CHAPTER V

IMPLEMENTING NEW MPSP ALGORITHMS

Many efficient SSSP and APSP algorithms and their implementations have been proposed in the literature. However, only few of them are targeted to solving the MPSP problems or problems with fixed topology but changeable arc lengths or requested OD pairs. These problems of course can be solved by repeated SSSP algorithms. Other methods such as the LP-based reoptimization algorithms (see Section 3.5.2) take advantage of previous optimal shortest paths and perform either primal network simplex methods (when arc lengths change) or dual network simplex methods (when OD pairs change). More recent computational experiments [58] indicate these reoptimization algorithms are still inferior to the repeated SSSP algorithms which repeatedly solve shortest path trees for different requested origins (or destinations, depending on which one has smaller cardinality).

A practical implementation of Carré's algorithm [65] by Goto et al. [150] tries to exploit the sparseness and topology of the networks. For networks with fixed topology, their implementation first does a preprocessing procedure to identify a good node pivoting order so that the fill-ins in the LU decomposition phase are decreased. To avoid unnecessary triple comparisons, they record all the nontrivial triple comparisons in the LU decomposition, forward elimination and backward substitution phases, and then generate an ad hoc APSP code. Their method only stores $O(m)$ data structures, which is the same as other combinatorial SSSP algorithms but is better than $O(n^2)$ as required in general algebraic algorithms. However, this comes with the price of storing the long codes of nontrivial triple comparisons, and may require more total hardware storage. Even worse, the long codes may not be compilable for some compilers.

In particular, we have tested a 1025-node, 6464-arc sparse graph and generated a $500MB$ long code using the code generation algorithm of Goto et al. [150]. The code we generated could not be compiled even on a fast Sun workstation with $1GB$ memory using *gcc*, a C

compiler by GNU, with optimization tags. If instead we only store the arc index of all the triple comparisons, the code will be short but still need temporary storage around $200MB$ to store these indices. Therefore, code generation is not practical for large networks.

In this chapter, we observe that Carré's algorithm can be implemented combinatorially which resolves the need of a huge storage quota by the code generation. Furthermore, we can decompose and truncate Carré's algorithm according to the indices of the distinct origins/destinations of the requested OD pairs. Section 5.1 introduces some notation and definitions appearing in this chapter. Section 5.2 describes major procedures of our algorithm SLU, a sparse combinatorial implementation of Carré's algorithm. Section 5.3 gives detailed implementation issues, and techniques for speeding up SLU. Implementations of algorithm $DLU2$, our proposed MPSP algorithm that appeared in Section 4.3, are given in Section 5.4. Computational experiments including a sparse implementation of the Floyd-Warshall algorithm, many state-of-the-art SSSP codes written by Cherkassky et al. [74], and networks that we generate, are presented in Section 5.5. Section 5.6 shows results of our computational experiments and draws conclusions.

5.1 Notation and definition

Most of the notation and definitions appearing in this chapter can be found in previous chapters. In particular, see Section 4.2.1 for the definition of the augmented graph $G'$and its induced subgraphs G'_L and G'_U, triple comparison, and fill-in arcs. See Section 4.1 for the definition of arc adjacency lists $ui(i)$, $uo(i)$, $di(i)$, and $do(i)$, and the subgraph denoted by $H(S)$ induced on the node set S. See Section 3.4.1.1 for Carré's algorithm, Section 4.2 for algorithm $DLU1$, and Section 4.3 for algorithm $DLU2$.

Let A' denote the arc set of G' with cardinality $|A'| = m'$. Let $[a_{ij}]$ denote a $n \times n$ matrix with m' nonzero entries, each of which represents the arc index of an arc in G'. We create four arrays of size m', $head(a_{ij})$, $tail(a_{ij})$, $c(a_{ij})$ and $succ(a_{ij})$, to store the head, tail, arc length and successor for each arc $(i, j) \in A'$ with index a_{ij}.

Instead of maintaining the $n \times n$ distance matrix $[x_{ij}]$ and successor matrix $[succ_{ij}]$ as algorithm DLU does in Section 3.2, for each distinct destination node j, algorithm SLU uses

two n dimensional vectors $d_n(i)$ and $succ_n(i)$ to denote the shortest distance from each node i to node j and the successor of each node i in the shortest path to node j, respectively. A $n \times 1$ indicator vector $label(i)$ indicates whether a node i is labeled ($label(i) = 1$) or not ($label(i) = 0$). After all the requested shortest distance lengths and successors for the requested OD pairs with the same destination node j have been computed, $d_n(i)$ and $succ_n(i)$ will be reset for the next destination node.

