CHAPTER IV

NEW MULTIPLE PAIRS SHORTEST PATH ALGORITHMS

We have reviewed most of the shortest path algorithms in the literature in Chapter 3. Our purpose is to design new algorithms that can efficiently solve the MPSP problem. In this Chapter, we propose two new MPSP algorithms that exploit ideas from Carré's APSP algorithm (see Section 3.4.1.1).

Section 4.1 introduces some definitions and basic concepts. Section 4.2 presents our first new MPSP algorithm ($DLU1$) and proves its correctness. Section 4.3 gives our second algorithm ($DLU2$) and describes why it is superior to the first one. Section 4.4 summarize our work.

4.1 Preliminaries

Based on the definition and notation introduced in Section 3.2, we define the *up-inward arc adjacency list* denoted $ui(i)$ of a node i to be an array that contains all arcs which point upwards into node i, and *down-inward arc adjacency list* denoted $di(i)$ to be an array of all arcs pointing downwards into node i. Similarly, we define the *up-outward arc adjacency list* denoted $uo(i)$ of a node i to be an array of all arcs pointing upwards out of node i, and *down-outward arc adjacency list* denoted $do(i)$ to be an array of all arcs pointing downwards out of node i.

For convenience, if there are multiple arcs between a node pair (i, j), we choose the shortest one to represent arc (i, j) so that c_{ij} in the measure matrix is unique without ambiguity. If there is no path from i to j, then $x_{ij} = \infty$.

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 $[s, t]$ denote the set of nodes $\{s, (s+1), \ldots, (t-1), t\}$. Figure 4 illustrates examples of $H([1, s] \cup t)$ and $H([s, t])$ which will be used to explain our algorithms later on.

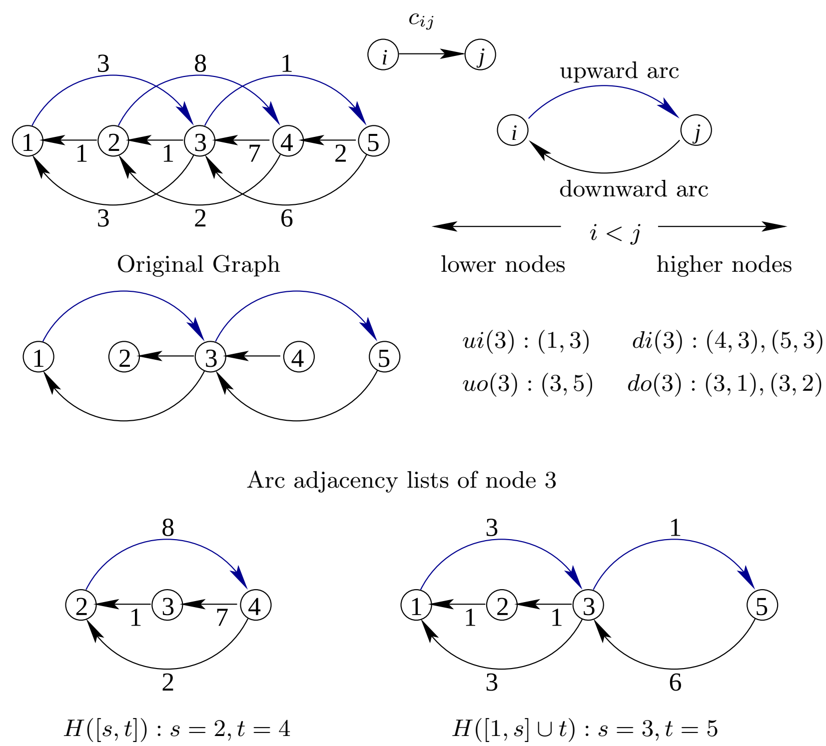

Figure 4: Illustration of arc adjacency lists, and subgraphs $H([2,4])$, $H([1,3] \cup 5)$

Carré's algebraic APSP algorithm [64, 65] uses Gaussian elimination to solve $X = CX \oplus I_n$. After a LU decomposition procedure, Carré's algorithm performs n applications of forward elimination and backward substitution procedures. Each forward/backward operation in turn gives an optimal column solution of X which corresponds to an ALL-1 shortest distance vector. This decomposability of Carré's algorithm makes it more attractive than the Floyd-Warshall algorithm for MPSP problems.

Inspired by Carré's algorithm, we propose two algorithms $DLU1$ and $DLU2$ that further reduce computations required for MPSP problems. We use the name DLU for our algorithms since they contain procedures similar to the LU decomposition in Carré's algorithm and are more suitable for dense graphs. Not only can our algorithms decompose a MPSP problem as Carré's algorithm does, they can also compute the requested OD shortest distances without the need of shortest path trees required by other APSP algorithms.

Therefore our algorithms save computational work over other APSP algorithms and are advantageous for problems where only distances (not paths) are requested. For problems that require tracing of shortest path for a particular OD pair (s,t), $DLU1$ solves the shortest path tree rooted at t as Carré's algorithm does, while $DLU2$ can trace the path without the need of computing that shortest path tree.

For sparse graphs, node ordering plays an important role in the efficiency of our algorithms. A bad node ordering will incur more $fill\text{-}in\ arcs$ which resemble the fill-ins created in Gaussian elimination. Computing an ordering that minimizes the fill-ins is NP-complete [272]. Nevertheless, many fill-in reducing techniques such as Markowitz's rule [230], minimum degree method, and nested dissection method (see Chapter 8 in [98]) used in solving systems of linear equations can be exploited here as well. 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. More details about the impact of node ordering will be discussed in Section 5.2.1. Here, we use a predefined node ordering to start with our algorithms.

The underlying ideas of our algorithms are as follows: Suppose the shortest path in G 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 $H([1,r])$ where $r > \max\{s,t\}$. Including the case where the shortest path is a single arc, our algorithms systematically calculate shortest paths for these cases and obtain the shortest path in G from s to t. When solving the APSP problem on a complete graph, our algorithms 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.

4.2 *Algorithm* $DLU1$

Our first shortest path algorithm $DLU1$ reads a set of q 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 $\min_i\{\max\{s_i, t_i\}\}$. $DLU1$ then computes x^*_{st} for all node pairs (s,t) satisfying $s \geq k_0$, $t \geq j_0$ or $s \geq i_0$, $t \geq k_0$ which covers all the OD pairs in Q. However, the solution does not contain sufficient information to trace their shortest paths, unless i_0 and k_0 are set to be 1 and j_0 respectively in which case $DLU1$ gives shortest path trees rooted at sink node t for each $t = j_0, \ldots, n$.

Algorithm 5 DLU1($\mathbf{Q := \{(s_i, t_i) : i = 1, \ldots, q\}}$)

---

begin
- Initialize: $\forall\ s$, $x_{ss} := 0$ and $succ_{ss} := s$
  - $\forall (s,t)$, ***if*** $(s,t) \in A$ ***then***
    - $x_{st} := c_{st}$; $succ_{st} := t$
    - ***if*** $s < t$ ***then*** add arc (s,t) to $uo(s)$ and $ui(t)$
    - ***if*** $s > t$ ***then*** add arc (s,t) to $do(s)$ and $di(t)$
  - ***else*** $x_{st} := \infty$; $succ_{st} := 0$
  - set $i_0 := \min_i s_i$; $j_0 := \min_i t_i$; $k_0 := \min_i\{\max\{s_i, t_i\}\}$
  - ***if*** shortest paths need to be traced
  - ***then*** reset $i_0 := 1$; $k_0 := j_0$
- G_LU;
- $Acyclic\_L(j_0)$;
- $Acyclic\_U(i_0)$;
- $Reverse\_LU(i_0, j_0, k_0)$;

end

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, Figure 5(a) illustrates the operations of procedure G_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 $[succ_{ij}]$. G_LU computes shortest paths for any node pair (s,t) in $H([1, \min\{s,t\}] \cup \max\{s,t\})$ (as defined in Section 4.1). Note that $x^*_{n,n-1}$ and $x^*_{n-1,n}$ will have been obtained after G_LU.

