Yokohama Publishers
ISSN 1349-8169 ONLINE JOURNAL

Pacific Journal of Optimization
Vol. 10, No. 3, pp. 561–576, July 2014

AN ALGEBRAIC DECOMPOSED ALGORITHM FOR ALL PAIRS SHORTEST PATHS*

I-Lin Wang

Abstract: Unlike most recent shortest path algorithms that converge based on graphical operations, we propose an algebraic all pairs shortest path algorithm based on LU decomposition of a measure matrix recording arc lengths. Our algorithm is capable of identifying negative cycles of the graph in fewer iterations than Floyd-Warshall algorithm does, and is at least as efficient as Floyd-Warshall algorithm on a complete graph. For more general graphs, the computational results indicate that the numerical performance of our algorithm is competitive by winning Floyd-Warshall algorithm on 12 out of 18 network families randomly created by two popular network generators SPGRID and SPRAND.

Key words: *shortest path, all pairs, algebraic method, LU decomposition*

Mathematics Subject Classification: *05C85, 05C12, 68R10*

1 Introduction

Shortest path problems seek the shortest paths between specific source and sink nodes in a network. The *Single Source (or Sink) Shortest Path* (SSSP) algorithms compute a shortest path tree for a specific source (or sink) node which usually employ combinatorial or network traversal techniques such as label-setting methods and label-correcting methods [2]; or linear programming (LP) based techniques like the primal network simplex method [15, 16] and the dual ascent method [24]. On the other hand, the *All Pairs Shortest Paths* (APSP) algorithms compute shortest paths for all the node pairs and are based on algebraic or matrix techniques such as Floyd-Warshall [12, 29] and Carré's [6, 7] algorithms. Recently, Wang et al. [28] gives an algebraic *Multiple Pairs Shortest Paths* (MPSP) algorithm which is more efficient than SSSP and APSP algorithms for computing shortest paths for specific node pairs.

For a *digraph* $G := (N, A)$ with $n = |N|$ nodes and $m = |A|$ arcs, obviously the APSP problem can be solved by applying an SSSP algorithm n times. We call such methods repeated SSSP algorithms. Repeated SSSP algorithms usually perform arc traversal operations and require $O(m)$ storage and therefore are more suitable for sparse networks. On the other hand, algebraic APSP algorithms perform operations on a $n \times n$ *distance matrix* $X = [x_{ij}]$ that stores temporary distance label for each node pair. Thus APSP algorithms require more storage ($O(n^2)$) and are more suitable for dense networks. This paper focus on the topics of algebraic APSP algorithms.

*I-Lin Wang was partially supported by the National Science Council of Taiwan under Grant NSC102-2221-E-006-141-MY3.

Algebraic shortest path algorithms are closely related to *path algebra* as discussed in [3, 7].whose operators $(\oplus, \otimes, null, e)$ 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. Using path algebra, the APSP problem is to determine the $n \times n$ shortest distance matrix $X = [x_{ij}]$ that satisfies the Bellman's equation $X = CX \oplus I_n$ [7], where $C = [c_{ij}]$ is the $n \times n$ *measure matrix* storing the length of arc (i, j) (represented as c_{ij}, $c_{ij} = \infty$ if $(i, j) \notin A$) and I_n is the identity matrix. Therefore, we may apply techniques of solving systems of linear equations to solve the APSP problem. For example, direct methods like the *Gauss-Jordan* and *Gaussian elimination* correspond to the *Floyd-Warshall* [12, 29] and *Carré's* [6, 7] algorithms, respectively; iterative methods like the *Jacobi* and *Gauss-Seidel* methods actually correspond to the SSSP algorithms by Bellman [4] and Ford [13], respectively (see [7] for proofs of their equivalence); and the relaxation method of Bertsekas [5] can also be interpreted as a Gauss-Seidel technique (see [24]). Since the same problem can also be viewed as inverting the matrix $(I_n - C)$, the *escalator method* [22] for inverting a matrix corresponds to an inductive APSP algorithm proposed by Dantzig [9]. Finally, the *decomposition algorithm* proposed by Mill [21] (also, Hu [18]) decomposes a huge graph into parts, solves APSP for each part separately, and then reunites the parts. This resembles the *nested dissection method* (see Chapter 8 in [10]), a partitioning or tearing technique to determine a good elimination ordering for maintaining sparsity, when solving a huge system of linear equations. All of these methods (except the iterative methods) have $O(n^3)$ time bounds and are believed to be efficient for dense graphs.

It can be shown that the solution to the Bellman's equation is $X^* = (I_n \oplus C)^{n-1}$. Shimbel [26] suggests a naive algorithm using $\log(n)$ matrix squarings of $(I_n \oplus C)$ to solve the APSP problem. To avoid many distance matrix squarings, some $O(n^3)$ distance matrix multiplication methods such as the *revised matrix* [17,30] and *cascade* [11,19,30] algorithms perform only two or three successive distance matrix squarings. However, Farbey et al. [11] show that these methods are still inferior to the Floyd-Warshall algorithm which only needs a single distance matrix squaring procedure.

Aho et al. (see [1], pp.202-206) show that computing $(I_n \oplus C)^{n-1}$ is as hard as a single distance matrix squaring which can be done in $O(n^{2.5})$ time by Fredman [14], or in $O(n^3((\log\log n)/\log n)^{\frac{1}{2}})$ time by Takaoka [27]. Recently, many algebraic APSP algorithms of subcubic time bounds exploit block decomposition and fast matrix multiplication techniques but are only applicable for specialized networks which are unweighted and undirected, or require the arc lengths to be either integers of small absolute value [31]. These methods are designed mainly for improving the theoretical complexity but not for practical efficiency consideration.

Inspired by Carré's algorithm [6,7] which use Gaussian elimination to solve $X = CX \oplus I_n$, we propose a new algebraic APSP algorithm *FRLU* that is as efficient as Carré's algorithm and Floyd-Warshall algorithm in solving APSP on a complete graph. We use the name *FRLU* for our algorithm since it contains procedures similar to the LU decomposition in linear algebra but the operations are conducted in both forward (F) and reverse (R) directions. *FRLU* conducts operations similar to the MPSP algorithm *DLU* proposed by Wang et al. [28], but converges to the optimal solution in different sequence of operations. *FRLU* can handle networks with general arc length (i.e. an arc may have negative length). In particular, when a network contains a negative cycle, *FRLU* can detect it with fewer operations than Floyd-Warshall algorithm (see the proof of Theorem 3.3 for details).