5.2 *Algorithm* SLU

Algorithm SLU can be viewed as an efficient sparse implementation of Carré's algorithm which resembles Gaussian elimination. The algorithm first performs a preprocessing procedure, $Preprocess$, to determine a good node ordering to reduce the number of fill-ins created by LU decomposition. Using the new node ordering, the topology (i.e., $ui(k)$, $uo(k)$, $di(k)$, and $do(k)$ for each node k) of the augmented graph G' is symbolically created. Suppose we want to compute shortest path lengths for a set of OD pairs $Q = \{(s_i, t_i) : i = 1, \ldots, q\}$ which contain $\hat{q}$ distinct destination nodes. For each distinct destination node $\hat{t}_i$, $Preprocess$ also determines the lowest indexed origin node $s_{\hat{t}_i}$ that appears in Q. That is, for each distinct $\hat{t}_i$, $s_{\hat{t}_i} := \min_i\{s_i : (s_i, \hat{t}_i) \in Q\}$.

Algorithm 7 SLU($\mathbf{Q} := \{(\mathbf{s_i}, \mathbf{t_i}) : \mathbf{i} = \mathbf{1}, \ldots, \mathbf{q}\}$)

begin
  $Preprocess$;
  G_LU0;
  for each distinct destination node $\hat{t}_i$ ***do***
    reset $label(k) = 0$, $d_n(k) = M$, $succ_n(k) = 0$ $\forall k \in N \setminus \{\hat{t}_i\}$; $d_n(\hat{t}_i) = 0$
    $G\_Forward(\hat{t}_i)$;
    $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$;
end

Based on the topology of G', SLU does a Gaussian elimination procedure G_LU0, and $\hat{q}$ iterations of procedure $G\_Forward(\hat{t}_i)$ followed by procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. All these three major procedures are "combinatorial" in a sense that they only operate on nodes and arcs of G'. In particular, for nodes $k = 1, \ldots, (n-2)$, G_LU0 scans each arc of $di(k)$ (with tail node s) and each arc of $uo(k)$ (with head node t) and updates the length and successor of arc (s, t) in G'. Procedure $G\_Forward(\hat{t}_i)$ can be viewed as computing

shortest path lengths from each node $s > \hat{t}_i$ to node $\hat{t}_i$ on the acyclic induced subgraph G'_L. Based on the distance label computed by Procedure $G\_Forward(\hat{t}_i)$ for each node $s > \hat{t}_i$, procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ computes shortest distance lengths in G'_U for nodes $s = (n-1), \ldots, s_{\hat{t}_i}$. Thus, unlike other shortest path algorithms that work on the original graph G, algorithm SLU works on the augmented graph G'. The sparser the augmented graph G' is, the more efficient SLU becomes.

After application of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, we have computed the shortest distance $d^*_{\hat{t}_i}(s)$ for every node pair $(s, \hat{t}_i)$ satisfying $s \geq s_{\hat{t}_i}$. Therefore after $\hat{q}$ iterations of $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ for each distinct destination node $\hat{t}_i$, algorithm SLU will give the shortest distance $d^*_{t_i}(s_i)$ for all the requested OD pairs (s_i, t_i) in Q.

5.2.1 Procedure *Preprocess*

This procedure determines the topology of the augmented graph G' by first determining a good node ordering and then performing a symbolic run of the procedure G_LU0 which determines all the arc adjacency data structures of G' (i.e., $di(k)$, $uo(k)$ and $ui(k)$ for each node k). In particular, G_LU0 requires $di(k)$ and $uo(k)$, $G\_Forward$ requires $di(k)$, and $G\_Backward$ requires $ui(k)$ for each node k. $di(k)$, $uo(k)$ and $ui(k)$ store the indices of the arcs that point up-inwards, down-inwards and up-outwards for each node k. These arc indices can be computed beforehand in *Preprocess* so that only $O(m')$ entries are required instead of $O(n^2)$ entries.

