SOLVING MULTIPLE PAIRS SHORTEST PATHS BY LU DECOMPOSITION: NEW ALGORITHM AND COMPUTATIONAL EVALUATION

I-Lin WANG
Assistant Professor
Department of Industrial & Information Management
National Cheng Kung University
No.1, University Rd.
Tainan 70101, Taiwan
Fax: +886-6-2362162
Email: ilinwang@mail.ncku.edu.tw

Abstract: The shortest path problem in real applications usually asks for shortest paths and distances between some specific pairs of origin-destination nodes which we call a multiple pairs shortest path (MPSP) problem. Prior research efforts usually focus on the development and empirical evaluation for single source shortest path (SSSP) algorithms which may do unnecessary computation to solve a MPSP problem. In this paper, we propose a new MPSP algorithm, suggest faster implementations, and provide computational evaluation of our algorithm with ten SSSP algorithms using a real flight network and families of randomly generated grid and acyclic networks. Based on our evaluation, recommended algorithms for computing shortest paths on different networks are identified.

Key Words: shortest path, multiple pairs, algorithm, flight network

1. INTRODUCTION

The Multiple Pairs Shortest Path (MPSP) problem on a digraph $G:=(N,A)$ where N and A are the set of nodes and arcs of G with cardinality $|N|=n$ and $|A|=m$ respectively, is to compute shortest paths for specific origin-destination (OD) pairs. This problem arises very often in telecommunication and transportation networks where the shortest paths have to be computed on the same graph with unchanged topology structure but different arc lengths or for different requested OD pairs. For example, on a daily traffic network if we use the traveling time along an arc to present its arc length, the shortest path will vary in different time during a day even for the same OD pair since the arc length vector varies over time.

Shortest path algorithms in the literature can be classified into 3 groups: (a) graphical traversal algorithms such as label-setting methods (Dijkstra 1959, Dial 1965), label-correcting methods (Ford 1956, Moore 1957, Pape 1974) and their hybrids (Glover et al 1984); (b) linear programming (LP) related methods (Goldfarb et al. 1990, Pallottino and Scutellà 1997); and (c) algebraic methods such as Floyd-Warshall (Floyd 1962, Warshall 1962) and Carré's (1971) algorithms. The first two groups solve the Single Source (or Sink) Shortest Path (SSSP) problem, which computes a shortest path tree for a specific source (or sink) node. Algebraic shortest path algorithms are more suitable for solving the All Pairs Shortest Paths (APSP) problem, which computes shortest paths for all the node pairs.

To the best of our knowledge, no specific algorithm has been designed to solve the MPSP problem in the literature. A SSSP algorithm can be repeatedly applied for different sources (or sinks) to solve a MPSP problem. The LP-based re-optimization algorithms (Florian et al 1981, Nguyan et al. 2001) take advantage of previous optimal shortest paths. However, more recent computational experiments (Burton 1993) indicate these re-optimization algorithms are still inferior to the repeated SSSP algorithms. Obviously, the MPSP problem can also be solved by applying an APSP algorithm. Algebraic algorithms usually require more storage ($O(n^2)$ space). Their naive implementations perform less efficiently than repeated SSSP algorithms, especially for sparse graphs. Since Carré's algorithm corresponds to the Gaussian elimination method (Carré 1971), it is more suitable for sparse graphs.

The basic operation in these shortest path algorithms is a *triple comparison* $s \rightarrow k \rightarrow t$ which compares $x_{sk} + x_{kt}$ with x_{st} where x_{ij} denotes temporary distance from i to j by a shortest path algorithm. In other words, each step in each shortest path algorithm either performs a triple comparison, or identifies a specific triple ordering (s,k,t) for triple comparisons. For example, in Dijkstra's algorithm, besides the distance label updating operations (which correspond to the triple comparisons), a node is selected to scan in a greedy fashion to avoid

unnecessary triple comparisons. Likewise, Floyd-Warshall's algorithm performs triple comparison in an order determined by three nested for loops for s, k and t. Thus an algebraic algorithm may become competitive, if a suitable sequence of triple ordering can be applied, especially for computing shortest paths on a graph with unchanged topology.

To avoid unnecessary triple comparisons in Carré's algorithm, Goto et al. (1976) record all the nontrivial triple comparisons in the LU decomposition, forward elimination and backward substitution phases, and then generate an ad hoc APSP code. Their method only stores $O(m)$ data structures, same as other graphical SSSP algorithms but better than $O(n^2)$ as required in general algebraic algorithms. Yet this comes with the price of storing the long codes of nontrivial triple comparisons (i.e. code generation), and requires more total hardware storage. Thus their implementation is not practical.

Inspired by Carré's algorithm, this paper proposes a new algebraic algorithm (SLU, where S stands for sparse and LU for LU decomposition) designed specifically for solving the MPSP problems. SLU can be viewed as a more efficient sparse implementation of Carré's algorithm. SLU resolves the need of a huge storage quota by the code generation. Furthermore, SLU decomposes and truncates Carré's algorithm according to the indices of the distinct origins/destinations of the requested OD pairs so that many unnecessary triple operations can be avoided. SLU can not only deal with negative arc lengths, but also detect negative cycles. To speed up SLU, we propose three efficient graphical implementations using techniques of efficient graphical SSSP algorithms such as the heap implementation or bucket implementation (Dial 1965) of Dijkstra's algorithm. We compare our algorithms with state-of-the-art SSSP codes written by Cherkassky et al. (1996) on several artificial networks and a real flight network.