This paper contains five Sections. Section 1 reviews APSP algorithms. Section 2 introduces some definitions and basic concepts. Section 3 presents our APSP algorithm (*FRLU*) and proves of its correctness. In Section 4, we demonstrate how *FRLU* may save some computational work in computing shortest distances for multiple pairs shortest path (MPSP)

problems. We also conduct computational tests to evaluate the empirical performance of *FRLU*. Section 5 concludes our work and proposes future research.

2 Preliminaries

Here we give notations and definitions used in this paper. The *distance matrix* $[x_{ij}]$ is an $n \times n$ array in which x_{ij}, initialized as c_{ij}, records the length of a path from i to j. Let $[succ_{ij}]$ denote an $n \times n$ *successor matrix* in which $succ_{ij}$, initialized as j, represents the node that immediately follows i in a path from i to j. Using $[succ_{ij}]$, a path from i to j can be traced. In particular, suppose $i \to k_1 \to k_2 \to \cdots \to k_r \to j$ 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} and $succ^*_{ij}$ denote the shortest distance and successor from i to j in G.

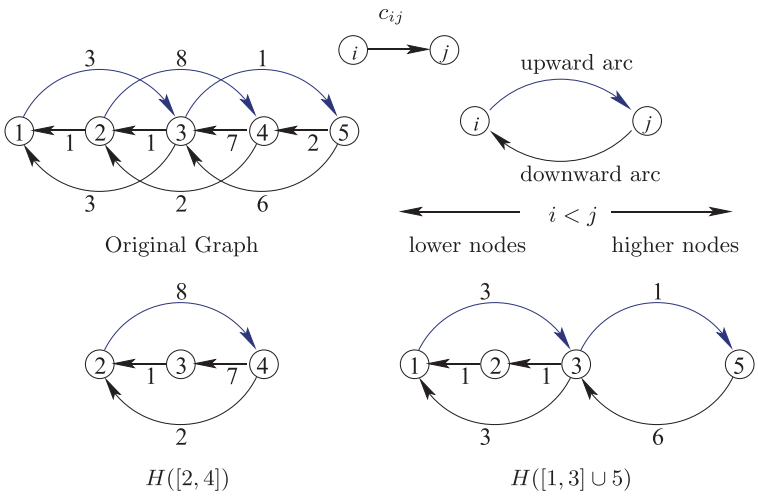

Figure 1: Illustration of node ordering and subgraphs $H([2,4])$, $H([1,3] \cup 5)$

A *triple comparison* $s \to k \to t$ compares $x_{sk} + x_{kt}$ with x_{st}, which 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 length equal to $x_{sk} + x_{kt}$, if $(s,t) \notin A$. Shortest path algorithms operate by performing sequences of triple comparisons [7]. For example, every SSSP algorithm performs distance label updating operation which updates $d[j] = \min_{i:(i,j)\in A} \{d[j], d[i] + c_{ij}\}$, and this is exactly a triple comparison. Actually, even network simplex method implicitly performs a form of triple comparison when it calculates reduced costs. Therefore, 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 index $i > j$ ($i < j$). A node i in a node set $LIST$ is said to be the *highest* (*lowest*) node in $LIST$ if $i \geq k$ ($i \leq k$) $\forall$ $k \in LIST$. 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 shortest paths among these three induced subgraphs. Here in this paper, we give a new algebraic algorithm that systematically calculate shortest paths for these cases to obtain a shortest path in G from s to t.

For solving MPSP problems that arise often in multicommodity networks such as telecommunication and transportation networks, *FRLU* may save some computational work compared with the Floyd-Warshall algorithm, with proper ordering on the indices of nodes. For convenience, we present *FRLU* as an MPSP algorithm which solves shortest distances for a set of q requested OD pairs $Q := \{(s_i,t_i) : i = 1,\ldots,q\}$. Thus the original APSP problem is just a special case where the requested OD pairs covers the entire $n \times n$ OD matrix except the diagonal entries.

3 Algorithm *FRLU*

Given a set of q requested OD pairs $Q := \{(s_i,t_i) : i = 1,\ldots,q\}$, we set i_0 to be the index of the lowest origin node in Q, j_0 to be the index of the lowest destination node in Q, and k_0 to be $\min_i\{\max\{s_i,t_i\}\}$. Algorithm *FRLU* computes x^*_{st} for each $s \geq k_0$, $t \geq j_0$ and for each $s \geq i_0$, $t \geq k_0$. Thus the shortest path lengths for all the OD pairs in Q will be computed without computing the entire shortest path trees as required by other SSSP algorithms. To trace shortest paths for all the requested OD pairs, *FRLU* has to compute the shortest path trees rooted at sink node t for each $t = j_0,\ldots,n$. This can be done by setting $i_0 := 1$ and $k_0 := j_0$ in the algorithm.

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

begin
  Initialize i_0, j_0, k_0, $[x_{ij}]$ and $[succ_{ij}]$;
  Forward_LU;
  Acyclic_LU(i_0, j_0);
  Reverse_LU(i_0, j_0, k_0);
end

Algorithm *FRLU* first initializes $[x_{ij}]$ and $[succ_{ij}]$, then performs three procedures: (1) *Forward_LU* (2) *Acyclic_LU*(i_0, j_0) and (3) *Reverse_LU*(i_0, j_0, k_0). In particular, to solve a shortest path in G from s to t, *Forward_LU* first calculates a shortest path in the subgraph $H([1,\min\{s,t\}]\cup\max\{s,t\})$, then *Acyclic_LU*$(i_0, j_0)$ further considers the subgraph $H([\min\{s,t\},\max\{s,t\}])$ and calculates a shortest path in $H([1,\max\{s,t\}])$. Then *Reverse_LU*(i_0, j_0, k_0) includes the subgraph $H(\min\{s,t\}\cup[\max\{s,t\},n])$ and finds a shortest path in G. Details about each procedure are discussed in the following sections.

3.1 Procedure *Forward_LU*

The first procedure *Forward_LU* resembles the LU decomposition in Gaussian elimination. In the k^{th} iteration of LU decomposition in Gaussian elimination, we use diagonal entry (k,k) to eliminate entry (k,t) for each $t > k$. This will update the $(n-k)\times(n-k)$ submatrix and create fill-ins. Similarly, *Forward_LU* sequentially uses each node $k = 1,\ldots,(n-2)$ as an intermediate node to update each entry (s,t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $k < s \leq n$ and $k < t \leq n$ as long as $x_{sk} < \infty$, $x_{kt} < \infty$ and $x_{st} > x_{sk} + x_{kt}$.

```
Procedure Forward_LU
begin
  for k = 1 to n − 2 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 ≠ t and x_st > x_sk + x_kt then
          x_st := x_sk + x_kt; succ_st := succ_sk;
end
```

Figure 2(a) illustrates the operations of *Forward_LU* on a 5-node graph. It sequentially uses node 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, *Forward_LU* can be viewed as a process of constructing the *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 with indices smaller than both end nodes of the path.

