A Decomposed Algebraic All Pairs Shortest Path Algorithm *

I-Lin Wang

Department of Industrial and Information Management
National Cheng Kung University, Tainan, Taiwan
ilinwang@mail.ncku.edu.tw

Abstract. This paper presents a new all pairs shortest path algorithm which computes shortest paths for all node pairs in a directed graph. Our algorithm contains three phases similar to LU decomposition in linear algebra and identifies negative cycle if one exists. It is as efficient as Floyd-Warshall algorithm for solving all pairs shortest path problems in a complete graph, and is usually faster than Floyd-Warshall algorithm for solving a multiple pairs shortest path problem.

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 [14, 15] and the dual ascent method [23]. 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 [11, 28] and Carré's [6, 7] algorithms. Recently, Wang et al. [27] 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.

Algebraic shortest path algorithms are closely related to *path algebra* as discussed in [7, 3].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* [11, 28] 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

* This research was partly supported by the National Science Council of Taiwan under Grant NSC92-2213-E006-094.

Ford [12], 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 [23]). Since the same problem can also be viewed as inverting the matrix $(I_n - C)$, the *escalator method* [21] for inverting a matrix corresponds to an inductive APSP algorithm proposed by Dantzig [8]. Finally, the *decomposition algorithm* proposed by Mill [20] (also, Hu [17]) 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 [9]), 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 $X^* = (I_n \oplus C)^{n-1}$. Shimbel [25] 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* [16, 29] and *cascade* [10, 18, 29] algorithms perform only two or three successive distance matrix squarings. However, Farbey et al. [10] 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 [13], or in $O(n^3((\log\log n)/\log n)^{\frac{1}{2}})$ time by Takaoka [26]. 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 [30]. 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. [27], but converges to the optimal solution in different sequence of operations.

FRLU can deal with networks with general arc length (i.e. an arc may have negative length) but without negative cycles. For networks containing a negative cycle, *FRLU* can detect it with fewer operations than Floyd-Warshall algorithm. When solving for shortest distances between several specific node pairs, *FRLU* usually performs fewer operations than Floyd-Warshall algorithm.

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 Algorithm *FRLU* saves computational work in computing shortest distances for multiple pairs shortest path (MPSP) problems. 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 $i \to k_1 \to k_2 \to \ldots \to k_r \to j$

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.

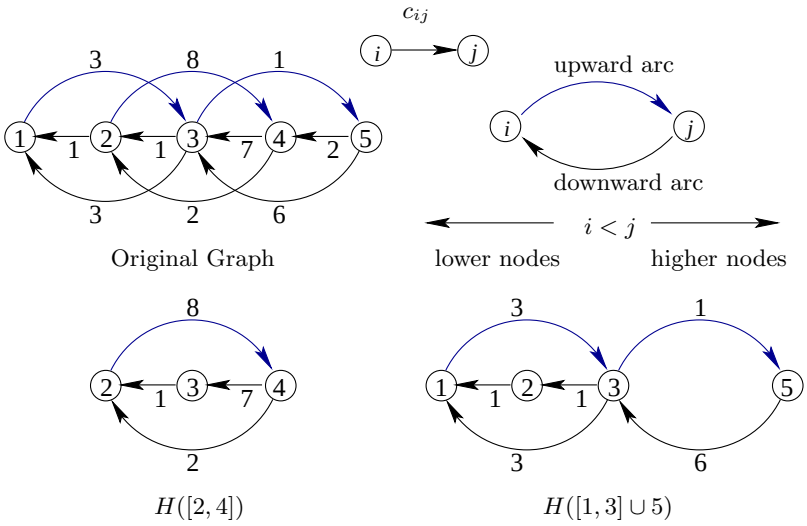

Fig. 1. Illustration of node ordering and subgraphs $H([2,4])$, $H([1,3] \cup 5)$

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 two algebraic algorithms 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, $FRLU$ can save more computational work than other APSP algorithms do.

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

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*

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 $s \neq t$ and $x_{st} > x_{sk} + x_{kt}$ ***then***
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

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 $x_{st} > x_{sk} + x_{kt}$. Figure 2(a) illustrates the operations of *Forward_LU* on a 5-node graph. It sequentially