The remainder of this paper is organized as follows:. Section 2 introduces our new algorithm together with their efficient implementations. Section 3 presents the settings and results of our computational experiments. Section 4 concludes our work and proposes future research.

2. NEW MPSP ALGORITHMS WITH FASTER IMPLEMENTATIONS

2.1 Preliminaries

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 $(i, j) \notin A$. The $n \times n$ distance matrix $[x_{ij}]$ is initialized as $[c_{ij}]$ to record the length of a shortest path from i to j. Let $[succ_{ij}]$ denote a $n \times n$ successor matrix in which $succ_{ij}$, initialized as j, represents the node that immediately follows i in a shortest path from i to j. Suppose $i \rightarrow k_1 \rightarrow k_2 \rightarrow \ldots \rightarrow k_r \rightarrow j$ is a shortest path in G from i to j, then $k_1 = succ_{ij}$,

$k_2 = succ_{k_1 j}$,…, $k_r = succ_{k_{r-1} j}$, and $j = succ_{k_r j}$. Let x_{ij}^* and $succ_{ij}^*$ denote the shortest distance and successor from i to j in G.

We say that node i is higher (lower) than node j if the index $i>j$ ($i<j$). A node i in a node set *LIST* is said to be the highest (lowest) node in *LIST* if $i \geq k$ ($i \leq k$) for each 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) $H(min\{s,t\} \cup [max\{s,t\},n])$. Any shortest path in G from s to t is the shortest paths among these three induced subgraphs. This paper gives an algebraic algorithm that systematically calculates shortest paths for these cases.

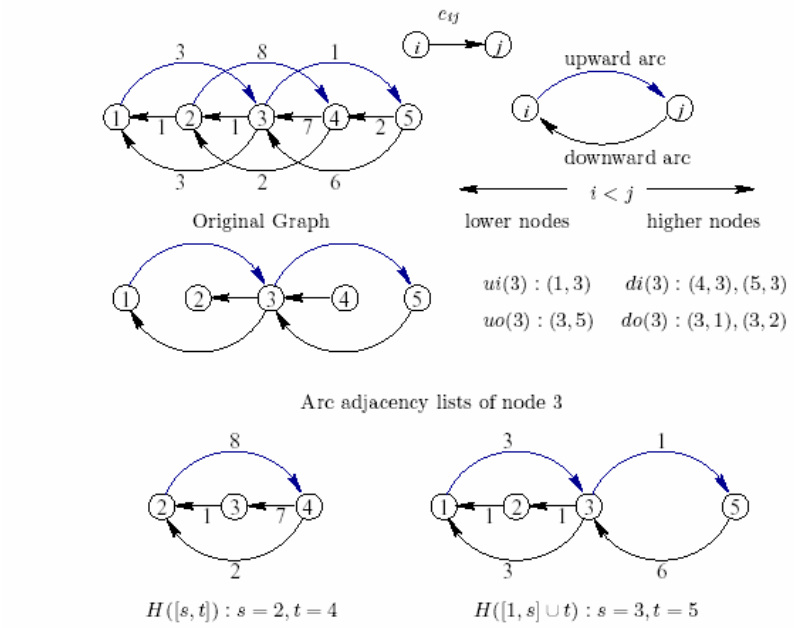

Figure 1. Illustration of Arc Adjacency Lists, and Subgraphs $H([2,4])$, $H([1,3] \cup 5)$

We define the up-inward (down-inward) arc adjacency list denoted $ui(i)$ ($di(i)$) of a node i to be an array that contains all arcs which point upwards (downwards) into node i. Similarly, we define the up-outward (down-outward) arc adjacency list denoted $uo(i)$ ($do(i)$) of a node i to be an array of all arcs pointing upwards (downwards) out of node i.

The underlying ideas of our algorithm are as follows: Suppose the shortest path from s to t contains more than one intermediate node and let r be the highest intermediate node in that shortest path. There are only three cases: (1) $r<min\{s,t\}$ (2) $min\{s,t\}<r<max\{s,t\}$ and (3)

$r>max\{s,t\}$. The first two cases correspond to a shortest path in $H([1,max\{s,t\}])$, and the last case corresponds to a shortest path in $H([1,r])$ for $r>max\{s,t\}$. Including the case where the shortest path is a single arc, our algorithm systematically calculate shortest paths for these cases. When solving the APSP problem on a complete graph, our algorithm have the same number of operations as the Floyd-Warshall algorithm does in the worst case. For general MPSP problems, our algorithms beat the Floyd-Warshall algorithm.

