computational results show that our algorithm *DLU* beats all of the other algorithms (the Floyd-Warshall algorithm (*FW*), label-correcting algorithms (*GOR*1, *BFP*, *PAPE*, and *TWOQ*), label-setting algorithms (*DIKH*, *DIKBD*, *DIKR*, and *DIKBA*), and their hybrid (*THRESH*)) when solving MPSP problems on real-flight networks. *DLU* also performs the least number of triple comparisons in all the cases tested. Because the SSSP algorithms we imported from Cherkassky, Goldberg, and Radzik (1996) are considered to be very efficient, the computational results suggest that *DLU* is efficient in solving real-world MPSP problems.

4. Conclusions

In this paper, we propose a new algorithm called *DLU* that is suitable for solving MPSP problems.Although

Table 2 Normalized Running Time for a $|Q_t| = 75\%|N|$ MPSP Problem on SPGRID-SQ

Grid/deg	*DLU*	*GOR*1	*BFP*	*THRESH*	*PAPE*	*TWOQ*	*DIKH*	*DIKBD*	*DIKR*	*DIKBA*
$10 \times 10/3$	6.20	5.50	1.30	3.30	1.10	1.00	7.30	6.10	10.90	26.10
$20 \times 20/3$	3.73	5.04	1.18	2.58	1.08	1.00	8.01	4.40	9.43	11.84
$30 \times 30/3$	3.79	4.27	1.13	2.01	1.05	1.00	6.82	3.27	7.15	6.52
$40 \times 40/3$	10.74	4.56	1.12	2.15	1.05	1.00	7.18	3.24	7.14	5.22
$50 \times 50/3$	5.05	5.11	1.13	2.04	1.04	1.00	7.27	3.09	6.81	4.56
$60 \times 60/3$	5.07	5.24	1.13	1.95	1.03	1.00	6.92	2.91	6.37	3.88
$70 \times 70/3$	5.84	4.67	1.14	2.13	1.05	1.00	7.35	3.04	6.62	3.73
$80 \times 80/3$	9.91	5.65	1.14	2.14	1.05	1.00	7.54	3.05	6.55	3.54

Table 3 **Normalized Running Time for a $|Q_t| = 75\%|N|$ MPSP Problem on SPGRID-WL**

Grid/deg	*DLU*	*GOR*1	*BFP*	*THRESH*	*PAPE*	*TWOQ*	*DIKH*	*DIKBD*	*DIKR*	*DIKBA*
$16 \times 64/3$	3.50	4.38	1.09	2.05	1.05	1.00	6.22	3.88	8.42	9.69
$16 \times 128/3$	3.59	5.15	1.12	1.99	1.05	1.00	5.82	3.61	8.20	8.12
$16 \times 256/3$	4.46	4.90	1.10	2.03	1.05	1.00	5.76	3.70	8.49	7.83
$16 \times 512/3$	3.18	5.60	1.09	2.04	1.03	1.00	5.41	3.53	8.33	7.24
$64 \times 16/3$	3.69	4.81	1.13	2.22	1.06	1.00	8.39	3.25	6.78	5.60
$128 \times 16/3$	3.76	4.97	1.15	2.02	1.03	1.00	8.46	2.90	5.78	3.68
$256 \times 16/3$	4.85	4.86	1.12	1.93	1.03	1.00	9.61	2.98	5.77	3.27
$512 \times 16/3$	5.00	5.06	1.15	1.80	1.04	1.00	10.62	2.97	5.64	3.02

its worst-case complexity $O(n^3)$ is equivalent to other algebraic APSP algorithms, for instance, Floyd-Warshall (Floyd 1962, Warshall 1962) and Carré's (1969, 1971) algorithms, *DLU* can, in practice, avoid significant computational work in solving MPSP problems. Algorithm *DLU* can deal with graphs containing negative arc lengths and detect negative cycles earlier than the Floyd-Warshall algorithm. It also saves storage and computational work for problems with special structures such as undirected or acyclic graphs.