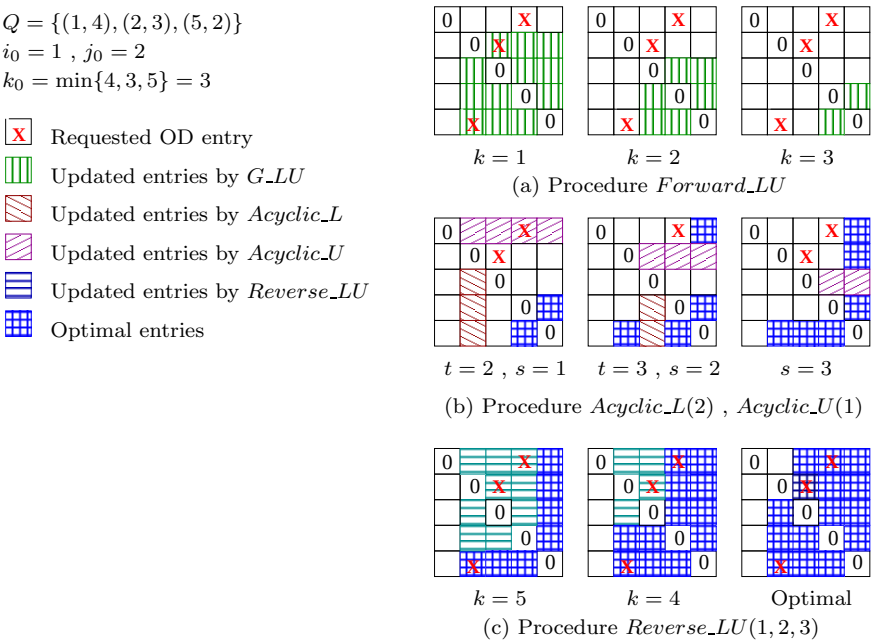

Fig. 2. Solving a 3 pairs shortest path problem on a 5-node graph by Algorithm $FRLU(Q)$

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 2)

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

Proof. See [27].

Corollary 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* $H([1,\min\{s,t\}]\cup \max\{s,t\})$.

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

Theorem 3. *Suppose there exists a p-node cycle* C_p, $i_1 \to i_2 \to i_3 \to \ldots \to i_p \to i_1$, *with negative length. Then, procedure Forward_LU will identify it.*

Proof. See [27].

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 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 5(c))

Procedure Acyclic_U($\mathbf{i_0}$)
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

$Acyclic\_U(i_0)$ 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 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 5(c))

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 such 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, $s\rightarrow v_1\rightarrow v_2\rightarrow\ldots\rightarrow v_{p-2}\rightarrow v_{p-1}\rightarrow t$ 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 with index smaller than $\min\{s,t\}=t<s$, Theorem 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, $\{u_i : i=1,\ldots,q\}$, in this shortest path in G from s to t, and $s:=u_0>u_1>u_2>\ldots>u_{r-1}>u_r>u_{r+1}:=t$. 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 $u_{k-1} \rightarrow u_k$ in G contains intermediate nodes that all have lower indices than u_k. Since Theorem 1 guarantees that $Forward\_LU$ will produce an arc (u_{k-1}, u_k) for any such shortest path segment $u_{k-1} \rightarrow u_k$ and G'_L is acyclic, the original shortest path $s \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{p-2} \rightarrow v_{p-1} \rightarrow t$ in G will be reduced to the shortest path $s \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_{r-1} \rightarrow u_r \rightarrow j$ in G'_L. (b) Using a similar argument to (a) above, the result follows immediately.

Corollary 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 $s \geq i_0$

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

```
Procedure Reverse_LU(i0, j0, k0)
begin
   for k = n down to k0 + 1 do
      for s = k − 1 down to i0 do
         for t = k − 1 down to j0 do
            if s ≠ t and x_st > x_sk + x_kt then
               x_st := x_sk + x_kt; succ_st := succ_sk;
end
```

the operations 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 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 $1 \leq r \leq n$.

Theorem 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 $k \leq (n-k_0)$.*

Corollary 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), $i = 1, \ldots, q$.*

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 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 $k_0 := 2$.*

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 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 [22]. 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 [24]. Nevertheless, many fill-in reducing techniques such as Markowitz's rule [19], minimum degree method, and nested dissection method (see Chapter 8 in [9]) 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 Solving MPSP problems by *FRLU*

MPSP problems usually arise in real-world applications in which only shortest paths between specific node pairs are requested. 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 [24].

5 Conclusions

In this paper we propose a new algebraic APSP algorithm called *FRLU* which can deal with graphs with general arc lengths but without negative cycles. It can also identify the existence of negative cycle in smaller time bound than Floyd-Warshall algorithm. Although *FRLU* has worst case complexity

$O(n^3)$ that is not better than other algebraic APSP algorithms such as Floyd-Warshall and Carré's algorithms, *FRLU* can, in practice, avoid computational work if a good node ordering could be identified in preprocessing to reduce triple comparisons.

First obtaining the shortest distances from or to the last node n, Algorithm $FRLU(i_0, j_0, k_0)$ systematically obtains shortest distances for all node pairs (s, t) such that $s \geq k_0$, $t \geq j_0$ and $s \geq i_0$, $t \geq k_0$ where $k_0 := \min_i\{\max\{s_i, t_i\}\}$. By setting $i_0 := 1$, it can be used to build the shortest path trees rooted at each node $t \geq k_0$. When solving MPSP problems, this algorithm 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 *FRLU* 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 algorithms. Such 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.