2.2 Algorithm SLU

Algorithm 1 SLU(Q := $\{(s_i, t_i) : i = 1, ..., q\}$)

begin

$Preprocess_S$;

G_LU_S;

for each distinct destination node $\hat{t}_i$ ***do***

reset $label(k) = 0,\ d_n(k) = M,\ succ_n(k) = 0\ \forall k \in N \setminus \{\hat{t}_i\};\ d_n(\hat{t}_i) = 0$

$G\_Forward(\hat{t}_i)$;

$G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$;

end

Algorithm SLU can be viewed as an efficient sparse implementation of Carré's algorithm which resembles Gaussian elimination. The algorithm first performs a preprocessing procedure, $Preprocess_S$, to determine a good node ordering that reduces the fill-ins created by LU decomposition. $Preprocess_S$ performs a symbolic LU decomposition just like G_LU_S, then determines all the fill-ins which then are used to construct an augmented graph G'.

Procedure Preprocess$_S$

begin

Decide a node ordering for each node;

Symbolic execution of procedure G_LU_S to determine the arc adjacency list

$di(k)$, $uo(k)$, and $ui(k)$, for each node k of the augmented graph G';

for each distinct destination node $\hat{t}_i$ ***do***

if shortest paths need to be traced ***then*** set $s_{\hat{t}_i} := 1$

else set $s_{\hat{t}_i} := \min_i \{s_i : (s_i, \hat{t}_i) \in Q\}$

Initialize: $\forall (s,t) \in A'$, ***if*** $(s,t) \in A$ with index a_{st} ***then***

$c(a_{st}) := c_{st};\ succ(a_{st}) := t$

else $c(a_{st}) := M;\ succ(a_{st}) := 0$

end

Based on the topology of the augmented graph G', procedure G_LU_S graphically performs the LU decomposition. Then, just like Gaussian elimination which inverts a square matrix column-wise, SLU also computes the shortest path column by column. For each column, SLU performs two acyclic operations: $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ which resembles the forward elimination and backward substitution. In particular, for nodes k=1,..,(n-2),

G_LU_S scans each arc of $di(k)$ (with tail node s) and each arc of $uo(k)$ (with head node t) and updates the length and successor of arc (s,t) in G'.

Procedure G_LU$_S$
begin
  for $k = 1$ to $n-2$ ***do***
    for each arc $(s,k) \in di(k)$ with index a_{sk} ***do***
      for each arc $(k,t) \in uo(k)$ with index a_{kt} ***do***
        if $s = t$ and $c(a_{sk}) + c(a_{kt}) < 0$ ***then***
          Found a negative cycle; ***STOP***
        if $s \neq t$ ***then***
          let arc (s,t) have index a_{st} ;
          if $c(a_{st}) > c(a_{sk}) + c(a_{kt})$
            $c(a_{st}) := c(a_{sk}) + c(a_{kt})$; $succ(a_{st}) := succ(a_{sk})$;
end

Procedure G_Forward($\hat{t}_i$)
begin
  initialize $d_n(\hat{t}_i) := 0$, $d_n(k) := M$ $\forall k \neq t_i$
  put node $\hat{t}_i$ in $LIST$
  while $LIST$ is not empty ***do***
    remove the lowest node k in $LIST$
    $label(k) = 1$
    for each arc $(s,k) \in di(k)$ with index a_{sk} ***do***
      if $s \notin LIST$, put s into $LIST$
      if $d_n(s) > d_n(k) + c(a_{sk})$ ***then***
        $d_n(s) := d_n(k) + c(a_{sk})$; $succ_n(s) := succ(a_{sk})$
end

Procedure G_Backward($s_{\hat{t}_i}, \hat{t}_i$)
begin
  put all the labeled nodes with index higher than or equal to $s_{\hat{t}_i}$ into $LIST$
  while $LIST$ is not empty ***do***
    remove the highest node k in $LIST$
    $label(k) = 1$
    for each arc $(s,k) \in ui(k)$ with index a_{sk} ***do***
      if $s \geq s_{\hat{t}_i}$ and $s \neq \hat{t}_i$ then
        if $s \notin LIST$, put s into $LIST$; $label(s) = 1$
        if $d_n(s) > d_n(k) + c(a_{sk})$ ***then***
          $d_n(s) := d_n(k) + c(a_{sk})$; $succ_n(s) := succ(a_{sk})$
end

$G\_Forward(\hat{t}_i)$ can be viewed as computing shortest path lengths from each node $s > \hat{t}_i$ to node $\hat{t}_i$ on the acyclic induced subgraph G'_L. Based on the distance label computed by $G\_Forward(\hat{t}_i)$, for each node $s > \hat{t}_i$, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ computes shortest distance lengths in G'_U for nodes $s = (n-1), \ldots, s_{\hat{t}_i}$. Unlike other shortest path algorithms that work

on the original graph G, algorithm SLU works on the augmented graph G'. The sparser the augmented graph G' is, the more efficient SLU becomes. After conducting $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, we have computed the shortest distance $d^*_{\hat{t}_i}(s)$ for every node pair $(s, \hat{t}_i)$ satisfying $s \geq s_{\hat{t}_i}$. Therefore after $\hat{q}$ iterations of $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ for each distinct destination node $\hat{t}_i$, algorithm SLU will give the shortest distance $d^*_{t_i}(s_i)$ for all the requested OD pairs (s_i, t_i) in Q.