DLU attacks each requested OD pair individually, so it is more suitable for problems with a scattered OD distribution. In extreme cases, it is especially efficient for solving MPSP instances in which there are exactly n OD pairs (s_i, t_i) corresponding to a matching in $N \times N$. That is, each node appears exactly once in each of the source and sink node sets, but not in the same OD pair. Such an MPSP problem requires as much work as an APSP problem for most shortest path algorithms known nowadays, even though only n OD pairs are requested.

When solving MPSP problems, *DLU* may be sensitive to the distribution of requested OD pairs and the node ordering. In particular, when the requested OD pairs are closely distributed in the right lower part of the $n \times n$ OD matrix, Algorithm *DLU* can terminate much earlier. On the other hand, scattered OD pairs might make the algorithm less efficient, although it will still be better than other APSP algorithms. A bad node ordering may incur many "fill-ins." These fill-ins make the modified graph denser, which in turn will require more triple comparisons when applying our algorithm. These difficulties may be resolved by reordering the node indices so that the requested OD pairs are grouped in a favored distribution or the creation of fill-in arcs is decreased.

Because *DLU* can often terminate much sooner, given a favorable node ordering, the algorithm can be especially beneficial as a subroutine in certain iterative algorithms. For a graph with fixed topology, where shortest paths of a fixed set of OD pairs must be repeatedly computed with different numerical values of arc lengths, *DLU* is especially beneficial because we may do a preprocessing step to select a node ordering that favors *DLU*. A speedup is then obtained at every repetition of MPSP, even if the arc costs change. Such problems appear often in real-world applications. For example, when solving the origin-destination multicommodity network flow problem (ODMCNF) using Dantzig-Wolfe decomposition and column generation (Barnhart et al. 1995), we generate columns by solving sequences of shortest path problems between specific OD pairs. The arc costs change in each stage, but both the topology and OD pairs are fixed. Another example is in the computation of parametric shortest paths where arc length is a linear function of some parameter. We need to solve for shortest paths repeatedly on the same graph (with different arc costs) to determine the critical value of the parameter.

We have shown the superiority of Algorithm *DLU* over the other APSP and SSSP algorithms for solving the MPSP problem. Computational results show that *DLU* performs better than SSSP and APSP algorithms on real-world flight networks. A more thorough computational experiment that compares the empirical efficiency of *DLU* with many modern SSSP and APSP algorithms will be conducted in our forthcoming paper (Wang 2005). In that paper, we will also address sparsity. Like all other algebraic algorithms in the literature, *DLU* requires $O(n^2)$ storage, which makes it suitable for use on dense graphs. We have developed techniques for sparse implementation that avoid nontrivial triple comparisons and lead to promising computational results (see Wang 2005), but they come with the price of extra storage for the adjacency data structures.

Table 4 **Size of 12 Flight Networks**

	A_1	A_2	A_3	A_4	A_5	A_6	R_1	R_2	R_3	R_4	R_5	R_6
$\|N\|$	175	229	233	236	251	330	134	189	363	678	705	1,093
$\|A\|$	748	1,120	811	829	1,295	985	800	779	1,727	6,309	6,497	8,692

Table 5 Normalized Running Time for a $|Q_t| = 100\%|N|$ MPSP Problem on Flight Networks