Procedure Preprocess
begin
  Decide a node ordering $perm(k)$ for each node k;
  Symbolic execution of procedure G_LU to determine the arc adjacency list
    $di(k)$, $uo(k)$, and $ui(k)$, for each node k of the augmented graph G' ;
  for each distinct destination node $\hat{t}_i$ ***do***
    if shortest paths need to be traced ***then***
      set $s_{\hat{t}_i} := 1$
    else set $s_{\hat{t}_i} := \min_i \{s_i : (s_i, \hat{t}_i) \in Q\}$
  Initialize: $\forall (s,t) \in A'$, ***if*** $(s,t) \in A$ with index a_{st} ***then***
    $c(a_{st}) := c_{st}$; $succ(a_{st}) := t$
    else $c(a_{st}) := M$; $succ(a_{st}) := 0$
end

The node ordering may be computed by many common techniques used in linear algebra to reduce fill-ins in LU decomposition. Most of the ordering techniques are based on the rationale of reducing fill-ins in the LU decomposition. In terms of path algebra, these methods try to create fewer artificial arcs when constructing the augmented graph G'. This rationale is especially effective for APSP problems since fewer artificial arcs will incur fewer triple comparisons in all of the three major procedures of SLU.

For MPSP problems, another ordering rationale based on the requested OD pairs may also be effective. In particular, the LU decomposition procedure does more triple comparisons for arcs with head and tail that are high-indexed. Thus, shortest distances between nodes with higher indices are usually computed earlier than nodes with lower indices. In other words, we may save some computational work by using an ordering that permutes nodes of the requested OD pairs to be as close and high as possible. Since such ordering rationale may in fact incur more fill-ins in the LU decomposition phase and is too problem-dependent and intractable, we do not use this ordering in our computational tests.

The node ordering is a permutation function $perm(k)$ that permutes node k in the original ordering to node $perm(k)$ in the new ordering. We also maintain an inverse permutation function, $iperm(k)$, where node k in the new ordering corresponds to node $iperm(k)$ in the original ordering.

After choosing a good ordering $perm$, we can construct the augmented graph G' by a symbolic run of the procedure G_LU0. For convenience, all the notation referring to node index in this chapter is in the new ordering. That is, when we say $i > j$, we actually mean $perm(i) > perm(j)$.

In the symbolic execution of G_LU0, for each arc (i, j) in G', we store its index (a_{ij}), head ($head(a_{ij}) = j$), tail ($tail(a_{ij}) = i$), and length ($c(a_{ij}) = c_{ij}$ if arc (i, j) is in the original graph, or otherwise $c(a_{ij}) = M$, a very large number). We also store the topology information of G' in $di(k)$, $ui(k)$ and $uo(k)$ for each node k.

The most two expensive operations in procedure $Preprocess$ are (1) identifying a fill-in reducing ordering, which may be NP-hard and (2) the symbolic run of procedure G_LU0.

We do not take the preprocessing time into consideration when comparing with other shortest path algorithms in our computational experiments. This may sound unfair; however, if we consider our problem as solving a MPSP on a graph with fixed topology but changeable arc lengths many times for days or even for years, the one-time execution of the preprocessing procedure would indeed be negligible.

In the initialization step, we first read the result of the preprocessing. That is, we read $perm(k)$, $iperm(k)$, $di(k)$, $ui(k)$ and $uo(k)$ for each node k, and $head(a_{ij})$, $tail(a_{ij})$, and $x(a_{ij})$ for each arc (i,j) in G'. We initiate $succ_n(a_{ij}) = j$ for each arc (i,j) in A.

5.2.2 Procedure G_LU0

This procedure is the same as the procedure G_LU introduced in Section 4.2.1, except it uses the $m' \times 1$ arrays $c(a_{ij})$ and $succ(a_{ij})$ to store the length and successor of arc a_{ij} instead of the $n \times n$ arrays x_{ij} and $succ_{ij}$ of algorithm DLU.

Procedure G_LU0
begin
  for $k = 1$ to $n-2$ ***do***
    for each arc $(s,k) \in di(k)$ with index a_{sk} ***do***
      for each arc $(k,t) \in uo(k)$ with index a_{kt} ***do***
        if $s = t$ and $c(a_{sk}) + c(a_{kt}) < 0$ ***then***
          Found a negative cycle; ***STOP***
        if $s \neq t$ ***then***
          let arc (s,t) have index a_{st} ;
          if $c(a_{st}) > c(a_{sk}) + c(a_{kt})$
            $c(a_{st}) := c(a_{sk}) + c(a_{kt})$; $succ(a_{st}) := succ(a_{sk})$;
end

Note that this procedure can detect negative cycle. The total number of triple comparisons is bounded by $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|))$, or $O(n^3)$ in the worst case. It is $\frac{n(n-1)(n-2)}{3}$ on a complete graph.