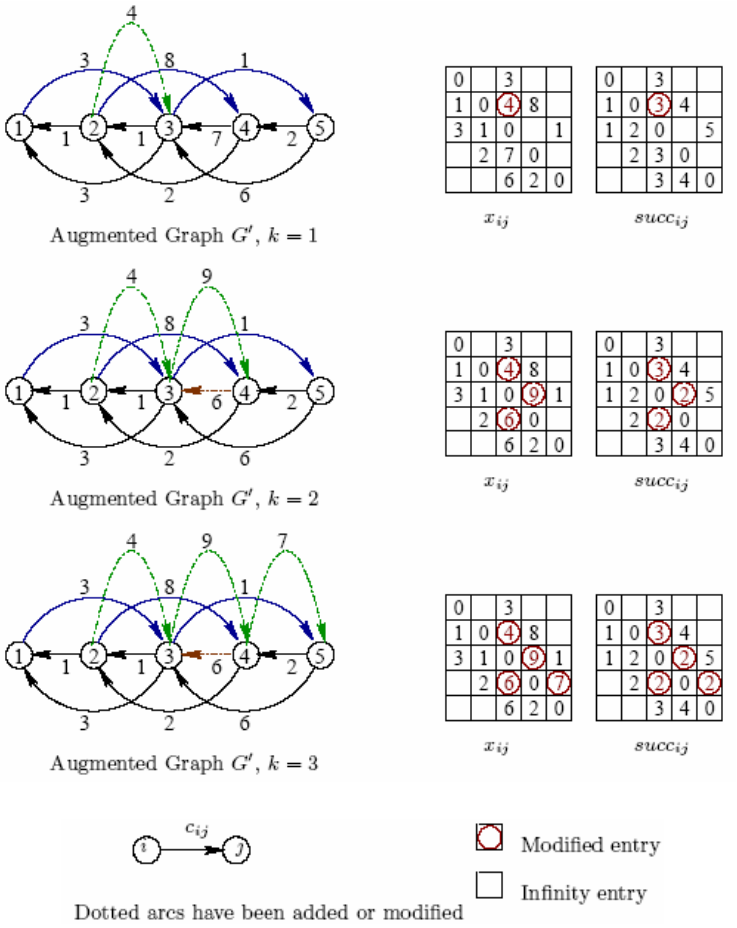

Figure 2. Illustration of Procedure G_LU_S on a Small Example in Figure 1.

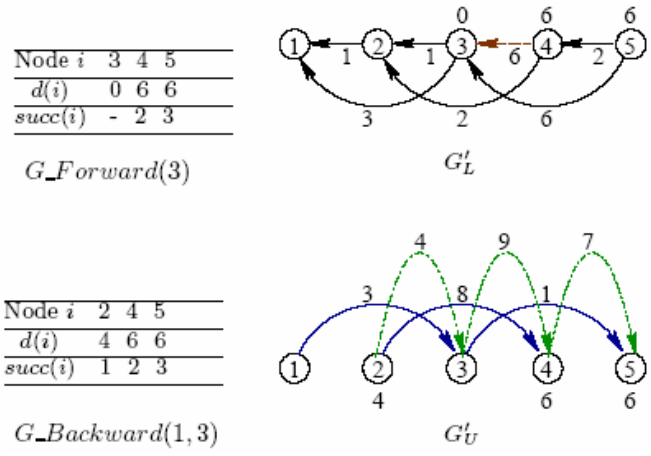

Figure 3. An Example of All Shortest Paths to Node 3

To better understand SLU, we illustrate the operations of its procedures in Figure 2 and Figure 3. Note that when we solve shortest paths on the same graph, we only need to perform the $O(n^3)$ preprocessing once to obtain a good node ordering. Thus when MPSP have to be computed repeatedly on the same graph, the time spent on preprocessing is not the bottleneck of SLU. After the node ordering and topology of *G'* have been determined, all the remaining procedures perform triple comparisons on *G'* and the complexity thus depends on the size of *G'* which should be strictly less than $O(n^3)$ and are case dependent.

Although SLU can take advantages when solving MPSP problems, it still does some unnecessary computations. In particular, to obtain shortest path lengths for OD pair (*s*,*t*), it will compute the shortest paths for OD pairs (*i*,*t*) for each $i \geq s$, even if many of these entries are not requested. Also, the correctness of SLU depends on the sequence of triple comparisons conducted, instead of the sequence of the nodes on the path. Therefore, it computes the shortest path length without tracing the entire shortest path as required in conventional SSSP algorithms. In the case when we have to trace the entire shortest path, SLU must solve the entire column, that is, to set $s_{t_i} := 1$, which solves the ALL-1 shortest path tree as conventional SSSP algorithms do.

2.3 Efficient implementations for SLU

To speed up SLU, we exploit the techniques developed in efficient SSSP algorithms. In preprocessing, we try to exploit techniques in minimizing fill-ins for Gaussian elimination so that the augmented graph *G'* becomes sparser which makes the other procedures faster.