Network	*DLU*	*FW*	*GOR*1	*BFP*	*THRESH*	*PAPE*	*TWOQ*	*DIKH*	*DIKBD*	*DIKR*	*DIKBA*
A_1	1	11.83	7.17	3.28	6.56	3.06	3.22	18.06	11.72	24.67	11.17
A_2	1	11.42	6.37	2.70	4.70	2.65	2.70	14.12	9.05	18.05	7.81
A_3	1	12.81	6.75	3.06	5.58	2.86	3.06	17.47	10.94	22.61	9.72
A_4	1	7.76	7.09	3.06	5.88	3.00	3.24	17.68	11.12	23.44	10.50
A_5	1	14.40	4.86	2.00	3.31	1.80	1.88	8.76	5.92	11.10	5.61
A_6	1	7.60	4.10	1.81	3.33	1.67	1.77	9.71	5.95	12.59	5.55
R_1	1	6.20	3.20	1.40	2.00	1.00	1.20	3.80	3.00	5.40	3.60
R_2	1	11.60	5.20	2.40	3.20	1.80	1.80	7.20	5.20	10.00	5.60
R_3	1	22.53	6.35	2.71	3.65	2.47	2.65	8.47	5.76	11.53	5.47
R_4	1	18.71	2.80	1.21	1.37	1.08	1.14	3.35	2.15	3.80	1.83
R_5	1	18.53	2.63	1.17	1.29	1.07	1.11	3.07	2.09	3.74	1.85
R_6	1	26.81	3.54	1.62	1.62	1.41	1.38	3.74	2.38	4.43	2.09

Table 6 Normalized Running Time for a $|Q_t| = 50\%|N|$ MPSP Problem on Flight Networks

Network	*DLU*	*FW*	*GOR*1	*BFP*	*THRESH*	*PAPE*	*TWOQ*	*DIKH*	*DIKBD*	*DIKR*	*DIKBA*
A_1	1	11.21	5.37	2.37	4.68	2.21	2.53	13.74	8.89	18.32	8.53
A_2	1	14.73	6.55	2.79	4.64	2.67	2.67	13.82	8.91	17.79	7.76
A_3	1	13.91	6.03	2.55	4.55	2.33	2.48	14.48	9.03	18.67	8.33
A_4	1	12.48	8.43	3.86	6.95	3.57	4.05	21.24	13.29	28.29	12.24
A_5	1	20.16	4.95	2.05	3.34	1.79	1.90	8.51	5.93	11.38	5.10
A_6	1	10.25	3.94	1.78	3.22	1.71	1.78	9.43	5.88	12.68	4.88
R_1	1	15.50	6.00	2.00	3.50	2.00	2.00	7.00	5.50	10.00	4.50
R_2	1	27.50	10.00	4.00	6.50	3.50	3.50	12.50	11.00	19.50	10.00
R_3	1	20.83	5.28	1.89	2.33	1.72	1.83	6.00	4.39	8.22	3.44
R_4	1	25.32	2.65	1.24	1.39	1.11	1.15	3.39	2.17	3.89	1.93
R_5	1	25.50	2.77	1.26	1.40	1.11	1.16	3.28	2.19	3.85	1.82
R_6	1	33.92	3.31	1.53	1.48	1.31	1.37	3.68	2.35	4.20	1.82

Table 7 Normalized Number of Triple Comparisons for a $|Q_t| = 100\%|N|$ MPSP Problem on Flight Networks

Network	*DLU*	*FW*	*GOR*1	*BFP*	*THRESH*	*PAPE*	*TWOQ*	*DIKH*	*DIKBD*	*DIKR*	*DIKBA*
A_1	1	6.39	14.59	24.64	24.41	24.68	24.68	23.71	23.71	23.71	23.71
A_2	1	6.38	17.80	24.58	23.49	25.06	25.06	22.89	22.89	22.89	22.89
A_3	1	7.24	15.04	24.48	23.43	24.46	24.46	22.55	22.55	22.55	22.55
A_4	1	26.18	50.83	83.41	77.87	87.63	87.63	75.33	75.33	75.33	75.33
A_5	1	2.72	8.55	10.37	10.03	10.35	10.35	9.83	9.83	9.83	9.83
A_6	1	7.88	11.42	21.10	20.35	21.12	21.11	20.06	20.06	20.06	20.06
R_1	1	1.89	4.76	5.26	4.79	5.21	5.21	4.76	4.76	4.76	4.76
R_2	1	3.47	7.05	7.83	6.78	7.69	7.68	6.76	6.76	6.76	6.76
R_3	1	3.76	11.95	12.93	10.24	12.75	12.75	10.11	10.11	10.11	10.11
R_4	1	1.38	3.87	4.30	3.37	4.26	4.25	3.26	3.26	3.26	3.26
R_5	1	1.16	4.38	4.98	4.01	5.01	4.98	3.78	3.78	3.78	3.78
R_6	1	1.67	4.92	5.55	3.98	5.52	5.38	3.77	3.77	3.77	3.77