Forward_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, a shortest path for any node pair (s, t) in $H([1, \min\{s, t\}] \cup \max\{s, t\})$ will be computed, and thus $x_{n,n-1} = x^*_{n,n-1}$ and $x_{n-1,n} = x^*_{n-1,n}$ since $H([1, n-1] \cup n) = G$. (see Corollary 3.2)

Theorem 3.1. *After performing procedure Forward_LU, $[x_{st}]$ represents the length of the shortest path from s to t in $H([1, \min\{s, t\}] \cup \max\{s, t\})$.*

Proof. See [28]. □

Corollary 3.2. (a) *Procedure Forward_LU will correctly compute $x^*_{n,n-1}$ and $x^*_{n-1,n}$.*
(b) *Procedure Forward_LU will correctly compute a shortest path for any node pair (s, t) in $H([1, \min\{s, t\}] \cup \max\{s, t\})$.*

Proof. See [28]. □

The next result demonstrates that any negative cycle will also be identified in procedure *Forward_LU*.

Theorem 3.3. *Suppose 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. Then, procedure Forward_LU will identify C_p as a negative cycle faster than the Floyd-Warshall algorithm does.*

Proof. Without loss of generality, let i_1 be the lowest node in the cycle C_p, i_r be the second lowest, i_s be the second highest, and i_t be the highest node. Let $length(C_p)$ denote the length function of cycle C_p. Assume that $length(C_p) = \sum_{(i,j) \in C_p} c_{ij} < 0$. In *Forward_LU*, before we begin iteration 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 we finish scanning the downward arcs entering i_1 and upward arcs leaving i_1, we can identify a smaller cycle C_{p-1} by skipping i_1 and updating $x_{i_p i_2} = \min\{x_{i_p i_2}, x_{i_p i_1} + x_{i_1 i_2}\}$ (it may add a new arc (i_p, i_2) to G if $(i_p, i_2) \notin A$). In particular, C_{p-1} is $i_p \to i_2 \to \cdots \to i_{p-1} \to i_p$, and $length(C_{p-1}) = length(C_p) - (x_{i_p i_1} +$

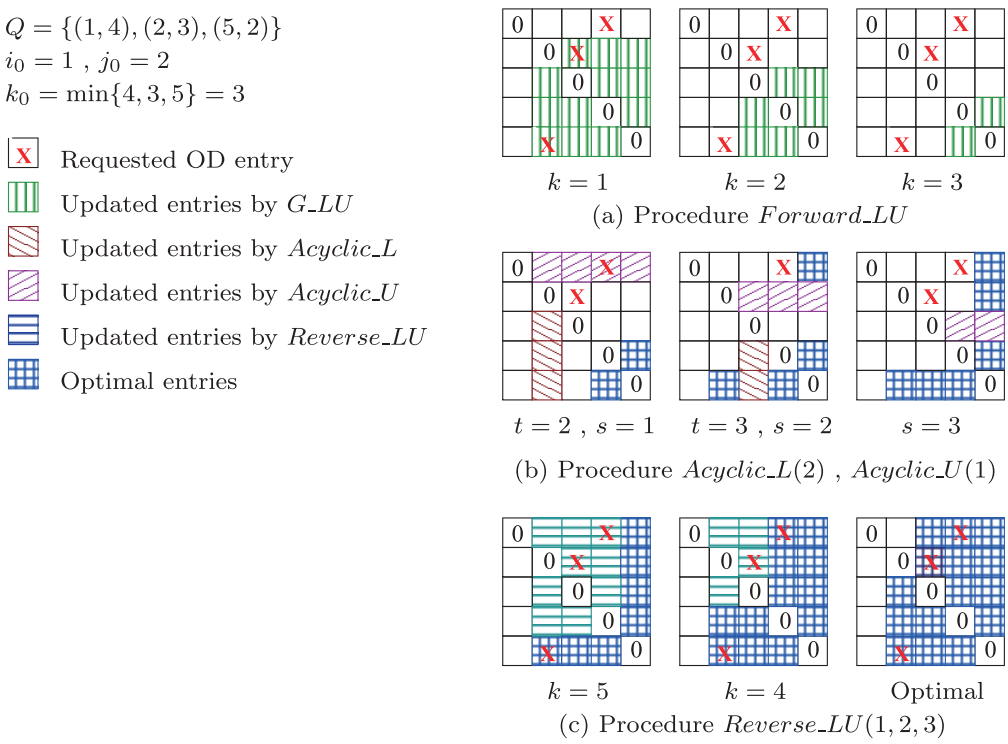

Figure 2: Solving a 3 pairs shortest path problem on a 5-node graph by Algorithm $FRLU(Q)$

$x_{i_1 i_2} - x_{i_p i_2}$). Since $x_{i_p i_2} \leq x_{i_p i_1} + x_{i_1 i_2}$ by the algorithm, we obtain $length(C_{p-1}) \leq length(C_p) < 0$. The lowest-index node in C_{p-1} is now i_r, thus we will again reduce the size of C_{p-1} by 1 arc in iteration i_r. We iterate this procedure, each time processing the current lowest node in the cycle and reducing the cycle size by 1 arc, until finally a 2-node cycle C_2, $i_s \to i_t \to i_s$, with $length(C_2) \leq length(C_3) \leq \cdots \leq length(C_p) < 0$ is obtained. In other words, any negative cycle C_p will induce a negative $x_{i_t i_t}$ in procedure $Forward\_LU$. Thus by checking whether any diagonal entry in $[x_{ij}]$ becomes negative after $Forward\_LU$, we shall know whether there exists a negative cycle. $Forward\_LU$ performs at most $\sum_{k=1}^{n-2} \sum_{s=k+1}^{n} \sum_{t=k+1, s \neq t}^{n} (1) = \frac{1}{3} n(n-1)(n-2)$ steps to identify a negative cycle, if one exists. On the other hand, Floyd-Warshall algorithm takes $\sum_{k=1}^{n-2} \sum_{\substack{s=1 \\ s \neq k}}^{n} \sum_{\substack{t=1 \\ t \neq s, t \neq k}}^{n} (1) = n(n-1)(n-2)$ steps to identify a negative cycle in the worst case. Thus, $FRLU$ can identify a negative cycle in a smaller time bound (up to a constant factor) than Floyd-Warshall algorithm □

Thus $Forward\_LU$ identifies a negative cycle, if one exists. 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 index lower than $\min\{s,t\}$.