Computing the optimal ordering that minimizes the fill-ins is NP-complete (Rose and Tarjan 1978). Many ordering techniques need to compute degree production, $i(k) \cdot o(k)$ for each node *k*, where $i(k)$ ($o(k)$) denotes the in-degree (out-degree) of node *k*. The ordering techniques that minimize fill-ins in the literature of solving systems of linear equations are applicable here. We have implemented the following ordering techniques: (1) Natural ordering (NAT) (2) Dynamic Markowitz (DM, see Markowitz 1957), (3) Dynamic Markowitz with tie-breaking (DMT, see Duff et al. 1989), (4) Static Markowitz (SM, see Markowitz 1957) (5) METISNodeND (MNDn, see Karypis and Kumar 1999) (6) METISEdgeND (MNDe, see Karypis and Kumar 1999) (7) Multiple Minimum Degree on C^TC (MMDm, see Liu 1985; where $C=[c_{ij}]$) (8) Multiple Minimum Degree on $C^T + C$ (MMTa, see Liu 1985) (9) Approximate Minimum Degree Ordering (AMD, see Davis et al. 2000).

Table 1. Fill-ins by Different Pivot Rules for an Asia-Pacific Flight Network (AP-NET)

Pivoting rule	**NAT**	**DM**	**DMT**	**SM**	**MNDn**	**MNDe**	**MMDm**	**MMTa**	**AMD**
# fill-ins	1084	153	153	170	193	210	430	341	2172

Table 1 shows the result of the fill-ins created by different pivot rules for a flight network. In our tests, usually the dynamic Markowitz rule with tie-breaking (DMT) produces the fewest fill-ins. The pure Dynamic Markowitz rule (DM) usually performs as well as DMT. Since the time to compute a good ordering is not a major concern of our research, the dynamic Markowitz rules fit our needs better since they produce better orderings in reasonable time

G_LU_S is the procedure to construct the augmented graph *G'* using arc adjacency lists *di*(*k*) and *uo*(*k*) for $k = 1 \ldots n-1$. In particular, when we scan node k, we scan each arc (i,k) in $di(k)$ and each arc (k,j) in $uo(k)$, then check whether the length of arc (i,j) needs to be modified or not. Thus we need fast access to the index of each arc (i,j) in *G'*. We store an $n \times n$ matrix [a_{ij}], to record all the arc index of the augmented graph so that we can access an arc very quickly. This implementation of G_LU_S takes $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|))$ time.

For each node *k*, *G_Forward* only requires *di*(*k*) storage, and *G_Backward* requires *ui*(*k*) storage. We use an indicator function $label(k)$ for each node *k*. $label(k) = 1$ means node *k* is labeled and 0 otherwise. When we scan a node k in $G\_Forward(\hat{t}_i)$ (or $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$), it is a triple comparison $s \to k \to \hat{t}_i$ for arcs (*s*,*k*) in *di*(*k*) (or *ui*(*k*)). When we scan an arc (*s*,*k*), we mark node *s* as labeled. *G_Forward* and *G_Backward* conduct very similar operations: to select a lowest (or highest) node *k* from *LIST*, scan and label the tails of arcs in *di*(*k*) (or *ui*(*k*)), and then select the next lowest (or highest) node to scan.

Both of these two procedures conduct a sorting process to select the lowest (or highest) node from *LIST* to scan. We propose three implementations. The bucket implementation (SLU1) uses $label(k)$ to check whether a node is labeled or not, which is similar to Dial's implementation (Dial 1965) of Dijkstra's algorithm. The single-heap implementation (SLU2) is to maintain a heap that stores nodes in ascending order in *G_Forward*, and stores nodes in descending order in *G_Backward*. The double-heap implementation (SLU3) maintains two heaps: a min-heap for *G_Forward* and a max-heap for *G_Backward*. In our tests, SLU1 and SLU3 run faster than the SLU2 (see Table 2 and Table 3), but whether SLU1 is faster than SLU3 depends on the platforms and the compilers.

3. COMPUTATIONAL EVALUATION AND RESULTS

To compare with other SSSP algorithms, we modify the label-correcting, label-setting, and topological ordering codes written by Cherkassky et al. (1996) for solving the MPSP problems. In particular, we modified five label-correcting SSSP codes: GOR1, BFP, THRESH, PAPE, and TWOQ written by Cherkassky et al. (1996) for comparison because of their better performance than other implementations of label-correcting algorithms. They differ with each other in the order of node selection for scanning. In particular, GOR1,

proposed by Goldberg and Radzik (1993), does a topological ordering on the candidate nodes so that the node with more descendants will be scanned earlier. BFP is a modified Bellman-Ford algorithm (Bellman 1958) which scans a node only if its parent is not in the queue. THRESH is the threshold algorithm by Glover et al. (1984). PAPE is a dequeue implementation (Pape 1974) which maintains two candidate lists: a stack and a queue. PAPE has been shown to have an exponential time bound (Kershenbaum 1981), but is practically efficient in most real-world networks. TWOQ by Pallottino (1984) is a similar algorithm which maintains candidate lists by two queues.

We also modify four label-setting codes (DIKH, DIKBD, DIKR, and DIKBA) which are variants of Dijkstra's algorithm and written by Cherkassky et al. (1996). They differ with each other in the way of selecting the node with smallest distance label. In particular, DIKH is the most common binary heap implementation. DIKBD, DIKR, and DIKBA can be viewed as variants of Dial's bucket implementation (Dial 1965). DIKBD is a double bucket implementation, DIKBA is an approximate bucket implementation, and DIKR is the radix-heap implementation. See Cherkassky et al. (1996) for more detailed introduction on these implementations and their complexities.