Table 8 Normalized Number of Triple Comparisons for a $|Q_t| = 50\%|N|$ MPSP Problem on Flight Networks

Network	*DLU*	*FW*	*GOR*1	*BFP*	*THRESH*	*PAPE*	*TWOQ*	*DIKH*	*DIKBD*	*DIKR*	*DIKBA*
A_1	1	7.92	13.55	22.88	22.63	22.92	22.92	21.99	21.99	21.99	21.99
A_2	1	8.20	17.13	23.83	22.68	24.35	24.35	22.09	22.09	22.09	22.09
A_3	1	9.63	14.86	24.41	23.39	24.41	24.41	22.51	22.51	22.51	22.51
A_4	1	33.00	48.07	79.02	73.65	83.11	83.11	71.21	71.21	71.21	71.21
A_5	1	3.46	8.14	9.85	9.52	9.83	9.82	9.34	9.34	9.34	9.34
A_6	1	10.70	11.35	21.05	20.32	21.08	21.07	20.02	20.02	20.02	20.02
R_1	1	2.47	4.70	5.18	4.71	5.13	5.13	4.69	4.69	4.69	4.69
R_2	1	4.57	6.99	7.74	6.72	7.61	7.61	6.69	6.69	6.69	6.69
R_3	1	4.81	11.47	12.44	9.83	12.31	12.32	9.71	9.71	9.71	9.71
R_4	1	1.78	3.71	4.14	3.25	4.10	4.09	3.15	3.15	3.15	3.15
R_5	1	1.48	4.15	4.72	3.84	4.77	4.75	3.61	3.61	3.61	3.61
R_6	1	2.12	4.66	5.27	3.79	5.22	5.10	3.59	3.59	3.59	3.59

Acknowledgments

The authors thank the associate editor and three anonymous referees for their helpful comments and suggestions, as well as B. V. Cherkassky, A. V. Goldberg, and T. Radzik for making their SSSP C codes available. I-Lin Wang was partially supported by the National Science Council of Taiwan under Grant NSC92-2213-E006-094.

References

Aho, A. V., J. E. Hopcroft, J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA.

Ahuja, R. K., T. L. Magnanti, J. B. Orlin. 1993. *Network Flows: Theory, Algorithms and Applications*. Prentice Hall, Englewood Cliffs, NJ.

Ahuja, R. K., K. Mehlhorn, J. B. Orlin, R. E. Tarjan. 1990. Faster algorithms for the shortest path problem. *J. ACM* **37**(2) 213–223.

Backhouse, R. C., B. A. Carré. 1975. Regular algebra applied to path finding problems. *J. Inst. Math. Appl.* **15**(2) 161–186.

Barnhart, C., E. L. Johnson, C. Hane, G. Sigismondi. 1995. An alternative formulation and solution strategy for multicommodity network flow problems. *Telecomm. Systems* **3** 239–258.

Bellman, R. E. 1958. On a routing problem. *Quart. Appl. Math.* **16** 87–90.

Bertsekas, D. P., S. Pallottino, M. G. Scutellà. 1995. Polynomial auction algorithms for shortest paths. *Comput. Optim. Appl.* **4**(2) 99–125.

Carré, B. A. 1969. A matrix factorization method for finding optimal paths through networks. *IEE Conf. Publ. (Comput.-Aided Design)*, Vol. 51, 388–397.

Carré, B. A. 1971. An algebra for network routing problems. *J. Inst. Math. Appl.* **7** 273–294.

Cherkassky, B. V., A. V. Goldberg, T. Radzik. 1996. Shortest paths algorithms: Theory and experimental evaluation. *Math. Programming* **73**(2) 129–174.