3.2 Procedure *Acyclic_LU*(i_0, j_0)

The second procedure, *Acyclic_LU*(i_0, j_0) contains two symmetric procedures, *Acyclic_L*(j_0) and *Acyclic_U*(i_0), which perform triple comparisons on the lower and upper triangular part of $[x_{ij}]$ and $[succ_{ij}]$ respectively. Figure 2(b) illustrates how *Acylic_L*(2) updates each entry (s,t) that satisfies $s > t \geq 2$ in the lower triangular part of $[x_{ij}]$ and $[succ_{ij}]$, and how *Acylic_U*(1) updates each entry (s,t) such that $t > s \geq 1$ in the upper triangular part of $[x_{ij}]$ and $[succ_{ij}]$.

```
Procedure Acyclic_LU(i0, j0)
begin
   Acyclic_L(j0);
   Acyclic_U(i0);
end

Procedure Acyclic_L(j0)
begin
   for t = j0 to n − 2 do
      Get_D_L(t);
end

Subprocedure Get_D_L(t)
begin
   for s = t + 2 to n do
      for k = t + 1 to s − 1 do
         if x_st > x_sk + x_kt then
            x_st := x_sk + x_kt; succ_st := succ_sk;
end
```

The lower and upper triangular parts of $[x_{ij}]$ induce two acyclic subgraphs, G'_L and G'_U, of augmented graph G'. They can be easily identified by aligning the nodes by ascending order of their indices from the left to the right, where G'_L (G'_U) contains all the downward (upward) arcs of G'.

Graphically, *Acyclic_L*(j_0) performs sequences of shortest path tree computations in G'_L. Its subprocedure *Get_D_L*(t), resembling the forward elimination in Gaussian elimination, performs triple comparisons to update x_{st} by $\min\{x_{st}, x_{sk}+x_{kt}\}$ for each $k = (t+1), \ldots, (s-1)$, and for each $s = (t+2), \ldots, n$. Since G'_L is acyclic, the updated x_{st} thus corresponds to the shortest distance in G'_L from each node $s > t$ to node t. *Acyclic_L*(j_0) repeats *Get_D_L*(t) for each root node $t = j_0, \ldots, (n-2)$. Thus for each OD pair (s,t) satisfying $s > t \geq j_0$, we obtain the shortest distance in G'_L from s to t which in fact corresponds to the shortest distance in $H([1,s])$ from s to t. (see Corollary 3.5(a)) Also, this procedure gives x^*_{nt}, the shortest distance in G from node n to any node $t \geq j_0$. (see Corollary 3.5(c))

Acyclic_U(i_0) is similar to *Acyclic_L*(j_0) except it is applied on the upper triangular part of $[x_{ij}]$ and $[succ_{ij}]$, which corresponds to the induced subgraph G'_U. Each application of subprocedure *Get_D_U*(s) gives the shortest distance in G'_U from each node s to each node $t > s$, and we repeat this subprocedure for each root node $s = i_0, \ldots, (n-2)$. Thus for each OD pair (s,t) satisfying $i_0 \leq s < t$, we obtain the shortest distance in G'_U from s to t which in fact corresponds to the shortest distance in $H([1,t])$ from s to t. (see Corollary 3.5(b)) Also, this procedure gives x^*_{sn}, the shortest distance in G from any node $s \geq i_0$ to node n. (see Corollary 3.5(c))

```
Procedure Acyclic_U(i_0)
begin
   for s = i_0 to n − 2 do
      Get_D_U(s);
end

Subprocedure Get_D_U(s)
begin
   for t = s + 2 to n do
      for k = s + 1 to t − 1 do
         if x_st > x_sk + x_kt then
            x_st := x_sk + x_kt; succ_st := succ_sk;
end
```

Theorem 3.4. (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 such a shortest path in $H([1,s])$ 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 $H([1,s])$ 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 with index smaller than $\min\{s,t\} = t < s$, Theorem 3.1 already shows that $Forward\_LU$ 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 some intermediate node with index in the range $[t+1, s-1]$. Suppose there exist r intermediate nodes, $\{u_i : i = 1,\ldots,r\}$, in this shortest path in $H([1,s])$ from s to t, and $s := u_0 > u_1 > u_2 > \cdots > u_{r-1} > u_r > u_{r+1} := t$. We can break this shortest path into $(r+1)$ segments: u_0 to u_1, u_1 to u_2,..., and u_r to u_{r+1}. Each shortest path segment $u_{k-1} \to u_k$ in $H([1,s])$ contains intermediate nodes that all have lower indices than u_k. Since Theorem 3.1 guarantees that $Forward\_LU$ will produce an arc (u_{k-1}, u_k) for any such 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 $H([1,s])$ will be reduced to the shortest path $s \to u_1 \to u_2 \to \cdots \to u_{r-1} \to u_r \to t$ in G'_L. (b) Using a similar argument to (a) above, the result follows immediately. □

Corollary 3.5. (a) *Procedure $Acyclic\_L(j_0)$ will correctly compute shortest paths in $H([1,s])$ for all node pairs (s,t) such that $s > t \geq j_0$.*
(b) *Procedure $Acyclic\_U(i_0)$ will correctly compute shortest paths in $H([1,t])$ for all node pairs (s,t) such that $i \leq s < t$.*
(c) *Procedure $Acyclic\_L(j_0)$ will correctly compute x^*_{nt} for each node $t \geq j_0$; Procedure $Acyclic\_U(i_0)$ will correctly compute x^*_{sn} for each node $s \geq i_0$*

Proof. (a) This procedure computes sequences of shortest path tree in G'_L rooted at node $t = j_0, \ldots, (n-2)$ from all other nodes $s > t$. By Theorem 3.4(a), a shortest path in G'_L from node $s > t$ to node t corresponds to a shortest path in G from s to t where s is its highest node since all other nodes in this path in G'_L have lower index than s. In other words, such a shortest path corresponds to the same shortest path in $H([1,s])$. Including the case of $t = (n-1)$ and $s = n$ as discussed in Corollary 3.2(a), the result follows directly.

(b) Using a similar argument as part (a), the result again follows directly. (c) These follow immediately from part (a) and part (b). □

Thus *Ayclic_LU*(i_0, j_0) will have computed the shortest distance in $H([1, \max\{s,t\}])$ from each node $s \geq i_0$ to each node $t \geq j_0$. In other words, this procedure computes shortest path lengths for those requested OD pairs (s,t) whose shortest paths have all intermediate nodes with index lower than $\max\{s,t\}$.