For acyclic networks, we compare our codes with an efficient acyclic code (ACC) written by Cherkassky et al. (1996). We also implement a graphical Floyd-Warshall (FWG) algorithm which is much faster than its naive algebraic implementation (FWA). All versions of SLU and FW share the same preprocessing procedure which determines a sparse node ordering

First we test our algorithms on a real Asia-Pacific flight network (AP-NET) which contains 112 nodes (48 center nodes and 64 rim nodes) and 1038 arcs, where each node represents a chosen city and each arc represents a chosen flight. Among the 1038 arcs, 480 arcs connect center cities to center cities; 277 arcs connect rim cities to center cities; and 281 arcs connect center cities to rim cities. We use the great circle distance between cities as the arc length.

Table 2. Relative Algorithmic Performance on Solving Four MPSP Problems on AP-NET

OD%	FWG	SLU 1	SLU 2	SLU 3	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
25%	57.09	4.3	10.22	10.52	3.78	1.09	2.04	1.09	1	6	4.35	8.78	10.57
50%	27.52	3.71	9.46	9.12	3.4	1.08	1.96	1.02	1	5.67	4.27	8.35	10.31
75%	18.21	3.44	8.84	8.86	3.33	1.07	1.97	1.05	1	5.67	4.21	8.36	9.52
100%	13.63	3.49	9.12	8.93	3.55	1.08	1.99	1.05	1	5.66	4.43	8.52	10.29

Table 2 shows the normalized running time with respect to the fastest code tested (in this case,

TWOQ) for requested OD pairs that occupies 25%|*N*|, 50%|*N*|, 75%|*N*|, and 100%|*N*| distinct origin nodes. All SLU implementations perform similarly to variants of Dijkstra's algorithm. When the requested OD pairs cover more distinct origin nodes, SLU perform better.

We also solve the APSP problem on AP-NET, compare our codes with Floyd-Warshall algorithm. Table 3 shows that all of our SLU implementations are faster than Floyd-Warshall algorithm on three computer operating systems.

Table 3. Relative Algorithmic Performance on Solving the APSP problem on AP-NET

OS	FWA	FWG	SLU1	SLU2	SLU3
Sun Solaris	8.94	3.91	1.00	2.61	2.56
Mandrake Linux	3.17	1.45	1.00	1.61	1.31
Windows 2000	5.08	2.35	1.00	2.19	2.14

Besides the real flight network, we generate artificial networks by SPGRID and SPACYC designed by Cherkassky et al. (1996). SPGRID generates grid-like networks with X horizontal lines and Y vertical lines crossing each other plus a super node. By changing X and Y we can specify the grid shape to be square (SPGRID-SQ), wide or long (SPGRID-WL). SPACYC generates acyclic networks. It first constructs a central path starting from node 1 that visits every other node exactly once, and then randomly connects lower nodes to higher nodes. We test two sparse family with positive arc length (SPACYC-PS) and negative arc length (SPACYC-NS). Here are detailed settings for these artificial networks:

- SPGRID-SQ: (SQ stands for square grid) $X = Y \in \{10, 20, 30, 40, 50, 60, 70, 80\}$; arc length is ranged from 10^3 to 10^4.
- SPGRID-WL: (WL stands for wide or long grid) Wide: $X = 16, Y \in \{64, 128, 256, 512\}$; Long: $Y = 16, X \in \{64, 128, 256, 512\}$; arc length is ranged from 10^3 to 10^4.
- SPACYC-PS: (P stands for positive arc length; S stands for sparse graphs) $|N| \in \{128, 256, 512\}$; average degree is either 4 or 16; arcs in the central path have length $= 1$, all other arcs have lengths uniformly distributed in $[0, 10^4]$
- SPACYC-NS: (N stands for negative arc length; S stands for sparse graphs) Similar as SPACYC-PS except arcs in the central path have length equal to -1, and all other arcs have lengths uniformly distributed in $[-10^4, 0]$
- In all of our tests on artificial networks (see Table 4, 5, 6, and 7), we generate random OD pairs that cover 75%|*N*| distinct nodes.

From Table 2 and Table 3, SLU1 seems to be the most efficient SLU implementation. This observation indeed applies for most of our computational tests. Thus we use SLU1 to represent the SLU implementation for the remaining comparisons.