5.2.3 Procedure $G\_Forward(\hat{t}_i)$

This procedure is exactly the same as the subprocedure $Get\_D\_L(\hat{t}_i)$ introduced in Section 4.2.2 which gives the shortest distance $d_n(s)$ in G'_L from each node $s > \hat{t}_i$ to node $\hat{t}_i$. It is a sparse implementation of triple comparisons $s \to k \to \hat{t}_i$ for any node s and node k satisfying $s > k > \hat{t}_i$.

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

The distance label $d_n(s)$ actually corresponds to the shortest distance in $H([1, s])$ from each node $s > \hat{t_i}$ to node $\hat{t_i}$ (see Corollary 4.2(a) in Section 4.2.5). Suppose the highest labeled node in this procedure has index $\widetilde{s}_{\hat{t_i}}$. It can be shown that there exists no path in G from any node $s > \widetilde{s}_{\hat{t_i}}$ to node $\widetilde{s}_{\hat{t_i}}$. Also, the distance label $d_n(\widetilde{s}_{\hat{t_i}})$ computed by this procedure in fact corresponds to the shortest distance (i.e., $d_n^*(\widetilde{s}_{\hat{t_i}})$) in G from node $\widetilde{s}_{\hat{t_i}}$ to node $\hat{t_i}$ (see Corollary 5.2 in Section 5.2.5).

The total number of triple comparisons in $G\_Forward(\hat{t_i})$ is bounded by $\sum_k |di(k)|$, where k are the index of labeled nodes. In the worst case, this bound is $\sum_{k=\hat{t_i}}^{n-1} |di(k)|$ and will be $O(n^2)$ for a complete graph. In general, for a MPSP problem, this procedure requires $\sum_{\hat{t_i} \in Q} \sum_{k=\hat{t_i}}^{n-1} |di(k)|$ triple operations. When we are solving an APSP problem on a complete graph, the total number of triple comparisons incurred by this procedure will be $\frac{n(n-1)(n-2)}{6}$.

The key to speeding up this procedure is to choose the lowest labeled node in $LIST$ as quickly as possible. We detail three different implementations in Section 5.3.3.

5.2.4 Procedure $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$

Since none of the nodes with index higher than $\widetilde{s}_{\hat{t_i}}$, the highest labeled node after $G\_Forward$ $(\hat{t_i})$, can reach node $\hat{t_i}$ in G (see Corollary 5.2(b) in Section 5.2.5), we only need to check nodes with index lower than or equal to $\widetilde{s}_{\hat{t_i}}$. The distance label $d_n(k)$ computed by $G\_Forward(\hat{t_i})$ represents the shortest distance in $H([1, k])$ from node $k \geq \hat{t_i}$ to node $\hat{t_i}$. Starting from the highest labeled node $\widetilde{s}_{\hat{t_i}}$ and based on the previously computed distance

label, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ efficiently computes shortest path lengths in G from each node $s \geq s_{\hat{t}_i}$ to node $\hat{t}_i$. It is a sparse implementation of triple comparisons $s \rightarrow k \rightarrow \hat{t}_i$ for any s and k such that $s_{\hat{t}_i} \leq s < k \leq \widetilde{s}_{\hat{t}_i}$.

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

In particular, for any labeled node $s \geq s_{\hat{t}_i}$, there exist paths in G to node $\hat{t}_i$. Let node $\widetilde{s}$ be the highest node in the shortest path from s to $\hat{t}_i$. Then this path can be decomposed into two parts: $s \rightarrow \widetilde{s}$ and $\widetilde{s} \rightarrow \hat{t}_i$. The shortest distance $d_n^*(\widetilde{s})$ from $\widetilde{s}$ to $\hat{t}_i$ in the second part is already computed by $G\_Forward(\hat{t}_i)$. The shortest distance from s to $\widetilde{s}$ in the first part will be computed by $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. Thus, this procedure sequentially computes $d_n^*(s)$ for $s = \widetilde{s}, \widetilde{s} - 1, \ldots, s_{\hat{t}_i}$.

The total number of triple comparisons in $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is bounded by $\sum_k |ui(k)|$, where k are the index of the set of nodes that have ever been labeled. In the worst case, this bound is $\sum_{k=s_{\hat{t}_i}+1}^{n} |ui(k)|$, and will be $O(n^2)$ for a complete graph. In general, for a MPSP problem, this procedure requires $\sum_{s_{\hat{t}_i} \in Q} \sum_{k=s_{\hat{t}_i}+1}^{n} |ui(k)|$ triple operations. When we are solving an APSP problem on a complete graph, $s_{\hat{t}_i} = 1$ for each $\hat{t}_i$ ($\hat{t}_i = 1, \ldots, n$), and the total number of triple comparisons incurred by this procedure will be $\frac{n(n-1)(n-2)}{2}$.