The second procedure, $Acyclic\_L(j_0)$, computes shortest paths for all node pairs (s,t) such that $s > t \geq j_0$ in $H([1,s])$. Figure 5(b) illustrates the operations of $Acylic\_L(2)$ which updates the column of each entry (s,t) that satisfies $s > t \geq 2$ in the lower triangular part

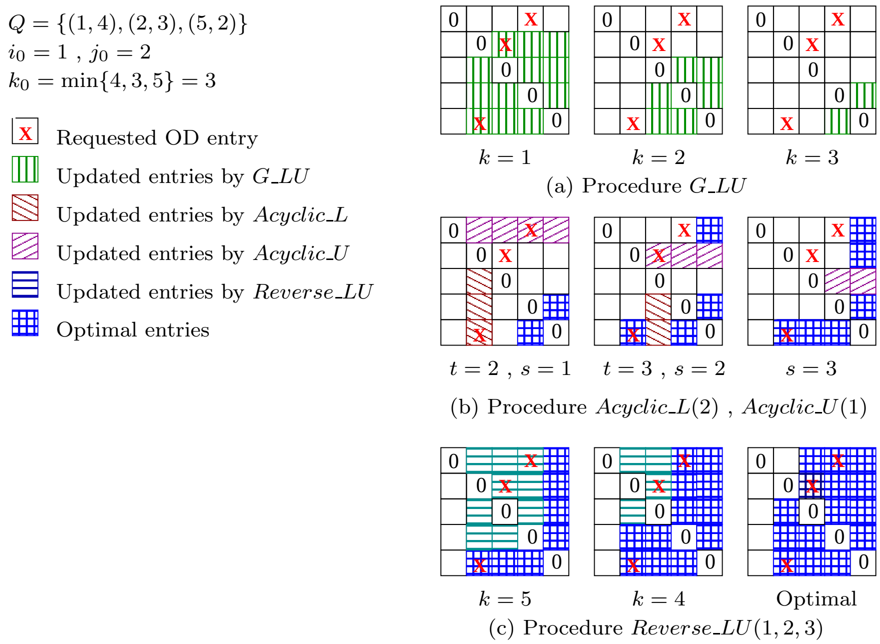

Figure 5: Solving a 3 pairs shortest path problem on a 5-node graph by Algorithm $DLU1(Q)$

of $[x_{ij}]$ and $[succ_{ij}]$ since node 2 is the lowest destination node in Q. Note that x^*_{nj} for all $j \geq j_0$ will have been obtained after $Acyclic\_L(j_0)$.

Similar to the previous procedure, $Acyclic\_U(i_0)$ computes shortest paths for all node pairs (s,t) such that $i_0 \leq s < t$ in $H([1,t])$. Figure 5(b) also illustrates the operations of $Acylic\_U(1)$ which updates the row of each entry (s,t) that satisfies $t > s \geq 1$ in the upper triangular part of $[x_{ij}]$ and $[succ_{ij}]$ since node 1 is the lowest origin node in Q. Note that x^*_{in} for all $i \geq i_0$ will have been obtained after $Acyclic\_U(i_0)$.

By now, for any OD pair (s,t) such that $s \geq i_0$ and $t \geq j_0$ the algorithm will have determined its shortest distance in $H(1, \max\{s,t\})$. The final step, similar to LU decomposition but in a reverse fashion, $Reverse\_LU(i_0, j_0, k_0)$ computes length of the shortest paths in $H([1,r])$ that must pass through node r for each $r = n, \ldots, (\max\{s,t\}+1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ or from each origin $s \geq i_0$ to each destination $t \geq k_0$. The algorithm then compares the x_{st} obtained by the last step with the one obtained previously, and chooses the smaller of the two which corresponds to x^*_{st} in G. Figure 5(c) illustrates the operations of $Reverse\_LU(1,2,3)$ which updates each entry (s,t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $1 \leq s < k$, $2 \leq t < k$. In this case, it runs for $k = 5$ and 4 until entry $(2,3)$ is updated. Note that x^*_{st} for all $s \geq i_0$, $t \geq k_0$ or $s \geq k_0$, $t \geq j_0$ 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.

To trace shortest paths for all the requested OD pairs, we have to set $i_0 = 1$ and $k_0 = j_0$ in the beginning of the algorithm. If $i_0 > 1$, $Acylic\_U(i_0)$ 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 5(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 $DLU1$ gives the shortest distance for the requested OD pairs earlier, tracing these shortest paths requires more computations.

It is easily observed that we can solve any APSP problem by setting $i_0 := 1$, $j_0 := 1$ when applying $DLU1$. For solving general MPSP problems where only shortest distances are requested, $DLU1$ can save more computations without retrieving the shortest path trees, thus makes it more efficient than other algebraic APSP algorithms. More details will be discussed in following sections.

4.2.1 Procedure G_LU

For any node pair (s, t), Procedure G_LU computes shortest path from s to t in $H($ $[1, \min\{s, t\}] \cup \max\{s, t\}$ $)$. The updated x_{st} and $succ_{ij}$ will then be used to determine the shortest distance in $H([1, \max\{s, t\}])$ from s to t by the next two procedures.

Figure 5(a) illustrates the operations of G_LU. It sequentially uses node $k = 1, \ldots, (n-2)$ as 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 $x_{st} > x_{sk} + x_{kt}$.

Thus this procedure is like the LU decomposition in Gaussian elimination. Graphically speaking, it 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 whose index is smaller than both end nodes of

Procedure G_LU
begin
  for $k = 1$ to $n-2$ ***do***
    for each arc $(s,k) \in di(k)$ ***do***
      for each arc $(k,t) \in uo(k)$ ***do***
        if $x_{st} > x_{sk} + x_{kt}$
          if $s = t$ and $x_{sk} + x_{kt} < 0$ ***then***
            Found a negative cycle; ***STOP***
          if $x_{st} = \infty$ ***then***
            if $s > t$ ***then***
              add a new arc (s,t) to $do(s)$ and $di(t)$
            if $s < t$ ***then***
              add a new arc (s,t) to $uo(s)$ and $ui(t)$
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

the path. For example, in Figure 6 we add fill-in arc $(2,3)$ because $2 \to 1 \to 3$ is a shorter path than the direct arc from node 2 to node 3 (infinity in this case). We also add arcs $(3,4)$ and $(4,5)$ and modify the length of original arc $(4,3)$.

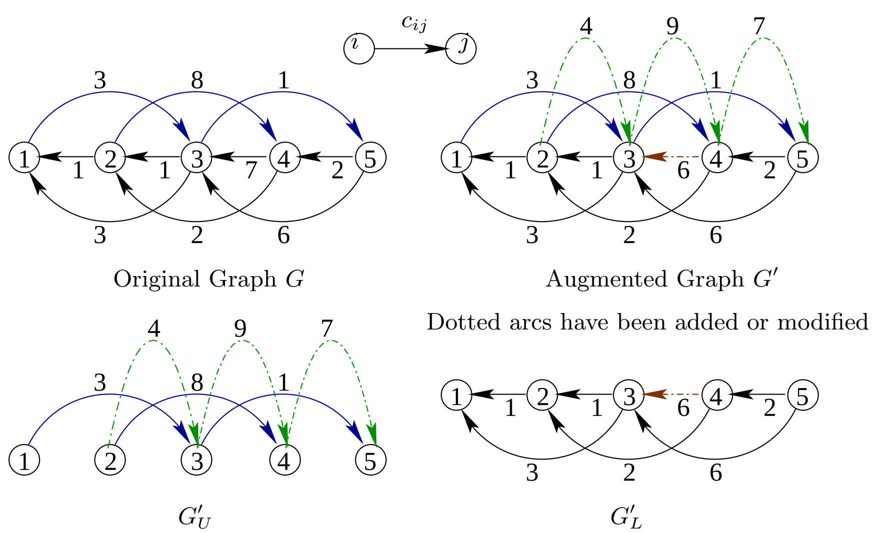

Figure 6: Augmented graph after procedure *G_LU*

If we align the nodes by ascending order of their index from the left to the right, we can easily identify the subgraph G'_L (G'_U) which contains all the downward (upward) arcs of G'. (see Figure 6)

We can initialize the arc adjacency lists $ui(i)$, $di(i)$, $uo(i)$ and $do(i)$ (see Section 4.1) when we read the graph data. If $x_{st} = \infty$ and $x_{sk} + x_{kt} < \infty$ where $k < \min\{s,t\}$, we add a fill-in arc (s, t) to G. The adjacency lists $di(k)$, $do(k)$, $ui(k)$, and $uo(k)$ for node $k > 1$ are updated during the procedure whenever a new arc is added. Given two arcs (s, k) and (k, t), a *triple comparison* $s \to k \to t$ compares $c_{sk} + c_{kt}$ with c_{st}. If there exists no arc (s, t), we add a fill-in arc (s, t) and assign $c_{sk} + c_{kt}$ to be its length; Otherwise, we update its length to be $c_{sk} + c_{kt}$.

G_LU is a sparse implementation of the triple comparisons $s \to k \to t$ for any (s, t) 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 stored as an arc (s, t) in G'. Thus $x_{n,n-1} = x^*_{n,n-1}$ and $x_{n-1,n} = x^*_{n-1,n}$ since $H([1, n-1] \cup n) = G$. *G_LU* can also detect negative cycles.