In practice, the three phases of *FRLU* can have efficient implementation since it involves acyclic operations on two acyclic subgraphs of the augmented graph induced by its first procedure *Forward_LU*. If the augmented graph is sparse, the second procedure *Acyclic_LU* can be implemented efficiently by topological ordering.

Since solving APSP problems is analogous to inverting a square matrix, we expect methods similar to *FRLU* can also be developed to compute q specific entries for some matrix B^{-1} without inverting the whole matrix B (as do by conventional APSP algorithms), and without computing whole columns of the matrix that cover the requested columns and rows (as do by SSSP algorithms). It remains an open question whether efficiently computing specific entries for the inverse of a matrix is difficult in most cases.

Acknowledgements

I-Lin Wang was partly supported by the National Science Council of Taiwan under Grant NSC92-2213-E006-094.

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, NJ, 1993.
3. R.C. Backhouse and B.A. Carré. Regular algebra applied to path finding problems. *Journal of the Institute of Mathematics and Its Applications*, 15(2):161–186, April 1975.
4. R.E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
5. D.P. Bertsekas. *Network ptimization: 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, pages 388–397, 1969.
7. B.A. Carré. An algebra for network routing problems. *Journal of Institute of Mathematics and Its Applications*, 7:273–294, 1971.
8. G.B. Dantzig. All shortest routes in a graph. In *Theory of Graphs (International Symposium., Rome, 1966)*, pages 91–92. Gordon and Breach, New York, 1967.

9. I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct methods for sparse matrices*. Oxford University Press Inc., New York, 1989.
10. B. A. Farbey, A. H. Land, and J. D. Murchland. The cascade algorithm for finding all shorest distances in a directed graph. *Management Science*, 14(1):19–28, September 1967.
11. R.W. Floyd. Algorithm 97, shortest path. *Comm. ACM*, 5:345, 1962.
12. L.R. Ford Jr. and D.R. Fulkerson. *Flows in networks*. Princeton University Press, Princeton, NJ, 1962.
13. M.L. Fredman. New bounds on the complexity of the shortest path problems. *SIAM Journal on Computing*, 5(1):83–89, March 1976.
14. D. Goldfarb, J. Hao, and S.R. Kai. Efficient shortest path simplex algorithms. *Operations Research*, 38(4):624–628, 1990.
15. D. Goldfarb and Z. Jin. An o(nm)-time network simplex algorithm for the shortest path problem. *Operations Research*, 47(3):445–448, 1999.
16. T.C. Hu. Revised matrix algorithms for shortest paths. *SIAM Journal of Applied Mathematics*, 15:207–218, January 1967.
17. T.C. Hu. A decomposition algorithm for shortest paths in a network. *Operations Research*, 16:91–102, 1968.
18. A.H. Land and S.W. Stairs. The extension of the cascade algorithm to large graphs. *Management Science*, 14:29–33, 1967.
19. H.M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3:255–269, April 1957.
20. G. Mills. A decomposition algorithm for the shortest-route problem. *Operations Research*, 14:279–291, 1966.
21. J. Morris. An escalator process for the solution of linear simultaneous equations. *Philos. Mag. (7)*, 37:106–120, 1946.
22. M. Nakamori. A note on the optimality of some all-shortest-path algorithms. *Journal of the Operations Research Society of Japan*, 15(4):201–204, December 1972.
23. S. Pallottino and M.G. Scutellà. Dual algorithms for the shortest path tree problem. *Networks*, 29(2):125–133, 1997.
24. D.J. Rose and R.E. Tarjan. Algorithmic aspects of vertex elimination on directed graphs. *SIAM Journal on Applied Mathematics*, 34(1):176–197, 1978.
25. A. Shimbel. Applications of matrix algebra to communication nets. *Bulletin of Mathematical Biophysics*, 13:165–178, 1951.
26. T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, 1992.
27. I.L. Wang, E.L. Jophnson, and J.S. Sokol. A multiple pairs shortest path algorithm. *Transportation Science*, 39(4):465–476, Nov 2005.
28. S. Warshall. A theorem on Boolean matrices. *Journal of ACM*, 9:11–12, 1962.
29. L. Yang and W.K. Chen. An extension of the revised matrix algorithm. In *IEEE international Symposium on Circuits and Systems*, pages 1996–1999, Portland, Oregon, May 1989. IEEE.
30. U. Zwick. All pairs shortest pahts in weighted directed graphs - exact and almost exact algorithms. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 310–319, Palo Alto, California, November 1998.