3.3 Procedure *Reverse_LU*(i_0, j_0, k_0)

The final step *Reverse_LU*(i_0, j_0, k_0) is similar to the first procedure *Forward_LU* but in a reverse fashion. It computes the length of the shortest paths in $H([1, \max\{s,t\}] \cup r)$ that must pass through intermediate node r for each $r = n$ down to $(\max\{s,t\} + 1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ and from each origin $s \geq i_0$ to each destination $t \geq k_0$. Since previous procedures already give the shortest distances in $H([1, \max\{s,t\}])$ from each node $s \geq i_0$ to each node $t \geq j_0$, *Reverse_LU*(i_0, j_0, k_0) continues the remaining necessary triple comparisons to compute the x^*_{st} in G. Figure 2(c) illustrates the operations

---

Procedure Reverse_LU$(\mathbf{i_0}, \mathbf{j_0}, \mathbf{k_0})$
begin
  for $k = n$ down to $k_0 + 1$ ***do***
    for $s = k - 1$ down to i_0 ***do***
      for $t = k - 1$ down to j_0 ***do***
        if $s \neq t$ and $x_{st} > x_{sk} + x_{kt}$ ***then***
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

---

of *Reverse_LU*$(1, 2, 3)$ which updates each entry (s,t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $1 \leq s < k$, $2 \leq t < k$ for each $k = 5$ and 4. Note that x^*_{st} for all $s \geq i_0$, $t \geq k_0$ and $s \geq k_0$, $t \geq j_0$ will have been obtained after *Reverse_LU*(i_0, j_0, k_0) and thus shortest distances for all the requested OD pairs in Q will have been computed.

Lemma 3.6. (a) *Every shortest path in G from s to t that has a highest node with index $h > \max\{s,t\}$ 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) *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 $v \leq r$, and* (ii) *the shortest path from s to t in G that must pass through some node $v > r$, where $1 \leq r \leq n$.*

Proof. (a) This follows immediately by combining Corollary 3.5(a) and (b). (b) It is easy to see that every path from s to t must either passes through some node $v > r$ or else not. Therefore the shortest path from s to t must be the shorter of the minimum-length paths of each type. □

Theorem 3.7. *After performing the k^{th} iteration of the outer loop, Reverse_LU(i_0, j_0, k_0) will correctly compute $x^*_{n-k,t}$ and $x^*_{s,n-k}$ for each $s = i_0, \ldots, (n-k-1)$, and for each $t = j_0, \ldots, (n-k-1)$ where $k \leq (n - k_0)$.*

Proof. After procedures *Acyclic_LU*(i_0, j_0), we will have obtained shortest paths in

$H([1, \max\{s,t\}])$ from each node $s \geq i_0$ to each node $t \geq j_0$. To obtain the shortest path in G from s to t, we remain to check those shortest paths that must pass through node h for each $h = (\max\{s,t\}+1), \ldots, n$. By Lemma 3.6(a), such a shortest path can be decomposed into two segments: from s to h and from h to t. Note that their shortest distances, x_{sh} and x_{ht}, will have been calculated by *Acyclic_U*(i_0) and *Acyclic_L*(j_0), respectively. For each node $s \geq i_0$ and $t \geq j_0$, Corollary 3.5(c) shows that x^*_{nt} and x^*_{sn} will have been computed by procedures *Acyclic_L*(j_0) and *Acyclic_U*(i_0), respectively. Define x^k_{st} to be the shortest known distance from s to t after the k^{th} application of *Reverse_LU*(i_0, j_0, k_0), and x^0_{st} to be the best known distance from s to t before this procedure. Now we will prove this theorem by induction. In the first iteration, *Reverse_LU*(i_0, j_0, k_0) updates $x^1_{st} := \min\{x^0_{st}, x^0_{sn} + x^0_{nt}\}$ $= \min\{x_{st}, x^*_{sn} + x^*_{nt}\}$ for each node pair (s,t) satisfying $i_0 \leq s < n$ and $j_0 \leq t < n$. For node pairs $(n-1, t)$ satisfying $j_0 \leq t < (n-1)$, $x^*_{n-1,t} = \min\{x^0_{n-1,t}, x^*_{n-1,n} + x^*_{nt}\}$ by applying Lemma 3.6(b) with $r = (n-1)$. Likewise, $x^*_{s,n-1}$ is also determined for each s satisfying $i_0 \leq s < (n-1)$ in this iteration. Suppose the theorem holds for iteration $k = \widehat{k} < (n - \max\{s,t\})$. That is, at the end of iteration $k = \widehat{k}$, *Reverse_LU*(i_0, j_0, k_0) gives $x^*_{n-\widehat{k},t}$ and $x^*_{s,n-\widehat{k}}$ for each s satisfying $i_0 \leq s < (n-\widehat{k})$, and each t satisfying $j_0 \leq t < (n-\widehat{k})$. In other words, we will have obtained $x^*_{n-r,t}$ and $x^*_{s,n-r}$ for each $r = 0, 1, \ldots, \widehat{k}$, and for all $s \geq i_0$, $t \geq j_0$. In iteration $k = (\widehat{k}+1)$, for each t satisfying $j_0 \leq t < (n - \widehat{k} - 1)$, $x^{\widehat{k}+1}_{n-\widehat{k}-1,t} := \min\{x^{\widehat{k}}_{n-\widehat{k}-1,t}, x^{\widehat{k}}_{n-\widehat{k}-1,n-\widehat{k}} + x^{\widehat{k}}_{n-\widehat{k},t}\} = \min\{x^{\widehat{k}}_{n-\widehat{k}-1,t}, x^*_{n-\widehat{k}-1,n-\widehat{k}} + x^*_{n-\widehat{k},t}\}$ by assumption of the induction. Note that the first term $x^{\widehat{k}}_{n-\widehat{k}-1,t}$ has been updated $\widehat{k}$ times in the previous $\widehat{k}$ iterations. In particular, $x^{\widehat{k}}_{n-\widehat{k}-1,t} = \min\limits_{0 \leq k \leq (\widehat{k}-1)} \{x^0_{n-\widehat{k}-1,t}, x^*_{n-\widehat{k}-1,n-k} + x^*_{n-k,t}\}$ where $x^0_{n-\widehat{k}-1,t}$ represents the length of a shortest path in G from node $(n-\widehat{k}-1)$ to node t that has node $(n-\widehat{k}-1)$ as its highest node. Substituting this new expression of $x^{\widehat{k}}_{n-\widehat{k}-1,t}$ into $\min\{x^{\widehat{k}}_{n-\widehat{k}-1,t}, x^*_{n-\widehat{k}-1,n-\widehat{k}} + x^*_{n-\widehat{k},t}\}$, we obtain $x^{\widehat{k}+1}_{n-\widehat{k}-1,t} := \min\limits_{0 \leq k \leq \widehat{k}} \{x^0_{n-\widehat{k}-1,t}, x^*_{n-\widehat{k}-1,n-k} + x^*_{n-k,t}\}$ whose second term $\min\limits_{0 \leq k \leq \widehat{k}} \{x^*_{n-\widehat{k}-1,n-k} + x^*_{n-k,t}\}$ corresponds to the length of a shortest path in G from node $(n-\widehat{k}-1)$ to node t that must pass through some higher node with index $v > (n-\widehat{k}-1)$ (v may be $(n-\widehat{k}), \ldots, n$). By Lemma 3.6(b) with $r = (n-\widehat{k}-1)$, we conclude $x^{\widehat{k}+1}_{n-\widehat{k}-1,t} = x^*_{n-\widehat{k}-1,t}$ for each t satisfying $j_0 \leq t < (n-k)$. Likewise, $x^{\widehat{k}+1}_{s,n-\widehat{k}-1} = x^*_{s,n-\widehat{k}-1}$ for each s satisfying $i_0 \leq s < (n-\widehat{k}-1)$ will also be determined in the end of iteration $(\widehat{k}+1)$. By induction, we have shown the correctness of this theorem. □