The complexity of this procedure depends on the topology and node ordering. The number of triple comparisons is bounded by $\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|)$, or $O(n^3)$ in the worst case. It is $\frac{n(n-1)(n-2)}{3}$ on a complete graph. In practice a good node ordering may reduce the number of comparisons for non-complete graphs. Determining a good node ordering is the same as determining a good permutation of columns when solving systems of equations so that the number of fill-ins required is reduced in the LU decomposition. More implementation details of our algorithms regarding sparsity techniques will be covered in Chapter 5.

4.2.2 Procedure *Acyclic_L*(j_0)

After obtaining the shortest paths in $H([1, t])$ from each node $s > t$ to each node t in previous procedure, *Acyclic_L*(j_0) computes their shortest paths in $H([1, s])$ from each node $s > t$ to each node $t \geq j_0$. The updated x_{st} and $succ_{ij}$ will then be used to compare with the shortest distances in $H([1, r])$ for each $r = (s+1), \ldots, n$ from each node $s > t$ to each node $t \geq j_0$ by the last procedure.

This procedure does sequences of shortest path tree computations in G'_L, the acyclic subgraph of augmented graph G' that contains all of its downward arcs (see Section 4.2.1). Its subprocedure, *Get_D_L*(t), resembles the forward elimination in Gaussian elimination. Each application of subprocedure *Get_D_L*(t) gives the shortest distance in G'_L from each

```
Procedure Acyclic_L(j0)
begin
    Initialize: ∀ node k, dô(k) := do(k)
    for t = j0 to n − 2 do
        Get_D_L(t);
end
```

```
Subprocedure Get_D_L(t)
begin
    put node t in LIST
    while LIST is not empty do
        remove the lowest node k in LIST
        for each arc (s, k) ∈ di(k) do
            if s ∉ LIST, put s into LIST
            if x_st > x_sk + x_kt then
                    if x_st = ∞ then
                        add a new arc (s, t) to dô(s)
                x_st := x_sk + x_kt ; succ_st := succ_sk
end
```

node $s > t$ to node t, and we repeat this subprocedure 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 4.2(a) in Section 4.2.5) Also, this procedure gives x^*_{nt}, the shortest distance in G from node n to any node $t \geq j_0$. (see Corollary 4.2(c) in Section 4.2.5)

Figure 5(b) in Section 4.2 illustrates the operations of $Acyclic\_L(j_0)$. Each application of the subprocedure $Get\_D\_L(t)$ updates each entry (s, t) in column t of $[x_{ij}]$ and $[succ_{ij}]$ satisfying $s > t$, to represent the shortest distance in G'_L from node s to node t and the successor of node s in that shortest path.

For each node $k > j_0$, we introduce a new down-outward arc adjacency list (defined in Section 4.1) denoted $\hat{do}(k)$ which is initialized to be $do(k)$ obtained from procedure G_LU. Whenever we identify a path (not a single arc) in G'_L from node $k > t$ to node $t \geq j_0$, we add a new arc (k, t) to $\hat{do}(k)$. This connectivity information will be used by procedure $Reverse\_LU$ for sparse operation and complexity analysis.

The number of triple comparisons is bounded by $\sum_{t=j_0}^{n-2} \sum_{k=t}^{n-1} |di(k)|$, or $O((n-j_0)^3)$ in the worst case. Note that if we choose a node ordering such that j_0 is large, then we may decrease the computational work in this procedure, but such an ordering may make the first procedure *G_LU* and the next procedure $Acyclic\_U(i_0)$ inefficient. When solving an APSP problem, we have to set $j_0 = 1$; thus, a $\sum_{t=1}^{n-2}\sum_{k=t}^{n-1}|di(k)|$ bound is obtained, which will be $\frac{n(n-1)(n-2)}{6}$ on a complete graph. Therefore this procedure is $O(n^3)$ in the worst case.

4.2.3 Procedure $Acyclic\_U(i_0)$

After obtaining the shortest paths in $H([1, s])$ from each node $s < t$ to each node t in procedure *G_LU*, $Acyclic\_U(i_0)$ computes their shortest paths in $H([1, t])$ from each node $s \geq i_0$ to each node $t > s$. The updated x_{st} will then be compared with the shortest distances in $H([1, r])$ for each $r = (t + 1), \ldots, n$ from each node $s \geq i_0$ to each node $t > s$ by the last procedure.

Procedure Acyclic_U($\mathbf{i_0}$)
begin
    Initialize: $\forall$ node k, $\hat{ui(k)} := ui(k)$
    for $s = i_0$ to $n - 2$ ***do***
        $Get\_D\_U(s)$;
end

Subprocedure Get_D_U(s)
begin
    put node s in $LIST$
    while $LIST$ is not empty ***do***
        remove the lowest node k in $LIST$
        for each arc $(k, t) \in uo(k)$ ***do***
            if $t \notin LIST$, put t into $LIST$
            if $x_{st} > x_{sk} + x_{kt}$ ***then***
                    if $x_{st} = \infty$ ***then***
                        add a new arc (s, t) to $\hat{ui(t)}$
                $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

This procedure 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 shortest distance from i to j and the successor of i in that shortest path in G'_U, the acyclic subgraph of augmented graph G' that contains all of its upward arcs (see Section 4.2.1). 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 4.2(b) in Section 4.2.5) Also, this procedure gives x^*_{sn}, the shortest distance in G from any node $s \geq i_0$ to node n. (see Corollary 4.2(c) in Section 4.2.5)

Figure 5(b) in Section 4.2 illustrates the operations of $Acyclic\_U(i_0)$. Each application of the subprocedure $Get\_D\_U(s)$ updates each entry (s,t) in row s of $[x_{st}]$ and $[succ_{st}]$ satisfying $s < t$ which represents the shortest distance in G'_U from node $s < t$ to node t and the successor of node s in that shortest path.

For each node $k > i_0$, we introduce a new up-inward arc adjacency list (defined in Section 4.1) denoted $\hat{ui(k)}$ which is initialized to be $ui(k)$ obtained from procedure G_LU. Whenever we identify a path (not a single arc) in G'_U from node $s \geq i_0$ to node $k > s$, we add a new arc (s,k) to $\hat{ui(k)}$. This connectivity information will be used by procedure $Reverse\_LU$ for sparse operation and complexity analysis.

The number of triple comparisons is bounded by $\sum_{s=i_0}^{n-2} \sum_{k=s}^{n-1} |uo(k)|$, or $O((n-i_0)^3)$ in the worst case. Note that if we choose a node ordering such that i_0 is large, then we may decrease the computational work in this procedure, but such an ordering may make the first procedure G_LU and previous procedure $Acyclic\_L(j_0)$ inefficient. When solving an APSP problem, we have to set $i_0 = 1$; thus, a $\sum_{s=1}^{n-2} \sum_{k=s}^{n-1} |uo(k)|$ bound is obtained, which will be $\frac{n(n-1)(n-2)}{6}$ on a complete graph. Therefore this procedure is $O(n^3)$ in the worst case.

4.2.4 Procedure $Reverse\_LU(i_0, j_0, k_0)$

After previous two procedures, the algorithm will have determined the length of shortest path in $H(1, \max\{s,t\})$ from each node $s \geq i_0$ to each node $t \geq j_0$. $Reverse\_LU(i_0, j_0, k_0)$ then computes shortest distances in $H([1,r])$ for each $r = n, \ldots, (\max\{s,t\}+1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ or from each origin $s \geq i_0$ to each destination $t \geq k_0$. Note that k_0 is set to be $\min_i\{\max\{s_i, t_i\}\}$ so that all the requested OD pairs in Q will be correctly updated by the procedure.

Procedure Reverse_LU($\mathbf{i_0}, \mathbf{j_0}, \mathbf{k_0}$)
begin
    for $k = n$ down to $k_0 + 1$ ***do***
        for each arc $(s, k) \in \hat{ui}(k)$ and $s \geq i_0$ ***do***
            for each arc $(k, t) \in \hat{do}(k)$ and $t \geq j_0$, $s \neq t$ ***do***
                if $x_{st} > x_{sk} + x_{kt}$ ***then***
                    if $x_{st} = \infty$ ***then***
                        if $s > t$ ***then*** add new arc (s, t) to $\hat{do}(s)$
                        if $s < t$ ***then*** add new arc (s, t) to $\hat{ui}(t)$
                    $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Figure 5(c) in Section 4.2 illustrates the operations of *Reverse_LU*(i_0, j_0, k_0). It sequentially uses node $k = n, \ldots, (k_0 + 1)$ as an intermediate node to update each entry (s, t) of $[x_{ij}]$ and $[succ_{ij}]$ that satisfies $i_0 \leq s < k$ and $j_0 \leq t < k$ as long as $x_{sk} < \infty$, $x_{kt} < \infty$ and $x_{st} > x_{sk} + x_{kt}$.