Dantzig, G. B. 1960. On the shortest route through a network. *Management Sci.* **6** 187–190.

Dial, R. 1965. Algorithm 360 shortest path forest with topological ordering. *Comm. ACM* **12** 632–633.

Dijkstra, E. W. 1959. A note on two problems in connection with graphs. *Numerische Mathematik* **1** 269–271.

Duff, I. S., A. M. Erisman, J. K. Reid. 1989. *Direct Methods for Sparse Matrices*. Oxford University Press, New York.

Floyd, R. W. 1962. Algorithm 97, shortest path. *Comm. ACM* **5** 345.

Ford, L. R., Jr. 1956. *Network Flow Theory*. The RAND Corp., Santa Monica, CA.

Fredman, M. L. 1976. New bounds on the complexity of the shortest path problems. *SIAM J. Comput.* **5**(1) 83–89.

Galil, Z., O. Margalit. 1997. All pairs shortest paths for graphs with small integer length edges. *J. Comput. System Sci.* **54**(2) 243–254.

Glover, F., R. Glover, D. Klingman. 1984. Computational study of an improved shortest path algorithm. *Networks* **14**(1) 25–36.

Goldberg, A. V., T. Radzik. 1993. A heuristic improvement of the Bellman-Ford algorithm. *Appl. Math. Lett.* **6**(3) 3–6.

Goldfarb, D., Z. Jin. 1999. An $o(nm)$-time network simplex algorithm for the shortest path problem. *Oper. Res.* **47**(3) 445–448.

Goldfarb, D., J. Hao, S. R. Kai. 1990. Efficient shortest path simplex algorithms. *Oper. Res.* **38**(4) 624–628.

Hu, T. C. 1968. A decomposition algorithm for shortest paths in a network. *Oper. Res.* **16** 91–102.

Johnson, E. L. 1972. On shortest paths and sorting. *Proc. ACM 25th Annual Conf.*, 510–517.

Markowitz, H. M. 1957. The elimination form of the inverse and its application to linear programming. *Management Sci.* **3** 255–269.

Mills, G. 1966. A decomposition algorithm for the shortest-route problem. *Oper. Res.* **14** 279–291.

Moore, E. F. 1957. The shortest path through a maze. *Proc. Internat. Sympos. Theory Switching, Part II.* The Annals of the Computation Laboratory of Harvard University, Harvard University, Cambridge, MA, 285–292.

Nakamori, M. 1972. A note on the optimality of some all-shortest path algorithms. *J. Oper. Res. Soc. Japan* **15**(4) 201–204.

Pallottino, S. 1984. Shortest-path methods: Complexity, interrelations and new propositions. *Networks* **14** 257–267.

Pallottino, S., M. G. Scutellà. 1997. Dual algorithms for the shortest path tree problem. *Networks* **29**(2) 125–133.

Pape, U. 1974. Implementation and efficiency of Moore algorithms for the shortest root problem. *Math. Programming* **7** 212–222.

Rose, D. J., R. E. Tarjan. 1978. Algorithmic aspects of vertex elimination on directed graphs. *SIAM J. Appl. Math.* **34**(1) 176–197.

Seidel, R. 1995. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. System Sci.* **51**(3) 400–403.

Takaoka, T. 1992. A new upper bound on the complexity of the all pairs shortest path problem. *Inform. Processing Lett.* **43**(4) 195–199.

Wang, I. L. 2005. On implementation of new multiple shortest paths algorithms. Technical Report NCKU-IIM-001, Dept. of Industrial and Information Management, National Cheng Kung University.

Warshall, S. 1962. A theorem on Boolean matrices. *J. ACM* **9** 11–12.

Zwick, U. 1998. All pairs shortest paths in weighted directed graphs—Exact and almost exact algorithms. *Proc. 39th Annual IEEE Sympos. Foundations Comput. Sci.*, Palo Alto, CA, 310–319.