Corollary 3.8. *Procedure Reverse_LU*(i_0, j_0, k_0) *will terminate after performing* $(n - k_0)$ *iterations of the outer loop, and correctly compute* $x^*_{s_i,t_i}$ *for each of the requested OD pairs* (s_i, t_i), $i = 1, \ldots, q$.

Proof. By setting $k_0 := \min\limits_i\{\max\{s_i, t_i\}\}$, the set of all the requested OD pairs Q is a subset of node pairs $\{(s,t) : s \geq k_0, t \geq j_0\} \cup \{(s,t) : s \geq i_0, t \geq k_0\}$ whose x^*_{st} and $succ^*_{st}$ is shown to be correctly computed by Theorem 3.7. Therefore *Reverse_LU*(i_0, j_0, k_0) terminates in $n - (k_0 + 1) + 1 = (n - k_0)$ iterations and correctly computed $x^*_{s_i t_i}$ and $succ^*_{s_i t_i}$ for each requested OD pair (s_i, t_i) in Q. □

To trace shortest paths for all the requested OD pairs by $[succ_{ij}]$, we set $i_0 = 1$ and $k_0 = j_0$ in the beginning of the algorithm so that at iteration $k = j_0$ the successor columns

$j_0, \ldots, n$ are valid for tracing a shortest path tree rooted at sink node k. Otherwise, if $i_0 > 1$, then *Acylic_U*(i_0) and *Reverse_LU*(i_0, j_0, k_0) will not update $succ_{st}$ for all $s < i_0$. This makes tracing shortest paths for some OD pairs (s, t) difficult if those paths contain intermediate nodes with index lower than i_0. Similarly, if $k_0 > j_0$, *Reverse_LU*(i_0, j_0, k_0) will not update $succ_{st}$ for all $t < k_0$. For example, in the last step of Figure 2(c), if node 1 lies in the shortest path from node 5 to node 2, then we may not be able to trace this shortest path since $succ_{12}$ has not been updated in *Reverse_LU*$(1, 2, 3)$. Therefore even if Algorithm *FRLU* gives the shortest distance for the requested OD pairs earlier, tracing these shortest paths requires more computations.

Corollary 3.9. (a) *To trace the shortest path for each requested OD pair* (s_i, t_i) *in* Q*, we have to initialize* $i_0 := 1$ *and* $k_0 := j_0$ *in the beginning of Algorithm* FRLU.
(b) *Every APSP problem can be solved by Algorithm* FRLU *with* $i_0 := 1$, $j_0 := 1$, *and* $k_0 := 2$.

Proof. (a) The entries $succ_{st}$ for each $s \geq i_0$ and $t \geq j_0$ are updated in all procedures whenever a better path from s to t is identified. To trace the shortest path for a particular OD pair (s_i, t_i), we need the entire t_i^{th} column of $[succ_{ij}^*]$ which contains information of the shortest path tree rooted at sink node t_i. Thus we have to set $i_0 := 1$ so that procedures *G_LU*, *Acyclic_L*(j_0) and *Acyclic_U*(1) will update entries $succ_{st}$ for each $s \geq 1$ and $t \geq j_0$. However, *Reverse_LU*$(1, j_0, k_0)$ will only update entries $succ_{st}$ for each $s \geq 1$ and $t \geq k_0$. Thus it only gives the t^{th} column of $[succ_{ij}^*]$ for each $t \geq k_0$ in which case some entries $succ_{st}$ with $1 \leq s < k_0$ and $j_0 \leq t < k_0$ may still contain incomplete successor information unless we set $k_0 := j_0$ in the beginning of this procedure. (b) We set $i_0 := j_0 := 1$ because we need to update all entries of the $n \times n$ distance matrix $[x_{ij}]$ and successor matrix $[succ_{ij}]$ when solving any APSP problem. Setting $k_0 := 1$ will make the last iteration of *Reverse_LU*$(1, 1, k_0)$ update x_{11} and $succ_{11}$, which is not necessary. Thus it suffices to set $k_0 := 2$ when solving any APSP problem. □

Thus *Reverse_LU*(i_0, j_0, k_0) compares the shortest path lengths obtained by previous procedures with the shortest path lengths for those requested OD pairs (s, t) whose shortest paths have some intermediate nodes with indices higher than $\max\{s, t\}$, which means the resultant x_{st} is optimal, by Corollary 3.8.

Now we discuss the theoretical complexity of *FRLU* and some implementation techniques to improve its practical efficiency.