Thus this procedure is similar to the first procedure, *G_LU*, but proceeds in reverse fashion. Since x^*_{sn} and x^*_{nt} for $s \geq i_0$ and $t \geq j_0$ will have been computed by *Acyclic_U*(i_0) and *Acyclic_L*(j_0) respectively, *Reverse_LU*(i_0, j_0, k_0) computes x^*_{sk} and x^*_{kt} for each s satisfying $i_0 \leq s < k$, each t satisfying $j_0 \leq t < k$, and for each $k = (n-1), \ldots, k_0$ where $k_0 := \min_i\{\max\{s_i, t_i\}\}$. In particular, this procedure computes length of shortest paths that must pass through node r in $H([1, r])$ for each $r = n, \ldots, (\max\{s, t\} + 1)$ from each origin $s \geq k_0$ to each destination $t \geq j_0$ or from each origin $s \geq i_0$ to each destination $t \geq k_0$. Since we will have obtained the shortest distances in $H([1, \max\{s, t\}])$ from each node $s \geq i_0$ to each node $t \geq j_0$ from previous procedures, the smaller of these two distances will be the x^*_{st} in G. This procedure stops when shortest distances for all the requested OD pairs are calculated.

Although this procedure calculates all shortest path lengths, not all of the paths themselves are known. To trace shortest paths for all the requested OD pairs by $[succ_{ij}]$, we must 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 the shortest path tree rooted at sink node k. Otherwise, there will exist some $succ_{st}$ with $s < i_0$ or $t < k_0$ that have not been updated by the algorithm and thus are not qualified to provide correct successor information for shortest path tracing purposes.

Note that for each node k, $\hat{do(k)}$ and $\hat{ui(k)}$ may be modified during this procedure, since they represent connectivity between node k and other nodes in G. In particular, if node $s \geq i_0$ can only reach node $t \geq j_0$ via some intermediate node $k > \max\{s, t\}$, this procedure will identify that path and add arc (s, t) to $\hat{do(s)}$ if $s > t$, or $\hat{ui(t)}$ if $s < t$.

When solving an APSP problem on a highly connected graph, their magnitude, $\left|\hat{do(k)}\right|$ and $\left|\hat{ui(k)}\right|$ tend to achieve their upper bound k for each node k. The number of triple comparisons is bounded by $\sum\limits_{k=k_0+1}^{n}(\left|\hat{ui(k)}\right| \cdot \left|\hat{do(k)}\right|)$, or $O((n-k_0)^3)$ in the worst case. Note that if we choose a node ordering such that k_0 is large, then we may decrease computational work in this procedure, but such an ordering may make the first procedure *G_LU* and one of the procedures *Acyclic_L*(j_0) and *Acyclic_U*(i_0) inefficient. This procedure has a time bound $\frac{n(n-1)(n-2)}{3}$ when solving an APSP problem on a complete graph. Therefore this procedure is $O(n^3)$ in the worst case.

4.2.5 Correctness and properties of algorithm $DLU1$

First we show the correctness of the algorithm, and then discuss some special properties of this algorithm. To prove its correctness, we will show how the procedures of $DLU1$ calculate shortest path lengths for various subsets of OD pairs, and then demonstrate that every OD pair must be in one such subset.

We begin by specifying the set of OD pairs whose shortest path lengths will be calculated by Procedure *G_LU*. In particular, *G_LU* will identify shortest path lengths for those requested OD pairs (s, t) whose shortest paths have all intermediate nodes with index lower than $\min\{s, t\}$.

Theorem 4.1. *A shortest path in G from s to t that has a highest node with index equal to* $\min\{s, t\}$ *will be reduced to arc* (s, t) *in G' by Procedure G_LU.*

Proof. Suppose such a shortest path in G from s to t contains p arcs. In the case where $p = 1$ and $r = s$ or t, the result is trivial. Let us consider the case where $p > 1$. That is, $s := v_0 \to v_1 \to v_2 \to \ldots \to v_{p-2} \to v_{p-1} \to v_p := t$ is a shortest path in G from s to t with $(p-1)$ intermediate nodes $v_k < \min\{s, t\}$ for $k = 1, \ldots, (p-1)$.

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

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

Corollary 4.1. *(a) Procedure G_LU will correctly compute $x^*_{n,n-1}$ and $x^*_{n-1,n}$.*
(b) Procedure G_LU will correctly compute a shortest path for any node pair (s,t) in $H([1, \min\{s,t\}] \cup \max\{s,t\})$.

Proof. (a) This follows immediately from Theorem 4.1, because all other nodes have index less than $(n-1)$.

(b) This follows immediately from Theorem 4.1. □

Now, we specify the set of OD pairs whose shortest path lengths will be calculated by Procedure *Acyclic_L*(j_0) and *Acyclic_U*(i_0). In particular, these two procedures will give shortest path lengths for those requested OD pairs (s,t) whose shortest paths have all intermediate nodes with index lower than $\max\{s,t\}$.

Theorem 4.2. *(a) A shortest path in G from node $s > t$ to node t that has s as its highest node corresponds to a shortest path from s to t in G'_L.*
(b) A shortest path in G from node $s < t$ to node t that has t as its highest node corresponds to a shortest path from s to t in G'_U.

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, $s \to v_1 \to v_2 \to \ldots \to v_{p-2} \to v_{p-1} \to t$ is a shortest path in G from node $s > t$ to node t with $(p-1)$ intermediate nodes $v_k < \max\{s,t\} = s$ for $k = 1, \ldots, (p-1)$.

In the case where every intermediate node $v_k < \min\{s,t\} = t < s$, Theorem 4.1 already shows that *G_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 v_k satisfies $s > v_k > t$.

Suppose that r of the p intermediate nodes in this shortest path in G from s to t satisfy $s > u_1 > u_2 > \ldots > u_{r-1} > u_r > t$. Define $u_0 := s$, and $u_{r+1} := t$. We can break this shortest path into $(r+1)$ segments: u_0 to u_1, u_1 to u_2,…, u_r to u_{r+1}. Each shortest path segment $u_{k-1} \to u_k$ in G contains intermediate nodes that all have lower index than u_k. Since Theorem 4.1 guarantees that *G_LU* will produce an arc (u_{k-1}, u_k) for any such shortest path segment $u_{k-1} \to u_k$, the original shortest path that contains p arcs in G actually can be represented as the shortest path $s \to u_1 \to u_2 \to \ldots \to u_{r-1} \to u_r \to j$ in G'_L.

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

Corollary 4.2. *(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 4.2(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 4.1(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). □

Finally, we demonstrate that procedure $Reverse\_LU(i_0, j_0, k_0)$ will correctly calculate all shortest path lengths. In particular, $Reverse\_LU(i_0, j_0, k_0)$ gives shortest path lengths for those requested OD pairs (s, t) whose shortest paths have some intermediate nodes with index higher than $\max\{s, t\}$.

Lemma 4.1. *(a) Any 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) Any shortest path in G from s to t can be determined by 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 4.2(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 4.3. *The k^{th} application of $Reverse\_LU(i_0, j_0, k_0)$ will correctly compute $x^*_{n-k,t}$ and $x^*_{s,n-k}$ for each s satisfying $i_0 \leq s < (n - k)$, and each t satisfying $j_0 \leq t < (n - k)$ where $k \leq (n - k_0)$.*

Proof. After procedures $Acyclic\_L(j_0)$ and $Acyclic\_U(i_0)$, 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 need only to check those shortest paths that must pass through node h for each $h = (\max\{s, t\} + 1), \ldots, n$. By Lemma 4.1(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 4.2(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 4.1(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 $k = \hat{k} < (n - \max\{s,t\})$. That is, at the end of iteration $k = \hat{k}$, *Reverse_LU*(i_0, j_0, k_0) gives $x^*_{n-\hat{k},t}$ and $x^*_{s,n-\hat{k}}$ for each s satisfying $i_0 \leq s < (n-\hat{k})$, and each t satisfying $j_0 \leq t < (n-\hat{k})$. In other words, we will have obtained $x^*_{n-r,t}$ and $x^*_{s,n-r}$ for each $r = 0, 1, \ldots, \hat{k}$, and for all $s \geq i_0$, $t \geq j_0$.