Table 4. Relative Algorithmic Performance on Solving a MPSP problem on SPGRID-SQ

	SLU	GOR	BFP	THR	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
Grid/deg	**1**	**1**		**ESH**						
10x10/3	5.00	5.50	1.30	3.30	1.10	1.00	7.30	6.10	10.90	26.10
20x20/3	5.54	5.04	1.18	2.58	1.08	1.00	8.01	4.40	9.43	11.84
30x30/3	4.97	4.27	1.13	2.01	1.05	1.00	6.82	3.27	7.15	6.52
40x40/3	5.85	4.56	1.12	2.15	1.05	1.00	7.18	3.24	7.14	5.22
50x50/3	5.55	5.11	1.13	2.04	1.04	1.00	7.27	3.09	6.81	4.56
60x60/3	6.00	5.24	1.13	1.95	1.03	1.00	6.92	2.91	6.37	3.88
70x70/3	6.86	4.67	1.14	2.13	1.05	1.00	7.35	3.04	6.62	3.73
80x80/3	8.64	5.65	1.14	2.14	1.05	1.00	7.54	3.05	6.55	3.54

Table 5. Relative Algorithmic Performance on Solving a MPSP problem on SPGRID-WL

	SLU	GOR	BFP	THR	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
Grid/deg	**1**	**1**		**ESH**						
16x64/3	5.66	4.38	1.09	2.05	1.05	1.00	6.22	3.88	8.42	9.69
16x128/3	5.70	5.15	1.12	1.99	1.05	1.00	5.82	3.61	8.20	8.12
16x256/3	5.65	4.90	1.10	2.03	1.05	1.00	5.76	3.70	8.49	7.83
16x512/3	7.11	5.60	1.09	2.04	1.03	1.00	5.41	3.53	8.33	7.24
64x16/3	3.93	4.81	1.13	2.22	1.06	1.00	8.39	3.25	6.78	5.60
128x16/3	3.38	4.97	1.15	2.02	1.03	1.00	8.46	2.90	5.78	3.68
256x16/3	3.39	4.86	1.12	1.93	1.03	1.00	9.61	2.98	5.77	3.27
512x16/3	3.51	5.06	1.15	1.80	1.04	1.00	10.62	2.97	5.64	3.02

Table 4 and Table 5 show that label-correcting algorithms such as PAPE or TWOQ perform the best in the grid networks. SLU1 performs similarly to GOR1 and is relatively faster for grid networks with long (or wide) shape than square shape. Dijkstra's algorithm performs relatively better for larger grid networks.

Table 6. Relative Algorithmic Performance on Solving a MPSP problem on SPACYC-PS

		ACC	SLU	GOR	BFP	THR	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
$\|N\|$**/deg**	**[***L***,** *U***]**		**1**	**1**		**ESH**						
128/4	$[0, 10^4]$	1.50	1.00	1.00	2.00	2.50	1.50	2.00	4.00	1.50	4.00	7.50
256/4	$[0, 10^4]$	1.00	1.22	1.11	1.89	2.00	1.56	1.78	3.67	1.67	3.33	3.89
512/4	$[0, 10^4]$	1.00	1.38	1.36	2.74	2.17	2.21	2.50	3.50	1.29	2.74	2.26
128/16	$[0, 10^4]$	1.67	1.00	2.00	2.67	2.33	2.33	2.67	3.00	2.33	3.33	7.00
256/16	$[0, 10^4]$	1.11	1.61	1.00	2.17	1.72	2.06	2.17	2.28	1.28	2.00	2.50
512/16	$[0, 10^4]$	1.00	1.50	1.15	2.69	1.97	2.87	2.86	2.28	1.17	1.81	1.63

Table 7. Relative Algorithmic Performance on Solving a MPSP problem on SPACYC-NS

|N|/deg	[L, U]	ACC 1	SLU 1	GOR 1	BFP ESH	THR PE	PAPE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[-10^4, 0]$	1.00	4.00	6.00	5.00	9.00	7.00	6.00	80.00	11.00	14.00	12.00
256/4	$[-10^4, 0]$	1.00	1.20	1.20	2.30	4.50	3.40	3.20	41.50	6.80	8.90	6.90
512/4	$[-10^4, 0]$	1.00	1.19	1.15	5.98	9.25	7.94	8.42	132.42	12.27	14.88	12.44
128/16	$[-10^4, 0]$	1.00	1.40	1.40	3.00	6.40	7.20	6.20	43.80	7.00	8.40	7.60
256/16	$[-10^4, 0]$	1.00	1.61	1.22	4.78	10.67	16.39	11.78	131.44	10.89	13.00	11.39
512/16	$[-10^4, 0]$	1.00	1.44	1.22	11.55	31.16	57.77	47.94	521.17	28.78	32.91	29.39

For acyclic networks, ACC performs the best, followed by SLU1 and GOR1. For acyclic networks with positive arc lengths, label-setting codes perform relatively worse for cases with fewer nodes. For cases with negative arc lengths, both label-correcting and label-setting codes perform worse than SLU1 and GOR1, especially for networks with larger degree.

4. CONCLUSIONS AND RECOMMENDATIONS

This paper presents the first computational study in the topics of solving multiple pairs shortest path problems, which are more general and closer to real applications. We suggest a new MPSP algorithm (SLU) exploiting the sparse LU decomposition techniques. We give three efficient implementations and conduct preliminary computational experiments on a real flight network and artificial networks with grid and acyclic structures. The computational results show SLU is definitely faster than Floyd-Warshall algorithm, and is competitive in solving MPSP problems. Label-correcting algorithms solve the MPSP problem in a real flight network using the smallest amount of time, then followed by SLU1 and label-setting algorithms. When solving MPSP problems of 75%|*N*| distinct origin nodes on grid networks generated by SPGRID, label-correcting algorithms perform the most efficient, then followed by Dijkstra's bucket implementations, SLU, and Dijkstra's heap implementations. SLU performs faster in grid networks with long (or wide) shape than square shape. Since SLU performs acyclic operations (*G_Forward* and *G_Backward*), it is very efficient in solving MPSP problems over acyclic networks generated by SPACYC, whether the arc lengths are positive or not. The conventional label-correcting and label-setting algorithms both perform inefficiently in the acyclic networks we tested except GOR1.