3.4 Complexity and Implementation of Algorithm *FRLU*

If we skip the triple comparisons for self-loops (i.e. $s \to k \to s$), then procedure *Forward_LU* performs $\sum_{k=1}^{n-2} \sum_{s=k+1}^{n} \sum_{t=k+1, s \neq t}^{n} (1) = \frac{1}{3} n(n-1)(n-2)$ triple comparisons, procedure *Acyclic_LU*(i_0, j_0) performs $\sum_{t=j_0}^{n-2} \sum_{s=t+2}^{n} \sum_{k=t+1}^{s-1} (1) + \sum_{s=i_0}^{n-2} \sum_{t=s+2}^{n} \sum_{k=s+1}^{t-1} (1) = \frac{1}{6}(n - j_0 - 1)(n - j_0)(n - j_0 + 1) + \frac{1}{6}(n - i_0 - 1)(n - i_0)(n - i_0 + 1)$ triple comparisons, and procedure *Reverse_LU*(i_0, j_0, k_0) performs $\sum_{k=k_0+1}^{n} \sum_{s=i_0}^{k-1} \sum_{t=j_0, s \neq t}^{k-1} (1) = \sum_{k=k_0+1}^{n} [(k - i_0)(k - j_0) - (k - \max\{i_0, j_0\})]$ triple comparisons. Thus *FRLU* has an $O(n^3)$ worst case complexity. When solving an APSP problem on a complete graph K_n, *FRLU* performs exactly $n(n-1)(n-2)$

triple comparisons, which is the least number of triple comparisons required as shown by Nakamori [23]. Floyd-Warshall and Carré's algorithms also perform the same amount of triple comparisons, and are better than most SSSP algorithms which require $O(n^3)$ for label-setting algorithms or $O(n^4)$ for label-correcting algorithms. 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 *Acyclic_L* (or *Acyclic_U*) is required.

For sparse graphs, node ordering plays an important role in the efficiency of our algorithms. A bad node ordering will incur more fill-in arcs which resemble the fill-ins created in Gaussian elimination. Computing an ordering that minimizes the fill-ins is NP-complete [25]. Nevertheless, many fill-in reducing techniques such as Markowitz's rule [20], minimum degree method, and nested dissection method (see Chapter 8 in [10]) used in solving systems of linear equations can be exploited here to speed up *FRLU*. Since our algorithms do more computations on higher nodes than lower nodes, optimal distances can be obtained for higher nodes earlier than lower nodes. Thus reordering the requested OD pairs to have higher indices may also shorten the computational time, although such an ordering might incur more fill-in arcs. In general, it is difficult to obtain an optimal node ordering that minimizes the computations required. Here, we use a predefined node ordering to start with our algorithms.

Although *FRLU* is an algebraic algorithm, its "graphical" implementation might greatly improve its practical efficiency. In particular, *Forward_LU* constructs the augmented graph G'. 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 that the algebraic algorithms must perform. Note that the efficiency of procedure *Acyclic_LU* depends on the sparsity of augmented graph G'. Therefore, any fill-in reducing techniques discussed in the previous paragraph will not only reduce the running time of *Forward_LU*, but also make *Acyclic_LU* faster.

4 Computational Analysis and Testing on MPSP Problems

MPSP problems usually arise in real-world applications in which only shortest paths between specific node pairs are requested. In Section 4.1, we demonstrate how *FRLU* may theoretically save more steps than Floyd-Warshall algorithm in calculating MPSPs on a complete graph. Then, we compare the numerical performance of *FRLU* and Floyd-Warshall algorithm for calculating MPSPs on 54 random graphs generated by SPGRID and SPRAND in Section 4.2. SPGRID and SPRAND are popular network generators designed by [8] , and have been commonly used for generating test cases for shortest paths.

4.1 An MPSP Example on a Complete Graph

Conventional algebraic APSP algorithms are "over-kill" when used to solve MPSP problems.

For example, when a MPSP problem of $\frac{1}{4}n^2 - \frac{1}{2}n$ OD pairs $Q := \{(s_i, t_k) : s_i = \frac{n}{2}, \frac{n}{2}+1, \ldots, n, s_i = \frac{n}{2}, \frac{n}{2}+1, \ldots, n, s_i \neq t_i\}$ in a complete graph K_n with even number of nodes is solved by Floyd-Warshall algorithm, it first conducts $(n-1)^2(n-2)$ triple comparisons (i.e. before conducting the last for loop) and then it computes the shortest distance for the $\frac{1}{4}n^2 - \frac{1}{2}n$ requested OD pairs in the last for loop of triple comparisons. On the other hand, *FRLU* can solve the same MPSP problem with fewer operations: $\frac{1}{3}n(n-1)(n-2)$ triple comparisons in *Forward_LU*, $\frac{1}{3}\frac{n}{2}(\frac{n}{2}-1)(\frac{n}{2}-2)$ triple comparisons in *Acyclic_LU* (by setting

$i_0 = j_0 = \frac{n}{2}$), and $\sum_{i=1}^{\frac{n}{2}-1}(i^2 - i) = \frac{1}{24}n(n-1)(n-2)$ triple comparisons in *Reverse_LU* (by setting $i_0 = j_0 = k_0 = \frac{n}{2}$). It saves $\frac{7}{12}n^3 - \frac{19}{8}n^2 + \frac{41}{12}n - 2$ triple comparisons when $n \geq 4$.

This example shows how our algorithm *FRLU* saves more computational work than Floyd-Warshall algorithm does. *FRLU* is especially efficient when the requested OD pairs are grouped in the right lower part of the $n \times n$ OD matrix. This may be achieved by rearranging the node indices. However, such rearrangement may also result in more fill-ins which incur more triple operations for each procedure of *FRLU*. How to obtain the best node ordering such that the total number of triple comparisons will be minimized is not clear and has shown to be NP-complete [25].

4.2 Numerical Performance of *FRLU* on Random Graphs

To further compare the empirical performance of *FRLU* and Floyd-Warshall on solving MPSPs, we conduct computational tests over 54 random networks generated by SPGRID and SPRAND designed in [8].

SPGRID generates a sparse grid network defined by $XY + 1$ nodes, where X and Y respectively represent number of nodes lying on the X and Y axes, and on average each node connects with 3 other nodes. On the other hand, SPRAND first constructs a Hamiltonian cycle, and then adds arcs with distinct random end points. Both SPGRID and SPRAND have been widely used (see [8].for details) for generating test cases for shortest paths. In our testings, 9 SPGRID and 9 SPRAND network families of different layout structures are generated, where each family contains 3 random networks of the same layout structure.

For each generated network that contains n nodes, we further generate $\frac{1}{2}n$ OD pairs of MPSPs that cover about $\frac{1}{2}n$ columns. We have employed the dynamic Markowitz's rule to speed up both algorithms. Note that the resultant node orderings do not necessarily group the requested OD pairs in the right lower part of the $n \times n$ OD matrix. As a result, such settings do not favor *FRLU* and make fair numerical comparisons for both algorithms.