In iteration $k = (\hat{k}+1)$, for each t satisfying $j_0 \leq t < (n-\hat{k}-1)$, $x^{\hat{k}+1}_{n-\hat{k}-1,t} := \min\{x^{\hat{k}}_{n-\hat{k}-1,t}, x^{\hat{k}}_{n-\hat{k}-1,n-\hat{k}} + x^{\hat{k}}_{n-\hat{k},t}\} = \min\{x^{\hat{k}}_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-\hat{k}} + x^*_{n-\hat{k},t}\}$ by assumption of the induction. Note that the first term $x^{\hat{k}}_{n-\hat{k}-1,t}$ has been updated $\hat{k}$ times in the previous $\hat{k}$ iterations. In particular, $x^{\hat{k}}_{n-\hat{k}-1,t} = \min\limits_{0 \leq k \leq (\hat{k}-1)} \{x^0_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-k} + x^*_{n-k,t}\}$ where $x^0_{n-\hat{k}-1,t}$ represents the length of a shortest path in G from node $(n-\hat{k}-1)$ to node t that has node $(n-\hat{k}-1)$ as its highest node. Substituting this new expression of $x^{\hat{k}}_{n-\hat{k}-1,t}$ into $\min\{x^{\hat{k}}_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-\hat{k}} + x^*_{n-\hat{k},t}\}$, we obtain $x^{\hat{k}+1}_{n-\hat{k}-1,t} := \min\limits_{0 \leq k \leq \hat{k}} \{x^0_{n-\hat{k}-1,t}, x^*_{n-\hat{k}-1,n-k} + x^*_{n-k,t}\}$ whose second term $\min\limits_{0 \leq k \leq \hat{k}} \{x^*_{n-\hat{k}-1,n-k} + x^*_{n-k,t}\}$ corresponds to the length of a shortest path in G from node $(n-\hat{k}-1)$ to node t that must pass through some higher node with index $v > (n-\hat{k}-1)$ (v may be $(n-\hat{k}), \ldots, n$). By Lemma 4.1(b) with $r = (n-\hat{k}-1)$, we conclude $x^{\hat{k}+1}_{n-\hat{k}-1,t} = x^*_{n-\hat{k}-1,t}$ for each t satisfying $j_0 \leq t < (n-k)$. Likewise, $x^{\hat{k}+1}_{s,n-\hat{k}-1} = x^*_{s,n-\hat{k}-1}$ for each s satisfying $i_0 \leq s < (n-\hat{k}-1)$ will also be determined in the end of iteration $(\hat{k}+1)$.

By induction, we have shown the correctness of this theorem. $\square$

Corollary 4.3. *(a) Procedure* $Reverse\_LU(i_0, j_0, k_0)$ *will terminate in* $(n - k_0)$ *iterations, and correctly compute* $x^*_{s_i,t_i}$ *for each of the requested OD pair* (s_i, t_i), $i = 1, \ldots, q$

(b) To trace 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* $DLU1$.

(c) Any APSP problem can be solved by Algorithm $DLU1$ *with* $i_0 := 1$*,* $j_0 := 1$*, and* $k_0 := 2$.

Proof. (a) By setting $k_0 := \min_i\{\max\{s_i, t_i\}\}$, 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 4.3. Therefore $Reverse\_LU(i_0, j_0, k_0)$ terminates in $n-(k_0+1)+1 = (n-k_0)$ iterations and correctly computed $x^*_{s_it_i}$ and $succ^*_{s_it_i}$ for each requested OD pair (s_i, t_i) in Q.

(b) 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.

(c) 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. □

Algorithm $DLU1$ can easily identify a negative cycle. In particular, any negative cycle will be identified in procedure G_LU.

Theorem 4.4. *Suppose there exists a k-node cycle* C_k, $i_1 \to i_2 \to i_3 \to \ldots \to i_k \to i_1$*, with negative length. Then, procedure* G_LU *will identify it.*

Proof. Without loss of generality, let i_1 be the lowest node in the cycle C_k, i_r be the second lowest, i_s be the second highest, and i_t be the highest node. Let $length(C_k)$ denote the length function of cycle C_k. Assume that $length(C_k) = \sum_{(i,j) \in C_k} c_{ij} < 0$.

In G_LU, before we begin iteration i_1(using i_1 as the intermediate node), the length of some arcs of C_k might have already been modified, but no arcs of C_k will have been removed nor will $length(C_k)$ have increased. After we finish scanning $di(i_1)$ and $uo(i_1)$, we can identify a smaller cycle C_{k-1} by skipping i_1 and adding at least one more arc (i_k, i_2) to G. In particular, C_{k-1} is $i_k \to i_2 \to \ldots \to i_{k-1} \to i_k$, and $length(C_{k-1}) = length(C_k) - (x_{i_1 i_2} + x_{i_k i_1} - x_{i_k i_2})$. Since $x_{i_k i_2} \leq x_{i_1 i_2} + x_{i_k i_1}$ by the algorithm, we obtain $length(C_{k-1}) \leq length(C_k) < 0$. The lowest-index node in C_{k-1} is now $i_r > i_1$, thus we will again reduce the size of C_{k-1} by 1 in iteration $k = i_r$.

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

To sum up, suppose the shortest path in G 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 case will be solved by G_LU, second case by $Acyclic\_L$ and $Acyclic\_U$, and third case by $Reverse\_LU$. For any OD pair whose shortest path is a single arc, Algorithm $DLU1$ includes it in the very beginning and compares it with all other paths in the three cases discussed previously.

When solving an APSP problem on an undirected graph, Algorithm $DLU1$ can save half of the storage and computational work. In particular, since the graph is symmetric, $ui(k) = do(k)$, and $uo(k) = di(k)$ for each node k. Therefore storing only the lower (or upper) triangular part of the distance matrix $[x_{ij}]$ and successor matrix $[succ_{ij}]$ is sufficient for the algorithm. In addition, Procedure G_LU and $Reverse\_LU(i_0, j_0, k_0)$ can save half of their computation. In particular, procedure $Reverse\_LU(i_0, j_0, k_0)$ can be replaced by $Reverse\_LU(l_0, l_0, k_0)$ where $l_0 := \min\{i_0, j_0\}$ with the modification that it only updates entries of the lower (or upper) triangular part of $[x_{ij}]$ and $[succ_{ij}]$. We can also integrate procedures $Acyclic\_L(j_0)$ and $Acyclic\_U(i_0)$ into one procedure $Acyclic\_L(l_0)$ (or $Acyclic\_U(l_0)$). Thus half of the computational work can be saved. The SSSP algorithm, on the other hand, can not take advantage of this feature of undirected graphs. In particular, we still have to apply any SSSP algorithm $(n-1)$ times, each time for a different source node, to obtain APSP.

For problems on acyclic graphs, we can reorder the nodes so that the upper (or lower) triangle of $[x_{ij}]$ becomes empty. Then we can skip procedure $Acyclic\_U$ (or $Acyclic\_L$).

A good node ordering may practically reduce much computational work, but the best ordering is difficult (NP-complete) to obtain. In general, an ordering that reduces fill-in arcs in the first procedure G_LU may be beneficial for the other three procedures as well. On the other hand, an ordering that makes i_0 or j_0 as large as possible seems to be more advantageous for the last three procedures, but it may create more fill-in arcs in procedure G_LU, which in turn may worsen the efficiency of the last three procedures.

Overall, the complexity of Algorithm $DLU1$ is $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|) + \sum_{t=j_0}^{n-2}\sum_{k=t}^{n-1}|di(k)| + \sum_{s=i_0}^{n-2}\sum_{k=s}^{n-1}|uo(k)| + \sum_{k=k_0+1}^{n}(\left|\hat{ui(k)}\right| \cdot \left|\hat{do(k)}\right|))$ which is $O(n^3)$ in the worst case. When solving an APSP problem on a complete graph, the four procedures G_LU, $Acyclic\_L(1)$, $Acyclic\_U(1)$ and $Reverse\_LU(1, 1, 2)$ will take $\frac{1}{3}, \frac{1}{6}, \frac{1}{6}$ and $\frac{1}{3}$ of the total $n(n-1)(n-2)$ triple comparisons respectively, which is as efficient as Carré's algorithm and Floyd-Warshall's algorithm in the worst case.

Algorithm $DLU1$ is more suitable for dense graphs than for sparse graphs, since the fill-in arcs we add in each procedure might destroy the graph's sparsity. In particular, in the beginning of procedure $Reverse\_LU(i_0, j_0, k_0)$, the distance matrix tends to be very dense, which makes its sparse implementation less efficient.