If shortest paths have to be computed repeatedly over the same network with changed arc lengths or requested OD pairs, SLU takes advantages by preprocessing which suggests a good node ordering that minimizes fill-ins produced in the LU decomposition phase. Minimizing fill-ins also improve the efficiency of *G_Forward* and *G_Backward*.

There are many factors that make computational evaluation on solving MPSP problems difficult. Algorithms may perform very differently on networks with different sizes, densities, topologies, ranges of arc length, or even the requested OD pairs (scattered or concentrated in the $n \times n$ OD matrix). Our experiments are a first step in evaluating algorithmic efficiency when solving general MPSP problems. More thorough experiments will be done in the future.

ACKNOWLEDGEMENTS

This research is supported by NSC 92-2213-E-006-094. The author also thanks Dr. Cherkassky, Dr. Goldberg, and Dr. Radzik for making their shortest path C source codes available.

REFERENCES

Bellman, R. (1958) On a routing problem. *Quarterly of Applied Mathematics*, Vol. 16, 87-90.

Burton, D. (1993) On the inverse shortest path problem. Ph.D. Thesis, Departement de Mathematique,Faculte des Sciences, Facultes Universitaires Notre-Dame de la Paix de Namur.

Carré, B. (1971) An algebra for network routing problems. *Journal of Institute of Mathematics and Its Applications*, Vol. 7, 273-294.

Cherkassky, B., Goldberg, A., and Radzik, T. (1996) Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, Vol. 73, 129-174.

Davis, T., Gilbert, J., Larimore, S., and Ng, E. (2000) A column approximate minimum degree ordering algorithm. Technical report tr-00-005, Department of Computer and Information Science and Engineering, University of Florida.

Dial, R. (1965) Algorithm 360 shortest path forest with topological ordering. *Communications of the ACM*, Vol. 12, 6323-633.

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

Duff, I., Erisman, A., and Reid, J. (1989) *Direct methods for sparse matrices*. New York: Oxford University Press Inc.

Florian, M., Nguyen, S., and Pallottino, S. (1981) A dual simplex algorithm for finding all shortest paths. *Networks*, Vol. 11, No. 4, 367-378.

Floyd, R. (1962) Algorithm 97, shortest path. *Comm. ACM*, Vol. 5, 345.

Ford Jr., L. R. (1956) *Network flow theory.* Santa Monica, California, The RAND Corp.

Glover, F., Glover, R., and Klingman, D. (1984) Computational study of an improved shortest path algorithm. *Networks*, Vol. 14, No. 1, 25-36.

Goldberg, A. and Radzik, T. (1993) A heuristic improvement of the bellman-ford algorithm. *Applied Mathematics Letters*, Vol. 6, No. 3, 3-6.

Goldfarb, D., Hao, J., and Kai, S. (1990) Efficient shortest path simplex algorithms. *Operations Research*, Vol. 38, No. 4, 624-628.

Goto, S., Ohtsuki, T., and Yoshimura, T. (1976) Sparse matrix techniques for the shortest path problem. *IEEE Transactions on Circuits and Systems*, Vol. CAS-23, 752-758.

Karypis, G. and Kumar, V. (1999) A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, 359-392.

Kershenbaum, A. (1981) A note on finding shortest paths trees. *Networks*, Vol. 11, No. 4, 399-400.

Liu, J. W. (1985) Modification of the minimum degree algorithm by multiple elimination, *ACM Trans. Math. Software*, Vol. 11, 141-153.

Markowitz, H. (1957) The elimination form of the inverse and its application to linear programming. *Management Science*, Vol. 3, 255-269.

Moore, E. (1957) The shortest path through a maze. *Proceedings of the International Symposium on theory of Switching* , *Part II*, pp. 285-292.

Nguyen, S., Pallottino, S., and Scutellà, M. (2001) A new dual algorithm for shortest path reoptimization. In Gendreau, M. and Marcotte, P. (eds.), *Transportation and Network Analysis - Current Trends*, Kluwer Academic Publishers.

Pallottino, S. (1984) Shortest-path methods: complexity, interrelations and new propositions. *Networks*, Vol. 14, 257-267.

Pallottino, S. and Scutellà, M. (1997) Dual algorithms for the shortest path tree problem. *Networks*, Vol. 29, No. 2, 125-133.

Pape, U. (1974) Implementation and efficiency of algorithms for the shortest root Moore problem. *Mathematical Programming*, Vol. 7, 212-222.

Rose, D. and Tarjan, R. (1978) Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, Vol. 34, No. 1, 176-197.

Warshall, S. (1962) A theorem on Boolean matrices. *Journal of ACM*, Vol. 9, 11-12.