Table 1: Performance of algorithms FRLU and Floyd-Warshall on SPGRID networks

SPGRID			**FRLU**	**Floyd-Warshall**
$X \times Y$/**deg**	**n**	**m**	**time(ms)**	**time(ms)**
16×16/3	257	768	3*	3
16×64/3	1025	3072	146*	156
32×32/3	1025	3072	89*	91
64×16/3	1025	3072	57*	57
128×16/3	2049	6144	241	230
16×128/3	2049	6144	1418	1347
16×256/3	4097	12288	19609	15007
256×16/3	4097	12288	1108	962
64×64/3	4097	12288	4112	4087

* represents cases where FRLU has better or equal performance

Table 1 and Table 2 list the average elapse time (in micro seconds) of *FRLU* and Floyd-Warshall algorithm for calculating MPSPs on random networks generated by SPGRID and SPRAND, respectively. As shown in Table 1, for solving MPSPs on SPGRID random

Table 2: Performance of algorithms FRLU and Floyd-Warshall on SPRAND networks

SPRAND			FRLU	Floyd-Warshall
n	**m**	**deg**	**time(ms)**	**time(ms)**
256	1019	4	37*	38
256	14393	56	72*	73
256	25735	101	66*	72
256	64084	250	67*	72
512	2048	4	299*	309
512	4096	8	435*	436
1024	4089	4	3692*	4165
1024	8170	8	5756*	5777
1024	231707	226	6710	5184

* represents cases where FRLU has better performance

networks, *FRLU* has better or similar performance than Floyd-Warshall algorithm in 4 of 9 network families of smaller size. On the other hand, *FRLU* outperforms Floyd-Warshall algorithm in 8 of 9 SPRAND network families, as shown in Table 2. These numerical results indicate *FRLU* is competitive in calculating MPSPs on general graphs.

5 Conclusions

In this paper we propose a new algebraic APSP algorithm called *FRLU*, which is inspired by the fact that calculating an APSP is equivalent to inverting an $n \times n$ matrix in path algebra as discussed in [3, 7]. Similar to the Floyd-Warshall algorithm, *FRLU* can also deal with graphs with general arc lengths. We have shown that *FRLU* performs exactly the same number of steps (i.e. $n(n-1)(n-2)$) as Floyd-Warshall algorithm does in calculating APSPs on a complete graph of n nodes. Therefore, *FRLU* also has the theoretically best complexity (same as Floyd-Warshall's) for calculating APSPs on dense graphs. In addition, *FRLU* can identify a negative cycle in a smaller time bound (up to a constant factor) than Floyd-Warshall algorithm.

In calculating APSPs on sparse graph by *FRLU* and Floyd-Warshall algorithms, we suggest using the conventional fill-in reducing techniques for solving systems of linear equations, such as Markowitz's rule [20]. The three phases of *FRLU* can have efficient sparse implementation since it involves acyclic operations on two acyclic subgraphs of the augmented graph G' induced by its first procedure $Forward\_LU$. As a result, if G' is sparse, the second procedure *Acyclic_LU* can be implemented efficiently by topological ordering. On the other hand, Floyd-Warshall algorithm may incur a dense graph in its early stages and thus could not have an effective sparse implementation.

When applied for calculating MPSPs, *FRLU* may save some computational works than Floyd-Warshall algorithm by reordering nodes such that the requested OD pairs are closely distributed in the right lower part of the $n \times n$ OD matrix. We have demonstrated that *FRLU* saves up to $\frac{7}{12}n^3 - \frac{19}{8}n^2 + \frac{41}{12}n - 2$ steps than Floyd-Warshall algorithm for calculating some MPSPs on a complete graph K_n. The results of our computational experiments indicate *FRLU* has competitive empirical performance since it outperforms Floyd-Warshall algorithm on 12 of 18 network families generated by SPGRID and SPRAND.

For future directions, we suggest sparse implementations of Carré's algorithm [6, 7] for calculating MPSPs on sparse graphs, since it should take even more advantages of sparsity

than *FRLU*.

References

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

[2] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, New Jersey, U.S.A., 1993.

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

[4] R.E. Bellman, On a routing problem, *Quart. Appl. Math.* 16 (1958) 87–90.

[5] D.P. Bertsekas, *Network Optimization: Continuous and Discrete Models*, Athena Scientific, P.O. Box 391, Belmont, MA 02178-9998, 1998.

[6] B.A. Carré, A matrix factorization method for finding optimal paths through networks, in *I.E.E. Conference Publication (Computer-Aided Design)*, number 51, 1969, pp 388–397.

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

[8] B.V. Cherkassky, A.V. Goldberg and T. Radzik, Shortest paths algorithms: theory and experimental evaluation, *Math. Program.* 73 (1996) 129–174.

[9] G.B. Dantzig, All shortest routes in a graph, in *Theory of Graphs (International Symposium., Rome, 1966)*, Gordon and Breach, New York, 1967, pp. 91–92.

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

[11] B. A. Farbey, A. H. Land and J. D. Murchland, The cascade algorithm for finding all shortest distances in a directed graph, *Management Sci.* 14 (1967) 19–28.

[12] R.W. Floyd, Algorithm 97, shortest path, *Comm. ACM* 5 (1962) 345.

[13] L.R. Ford Jr. and D.R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

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

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

[16] D. Goldfarb and Z. Jin, An O(nm)-time network simplex algorithm for the shortest path problem, *Oper. Res.* 47 (1999) 445–448.

[17] T.C. Hu, Revised matrix algorithms for shortest paths, *SIAM J. Appl. Math.* 15 (1967) 207–218.

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

[19] A.H. Land and S.W. Stairs, The extension of the cascade algorithm to large graphs, *Management Sci.* 14 (1967) 29–33.

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

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

[22] J. Morris, An escalator process for the solution of linear simultaneous equations, *Philos. Mag.* 37 (1946) 106–120.

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

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

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

[26] A. Shimbel, Applications of matrix algebra to communication nets, *Bull. Math. Biophys.* 13 (1961) 165–178.

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

[28] I.-L. Wang, E.L. Johnson, and J.S. Sokol, A multiple pairs shortest path algorithm, *Transp. Sci.* 39 (2005) 465–476.

[29] S. Warshall, A theorem on Boolean matrices. *J. ACM* 9 (1962) 11–12.

[30] L. Yang and W.K. Chen, An extension of the revised matrix algorithm, in *IEEE ISCAS*, Portland, Oregon, May 1989, pp. 1996–1999.

[31] U. Zwick, All pairs shortest paths in weighted directed graphs - exact and almost exact algorithmsin *Proceedings of the 39th Annual IEEE Symposium on Found. Comput. Sci.*, Palo Alto, California, November 1998, pp. 310–319.

Manuscript received 11 December 2013
revised 25 June 2014
accepted for publication 26 June 2014

I-Lin Wang
Department of Industrial and Information Management
National Cheng Kung University, Tainan 701, Taiwan
E-mail address: ilinwang@mail.ncku.edu.tw