As stated in Corollary 4.3, Algorithm $DLU1$ obtains shortest distances for all node pairs (s, t) satisfying $s \geq k_0$, $t \geq j_0$ or $s \geq i_0$, $t \geq k_0$ where $k_0 := \min_i\{\max\{s_i, t_i\}\}$. This of course includes all the requested OD pairs in Q which makes it more efficient than other algebraic APSP algorithms since other APSP algorithms usually need to update all the $n \times n$ entries of $[x_{ij}]$ and $[succ_{ij}]$. However, $DLU1$ still does some redundant computations for many other unwanted OD pairs..Such computational redundancy can be remedied by our next algorithm $DLU2$. In addition, $DLU1$ has a major drawback in the computational dependence of x^*_{st} and $succ^*_{st}$ from higher nodes to lower nodes. In particular, to obtain $x^*_{s't'}$, we rely on the availability of $x^*_{st'}$ for each $s > s'$ and $x^*_{st'}$ for each $t > t'$. We propose two ways to overcome this drawback: one is the algorithm $DLU2$ presented in Section 4.3, and the other is a sparse implementation that will appear in Chapter 5.

Another drawback of this algorithm is the necessity of setting $i_0 := 1$ for the traceability of shortest paths. Unfortunately this is inevitable in our algorithms. However, Algorithm $DLU1$ still seems to have an advantage over other algebraic APSP algorithms because it can solve sequences of SSSP problems rather than having to solve an entire APSP problem. Although Algorithm $DLU1$ has to compute a higher rooted shortest path tree to obtain the lower rooted one, our second algorithm $DLU2$ and the other new sparse algorithm in Chapter 5 can overcome such difficulty.

4.3 Algorithm $DLU2$

Our second MPSP algorithm, $DLU2$, not only obtains shortest distances x^*_{st} faster but also traces shortest paths more easily than $DLU1$. Suppose Q is the set of q OD pairs (s_i, t_i) for $i = 1, \ldots, q$. Unlike $DLU1(Q)$ that not only computes x^*_{st} for all (s,t) in Q but also other unrequested OD pairs, $DLU2(Q)$ specifically attacks each requested OD pair in Q after the common LU decomposition procedure G_LU. Thus it should be more efficient, especially for problems where the requested OD pairs are sparsely distributed in the $n \times n$ OD matrix (i.e. only few OD pairs, but with their origin or destination indices scatteredly ordered among $[1, n]$).

To cite an extreme example, suppose we want to compute shortest path lengths on a graph with n nodes for OD pairs $Q = \{(1,2),(2,1)\}$. Suppose we are not allowed to alter the node ordering (otherwise it becomes easy since we can reorder them to be $\{(n-1,n),(n,n-$

2)} which can be solved by one run of G_LU). Then applying Algorithm $DLU1(Q)$, we will end up with solving an APSP problem, which is not efficient at all. On the other hand, Algorithm $DLU2(Q)$ only needs two runs of its procedure Get_D to directly calculate shortest distances for each requested OD pair individually after the common procedure G_LU.

Algorithm 6 DLU2($\mathbf{Q} := \{(\mathbf{s_i}, \mathbf{t_i}) : \mathbf{i} = \mathbf{1}, \ldots, \mathbf{q}\}$)

```
begin
    Initialize: ∀ s, x_ss := 0 and succ_ss := s
                ∀(s,t), if (s,t) ∈ A then
                            x_st := c_st; succ_st := t
                            if s < t then add arc (s,t) to uo(s)
                            if s > t then add arc (s,t) to di(t)
                        else x_st := ∞ ; succ_st := 0
            opt_ij := 0 ∀ i = 1,...,n, j = 1,...,n
            subrow_i := 0 ; subcol_i := 0 ∀ i = 1,...,n
    G_LU;
    set opt_{n,n-1} := 1, opt_{n-1,n} := 1
    for i = 1 to q do
        Get_D(s_i, t_i);
        if shortest paths need to be traced then
            if x_{s_i t_i} ≠ ∞ then
                Get_P(s_i, t_i);
            else there exists no path from s_i to t_i
end
```

Algorithm $DLU2$ improves the efficiency of computing $x^*_{s_it_i}$, $succ^*_{s_it_i}$ and shortest paths. The first procedure G_LU and the two subprocedures, $Get\_D\_U(s_i)$ and $Get\_D\_L(t_i)$, are imported from Algorithm $DLU1$ as in Section 4.2.1, Section 4.2.3, and Section 4.2.2. The new procedure $Get\_D(s', t')$ gives $x^*_{s't'}$ directly without the need of $x^*_{st'}$ for each $s > s'$ and $x^*_{st'}$ for each $t > t'$ as required in Algorithm $DLU1$. Thus it avoids many redundant computations which would be done by $DLU1$.

Figure 7(b) illustrates how Get_D individually solves $x^*_{s_it_i}$ for each requested OD pair (s_i, t_i). For example, to obtain x^*_{23}, it first applies $Get\_D\_U(2)$ to update x_{23}, x_{24}, and x_{25}, then updates x_{43}, and x_{53} by $Get\_D\_L(3)$. Finally it computes $\min\{x_{23}, (x_{24} + x_{43}), (x_{25} + x_{53})\}$ which corresponds to x^*_{23}. On the other hand, Algorithm $DLU1$ requires computations on x^*_{24}, x^*_{43}, x^*_{25} and x^*_{53} which requires more works than $DLU2$.

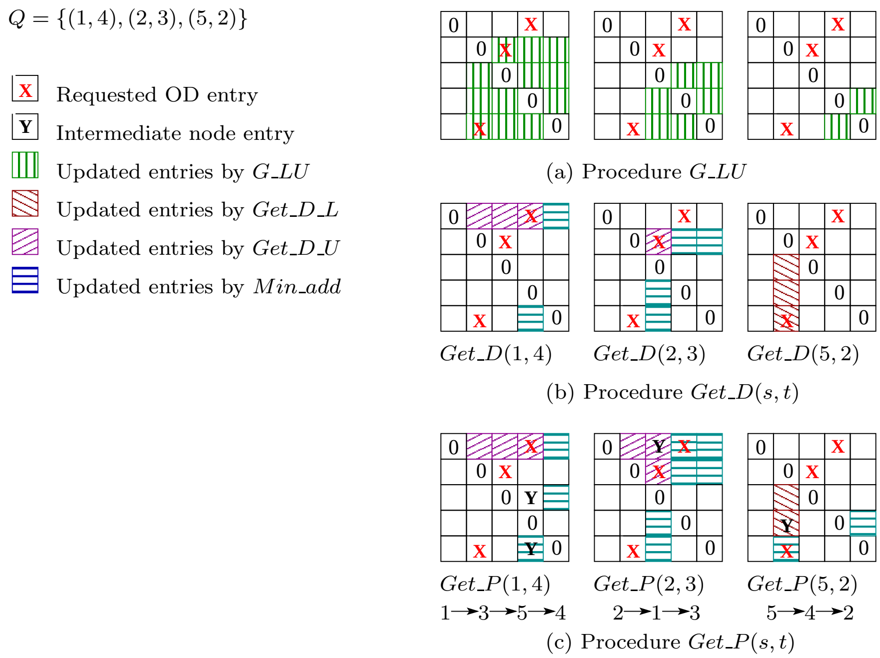

Figure 7: Solving a 3 pairs shortest path problem on a 5-node graph by Algorithm $DLU2(Q)$

If we need to trace the shortest path from s to t, procedure $Get\_P(s,t)$ will iteratively compute all the intermediate nodes in the shortest path from s to t. Therefore, $DLU2$ only does the necessary computations to get the shortest distance and path for the requested OD pair (s,t), whereas $DLU1$ needs to compute the entire shortest path tree rooted at t. For example, suppose $1 \to 3 \to 5 \to 4$ is the shortest path from node 1 to node 4 in Figure 7(c). $DLU2$ first computes x^*_{14} and $succ^*_{14}$. Based on $succ^*_{14} = 3$, which means node 3 is the successor of node 1 in that shortest path, it then computes x^*_{34} and $succ^*_{34} = 5$. Finally it computes x^*_{54} and $succ^*_{54} = 4$, which means node 5 is the last intermediate node in that shortest path. Thus procedure $Get\_P(1,4)$ gives all the intermediate nodes and their shortest distances to the sink node 4. On the other hand, Algorithm $DLU1$ requires additional computations on $succ^*_{24}$ and $succ^*_{25}$.

We introduce a new $n \times n$ optimality indicator array $[opt_{ij}]$ and two $n \times 1$ indicator arrays $[subrow_i]$ and $[subcol_j]$. $opt_{ij} = 1$ indicates that x^*_{ij} and $succ^*_{ij}$ are already obtained, and 0 otherwise. $subrow_i = 1$ if the subprocedure $Get\_D\_U(i)$ has already been run, and 0 otherwise. $subcol_j = 1$ if the subprocedure $Get\_D\_L(j)$ has already been run, and 0 otherwise. Each application of $Get\_D\_U(i)$ gives shortest distances in G'_U from node i to each node $t > i$, and $Get\_D\_L(j)$ gives shortest distances in G'_L from each node $s > j$ to node j. These indicator arrays are used to avoid repeated computations in procedure Get_D. For example, to compute x^*_{23} and x^*_{13} in Figure 7(c), we only need one application of $Get\_D\_L(3)$ which updates x_{43} and x_{53}. Since the updated x_{43} and x_{53} can also be used in computing x^*_{23} and x^*_{13}, setting $subcol_3 := 1$ will avoid repeated applications of $Get\_D\_L(3)$.

To obtain a shortest path tree rooted at sink node t, we set $Q := \{(i,t) : i \neq t,\ i = 1,\ldots,n\}$. Thus setting $Q := \{(i,j)\ i \neq j,\ i = 1,\ldots,n,\ j = 1,\ldots,n\}$ is sufficient to solve an APSP problem. To trace shortest paths for q specific OD pairs $Q := \{(s_i,t_i) : i = 1,\ldots,q\}$, $DLU2(Q)$ can compute these q shortest paths by procedure Get_P without building q shortest path trees as required in Algorithm $DLU1$. If, however, only shortest distances are requested, we can skip procedure Get_P and avoid many unnecessary computations. More details will be discussed in following sections.

4.3.1 Procedure $Get\_D(s_i, t_i)$

This procedure can be viewed as a decomposed version of the three procedures $Acyclic\_L(j_0)$, $Acyclic\_U(i_0)$, and $Reverse\_LU(i_0, j_0, k_0)$ in Algorithm $DLU1$. Given an OD pair (s_i, t_i), it will directly compute its shortest distance in G without the need of all entries x^*_{st} satisfying $s > s_i$ and $t > t_i$ as required by Algorithm $DLU1$.

Procedure $\mathbf{Get\_D(s_i, t_i)}$
begin
    if $opt_{s_it_i} = 0$ ***then***
        if $subcol_{t_i} = 0$ ***then*** $Get\_D\_L(t_i)$; $subcol_{t_i} := 1$
        if $subrow_{s_i} = 0$ ***then*** $Get\_D\_U(s_i)$; $subrow_{s_i} := 1$
        $Min\_add(s_i, t_i)$;
end

Subprocedure $\mathbf{Min\_add(s_i, t_i)}$
begin
    $r_i := \max\{s_i, t_i\}$
    for $k = n$ down to $r_i + 1$ ***do***
        if $x_{s_it_i} > x_{s_ik} + x_{kt_i}$ ***then***
            $x_{s_it_i} := x_{s_ik} + x_{kt_i}$; $succ_{s_it_i} := succ_{s_ik_i}$
    $opt_{s_it_i} := 1$
end

Subprocedure $Get\_D\_L(t_i)$ gives the shortest distance in G'_L from each node $s > t_i$ to t_i, which corresponds to entries x_{st_i} obtained by $Acyclic\_L(j_0)$ in column t_i for each $s > t_i$. $Get\_D\_U(s_i)$ gives shortest distance in G'_U from node s_i to each node $t > s_i$, which corresponds to entries x_{s_it} obtained by $Acyclic\_U(i_0)$ in row s_i for each $t > s_i$.

In $DLU1$, to compute $x^*_{s_it_i}$ for OD pair (s_i, t_i), we have to apply $Reverse\_LU(s_i, t_i, r_i)$ where $r_i := \max\{s_i, t_i\}$, which not only updates $x^*_{s_it_i}$ but also updates other entries in the $(n-s_i)\times(n-t_i)$ submatrix x^*_{st} for all (s,t) satisfying $s_i < s < n$ and $t_i < t < n$. On the other hand in $DLU2$, the subprocedure $Min\_add(s_i, t_i)$ gives $x^*_{s_it_i}$ by min-addition operations only on entries x_{s_ik} and x_{kt_i} for each $k = (r_i + 1), \ldots, n$. Therefore $Min\_add(s_i, t_i)$ avoids more unnecessary updates than $Reverse\_LU(i_0, j_0, k_0)$.

For each requested OD pair (s_i, t_i), the number of triple comparisons in $Get\_D(s_i, t_i)$ is bounded by $\sum\limits_{k=t_i}^{n-1} |di(k)| + \sum\limits_{k=s_i}^{n-1} |uo(k)| + \sum\limits_{k=\max\{s_i,t_i\}+1}^{n} (1)$, or $O((n - \min\{s_i, t_i\})^2)$ in the worst case. As discussed in $DLU1$, reordering the node indices such that s_i and t_i are as large as possible may potentially reduce the computational work of Get_D. However, this may incur more fill-ins and make both G_LU and Get_D less efficient. Overall, when solving a MPSP problem of q OD pairs, this procedure will take $\sum\limits_{i=1}^{q} (\sum\limits_{k=t_i}^{n-1} |di(k)| + \sum\limits_{k=s_i}^{n-1} |uo(k)| + (n-\max\{s_i, t_i\}))$ triple comparisons which is $O(qn^2)$ time in the worst case. Thus in general it is better than $O(n^3)$, the complexity of last three procedures of Algorithm $DLU1$. Note that in the case where $q \approx O(n^2)$, this procedure will take $O(\min\{qn^2, n^3\})$ since at most we have to apply Get_D_L and Get_D_U n times, which takes $O(n^3)$, and Min_add $n(n-1)$ times, which takes $O(n^3)$ as well.

In the worst case when solving an APSP problem on a complete graph, $DLU2$ will perform $\frac{n(n-1)(n-2)}{6}$ triple comparisons on each subprocedure Get_D_L and Get_D_U as does $DLU1$. Its subprocedure Min_add will have $2\cdot\sum\limits_{i=1}^{n-1}\sum\limits_{j=1,j<i}^{n-1}(n-\max\{i,j\}) = \frac{n(n-1)(n-2)}{3}$ times of triple comparisons, which is the same as $Reverse\_LU(i_0, j_0, k_0)$.

Therefore $DLU2$ is as efficient as $DLU1$ in the worst case, but should be more efficient in general when number of OD pairs q is not large.

4.3.2 Procedure $Get\_P(s_i, t_i)$

This procedure iteratively calls procedure $Get\_D(k, t_i)$ to update x_{kt_i} and $succ_{kt_i}$ for any node k that lies on the shortest path from s_i to t_i. Thus given an OD pair (s_i, t_i), it will directly compute the shortest distance label and successor for each intermediate node on its shortest path in G, avoiding computations to other nodes that would be required in Algorithm $DLU1$.

Procedure Get_P($\mathbf{s_i}, \mathbf{t_i}$)
begin
    let $k := succ_{s_i t_i}$
    while $k \neq t_i$ ***do***
        $Get\_D(k, t_i)$;
        let $k := succ_{kt_i}$
end

Starting from the successor of s_i, we check whether it coincides with the destination t_i. If not, we update its shortest distance and successor, and then visit the successor. We iterate this procedure until eventually the destination t_i is encountered. Thus the entire shortest path is obtained since each intermediate node on this path has correct shortest distance and successor by the correctness of procedure *Get_D* (see Theorem 4.5(a) in Section 4.3.3).

On the other hand, in order to trace shortest paths for OD pairs in Q by Algorithm $DLU1$, we have to obtain the shortest path trees rooted at distinct destination nodes in Q, which is better than applying the Floyd-Warshall algorithm, but is still an "over-kill". Here in $DLU2$, procedure *Get_P* simply does the necessary computational work to retrieve each intermediate node lying on the shortest path. Therefore it resolves the computational redundancy problem of $DLU1$ and should be more efficient.

For a particular OD pair (s_i, t_i), obtaining $x^*_{s_i t_i}$ takes $O((n - \min\{s_i, t_i\})^2)$. It also makes $subcol_{t_i} = 1$ so that later each application of $Get\_D(s, t_i)$ only takes $\sum\limits_{k=s}^{n-1} |uo(k)| + (n - \max\{s_i, t_i\})$ which is $O((n - s)^2)$ for each node s lying on the path from s_i to t_i. Suppose this shortest path contains p arcs and has lowest intermediate node i_0. Then $Get\_P(s_i, t_i)$ takes at most $O(p(n - i_0)^2)$ time. This is better than $DLU1$ since $DLU1$ requires $Acyclic\_L(t_i)$, $Acyclic\_U(1)$ and $Reverse\_LU(1, t_i, t_i\})$ which are $O(\sum\limits_{t=t_i}^{n-2} \sum\limits_{k=t}^{n-1} |di(k)|$

$+\sum_{s=1}^{n-2}\sum_{k=s}^{n-1}|uo(k)| + \sum_{k=t_i+1}^{n}(\left|\hat{ui(k)}\right| \cdot \left|\hat{do(k)}\right|)) = O((n-1)^3)$ and is dominated by procedure $Acyclic\_U(1)$. In the worst case where $p = (n-1)$ and $i_0 = 1$, procedure $Get\_P(s_i, t_i)$ takes $O(n^3)$ time.

When solving an APSP problem, complexity of Get_P bound remains $O(n^3)$ since it at most applies Get_D_L and Get_D_U n times, which takes $O(n^3)$ time, while the $Min\_add(s,t)$ for each $s = 1, \ldots, n$ and $t = 1, \ldots, n$ takes $O(n^3)$ time as well.

4.3.3 Correctness and properties of algorithm $DLU2$

First we show the correctness of this algorithm, then discuss some special properties.

Theorem 4.5. *(a) Procedure $Get\_D(s_i, t_i)$ will correctly compute $x^*_{s_it_i}$ and $succ^*_{s_it_i}$ for a given OD pair (s_i, t_i)*

(b) Procedure $Get\_P(s_i, t_i)$ will correctly compute $x^*_{st_i}$ and $succ^*_{st_i}$ for any node s that lies on the shortest path in G from node s_i to node $t_i \geq j_0$.

Proof. (a) After subprocedures $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, we will have obtained shortest paths in $H([1, r_i])$ from s_i to t_i where $r_i := \max\{s_i, t_i\}$. To obtain the shortest path in G from s_i to t_i, we only need to check those shortest paths from s_i to t_i that have highest node h for each $h = (r_i + 1), \ldots, n$. By Lemma 4.1(a), such a shortest path can be decomposed into two segments: from s_i to h and from h to t_i. Note that their shortest distances, x_{s_ih} and x_{ht_i}, will have been calculated by $Get\_D\_U(s_i)$ and $Get\_D\_L(t_i)$, respectively. Note also that their sum, $x_{s_ih} + x_{ht_i}$, represents the shortest distance in $H([1, h])$ from s to t among paths that pass through h.

Procedure $Get\_D(s_i, t_i)$ updates $x_{s_it_i} := \min_{h>r}\{x_{s_it_i}, x_{s_ih} + x_{ht_i}\}$. Now we will show this value corresponds to $x^*_{s_it_i}$. First we consider the case where $h = (r_i + 1)$. Since $x_{s_it_i}$ is the length of a shortest path in $H([1, r_i])$ and $x_{s_ir_i} + x_{r_it_i}$ represents the shortest distance in $H([1, r_i])$ from s_i to t_i among paths that pass through r_i, $\min\{x_{s_it_i}, x_{s_ir_i} + x_{r_it_i}\}$ therefore corresponds to the shortest distance in $H([1, r_i])$ from s_i to t_i. Applying the same argument on $\min\{x_{s_it_i}, x_{s_ih} + x_{ht_i}\}$ for $h = (r_i + 2), \ldots, n$, shows that this value will correspond to the shortest distance in $H([1, n])$ from s to t, which is in fact $x^*_{s_it_i}$, the shortest distance in G from s to t.

Similarly we can show the successor $succ_{s_i t_i}$ is correctly updated in this procedure.

(b) Since we only apply $Get\_P(s_i, t_i)$ when $x^*_{s_i t_i} < \infty$, the shortest path from s_i to t_i is well defined. Whenever an intermediate node k that lies on the shortest path from s_i to t_i is encountered, $Get\_D(k, t_i)$ will return $x^*_{kt_i}$ and $succ^*_{kt_i}$. Then the procedure goes on to the successor of node k. By induction, when t_i is eventually encountered, we will have updated $x^*_{kt_i}$ and $succ^*_{kt_i}$ for each node k on the shortest path from s_i to t_i. $\square$

Like in Algorithm $DLU1$, we can save half of the storage and computational burden when applying Algorithm $DLU2$ to solve an APSP problem on an undirected graph. Although $DLU2$ introduces one $n \times n$ array $[opt_{ij}]$ and two $n \times 1$ array $[subrow_i]$ and $[subcol_j]$ than $DLU1$, it does not need arc adjacency arrays $ui(i)$, $\hat{ui}(i)$, $do(i)$ and $\hat{do}(i)$ for each node i which saves up to $2n^2$ storage. Thus in terms of storage requirement, $DLU2$ requires approximately the same storage as does $DLU1$.

Overall, the complexity of Algorithm $DLU2(Q)$ is $O(\sum\limits_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|) + \sum\limits_{i=1}^{q}(\sum\limits_{k=t_i}^{n-1}|di(k)| + \sum\limits_{k=s_i}^{n-1}|uo(k)| + (n - \max\{s_i, t_i\})) + O(qp(n-i_0)^2))$ where q is number of requested OD pairs, p is the maximal number of arcs contained among all the requested shortest paths, and i_0 is the lowest intermediate node appeared among all the requested shortest paths. This time bound is $O(n^3)$ in the worst case, mostly contributed by procedure G_LU when $q < n^2$. When solving an APSP problem on a complete graph, we can skip procedure Get_P. The other two procedures, G_LU and Get_D, will take $\frac{1}{3}$ and $\frac{2}{3}$ of the total $n(n-1)(n-2)$ triple comparisons respectively, which is as efficient as Algorithm $DLU1$ and the Floyd-Warshall algorithm in the worst case.

All the discussion of node ordering in $DLU1$ also applies to $DLU2$. In other words, techniques to reduce the fill-in arcs in G_LU or to make the indices of the requested origin and destination nodes as large as possible can similarly reduce the computational work of $DLU2$.

In general, when solving a MPSP problem of $q < n^2$ OD pairs, Algorithm $DLU2$ saves more computational work in the last procedures than Algorithm $DLU1$. Unlike $DLU1$ which has to compute the full shortest path tree rooted at t so that the shortest path for a specific OD pair (s, t) can be traced, $DLU2$ can retrieve such a path by successively trespassing each intermediate node on that path, and thus it is more efficient.

4.4 Summary

In this chapter we propose two new algorithms called $DLU1$ and $DLU2$ that are suitable for solving MPSP problems. Although their worst case complexity $O(n^3)$ is no more efficient than other algebraic APSP algorithms such as Floyd-Warshall [113, 304] and Carré's [64, 65] algorithms, our algorithms can, in practice, avoid significant computational work in solving MPSP problems.

First obtaining the shortest distances from or to the last node n, algorithm $DLU1(i_0, j_0, k_0)$ systematically obtains shortest distances for all node pairs (s, t) such that $s \geq k_0$, $t \geq j_0$ or $s \geq i_0$, $t \geq k_0$ 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 $DLU1$ 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 require 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 favorable distribution and/or the required number of fill-in arcs is decreased.

Algorithm $DLU2$ attacks each requested OD pair individually, so it is more suitable for problems with a scattered OD distribution. It also overcomes the computational inefficiency that algorithm $DLU1$ has in tracing shortest paths. It is especially efficient for solving a special MPSP problem of n OD pairs (s_i, t_i) that correspond to a matching. That is, each node appears exactly once in the source and sink node set but not the same time. Such an MPSP problem requires as much work as an APSP problem using most of the shortest path algorithms known nowadays, even though only n OD pairs are requested.

Our algorithms (especially $DLU2$) are advantageous when only the shortest distances between some OD pairs are required. For a graph with fixed topology, or a problem with fixed requested OD pairs where shortest paths have to be repeatedly computed with different numerical values of arc lengths, our algorithms are especially beneficial since we may do a preprocessing step in the beginning to arrange a good node ordering that favors our algorithms. These problems appear often in real world applications. For example, when solving the origin-destination multicommodity network flow problem (ODMCNF) using Dantzig-Wolfe decomposition and column generation[36], we generate columns by solving sequences of shortest path problems between some fixed OD pairs where the arc cost changes in each stage but the topology and requested OD pairs are both fixed. Also, in the computation of parametric shortest paths where arc length is a linear function of some parameter, we may solve shortest distances iteratively on the same graph to determine the critical value of the parameter.

Our algorithms can deal with graphs containing negative arc lengths and detect negative cycles as well. The algorithms save storage and computational work for problems with special structures such as undirected or acyclic graphs.

Although we have shown the superiority of our algorithms over other algebraic APSP algorithms, it is, however, still not clear how our algorithms perform when they are compared with modern SSSP algorithms empirically. Like all other algebraic algorithms in the literature, our algorithms require $O(n^2)$ storage which makes them more suitable for dense graphs. We introduced techniques of sparse implementation that avoid nontrivial triple comparisons, but they come with the price of extra storage for the adjacency data structures.

A more thorough computational experiment to compare the empirical efficiency of $DLU1$ and $DLU2$ with many modern SSSP and APSP algorithms is conducted in Chapter 5, in which we introduce additional sparse implementation techniques that lead to promising computational results.