The key to speeding up this procedure is to choose the highest labeled node in $LIST$ as quickly as possible. We give three different implementations to speed up both $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ in Section 5.3.3.

5.2.5 Correctness and properties of algorithm SLU

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 SLU calculate shortest path lengths for various subsets of the requested OD pairs Q, and then demonstrate that every requested OD pair must be in one such subset.

Without loss of generality, we only discuss the case with one OD pair $Q = \{(s_{\hat{t_i}}, \hat{t_i})\}$ since if there is more than one distinct requested destination node $\hat{t_j}$, we simply repeat the same procedures for each $\hat{t_j}$. Also, if there is more than one distinct origin node, say, $s_1, \ldots s_q$, for the same destination node $\hat{t_i}$, we only require the one with lowest index $s_{\hat{t_i}} = \min\{s_1, \ldots s_q\}$ for $G\_Backward(s_{\hat{t_i}}, \hat{t_i})$. Thus it suffices to show that algorithm $SLU((s_{\hat{t_i}}, \hat{t_i}))$ computes the shortest path lengths in G for all the node pairs $(s, \hat{t_i})$ satisfying $s \geq s_{\hat{t_i}}$. To trace the shortest path from any node s to node $\hat{t_i}$, we have to set $s_{\hat{t_i}} = 1$ and apply algorithm $SLU((1, \hat{t_i}))$.

We begin by specifying the set of OD pairs whose shortest path lengths will be calculated by G_LU0. In particular, G_LU0 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 5.1. *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_LU0.*

Proof. Same as Theorem 4.1 in Section 4.2.5. □

Corollary 5.1. *Procedure G_LU0 will correctly compute a shortest path for any node pair (s, t) in $H([1, \min\{s, t\}] \cup \max\{s, t\})$.*

Proof. This follows immediately from Theorem 5.1. □

Now, we specify the set of OD pairs whose shortest path lengths will be calculated by Procedure $G\_Forward(\hat{t_j})$. In particular, this procedure will give shortest path lengths for OD pairs $(s, \hat{t_j})$ satisfying $s > \hat{t_j}$ and these shortest paths have all intermediate nodes with index lower than s.

Theorem 5.2. *(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. Same as Theorem 4.2 in Section 4.2.5. □

Lemma 5.1. *(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 5.2(a) and (b).

(b) It is easy to see that every path from s to t must either pass 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. □

Corollary 5.2. *(a) Procedure $G\_Forward(\hat{t_j})$ will correctly compute shortest paths in $H([1, s])$ for all node pairs $(s, \hat{t_j})$ such that $s > \hat{t_j}$.*
(b) Suppose the highest labeled node has index $\widetilde{s}_{\hat{t_i}}$ after procedure $G\_Forward(\hat{t_j})$. Then
(i) there exists no path in G from node $s > \widetilde{s}_{\hat{t_i}}$ to node $\hat{t_j}$, and
(ii) $d_n(\widetilde{s}_{\hat{t_i}})$ computed by $G\_Forward(\hat{t_j})$ represents the shortest path length $d^*_n(\widetilde{s}_{\hat{t_i}})$ from node $\widetilde{s}_{\hat{t_i}}$ to node $\hat{t_j}$.

Proof. (a) $G\_Forward(\hat{t_j})$ computes shortest path length in G'_L rooted at node $\hat{t_j}$ from all other nodes $s > \hat{t_j}$. By Theorem 5.2(a), a shortest path in G'_L from node $s > \hat{t_j}$ to node $\hat{t_j}$ corresponds to a shortest path in G from s to $\hat{t_j}$ 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 $H([1, s])$.

(b.i) Suppose there exists at least one path in G from node $s > \widetilde{s}_{\hat{t}_i}$ to node $\hat{t}_j$. Let $\widetilde{s}$ be the highest indexed intermediate node in the shortest path from s to $\hat{t}_j$. By Lemma 5.1(a), there exists a path in G'_L from $\widetilde{s}$ to $\hat{t}_j$; thus $\widetilde{s}$ will be labeled by $G\_Forward(\hat{t}_j)$. Since $\widetilde{s} \geq s > \widetilde{s}_{\hat{t}_i}$, $\widetilde{s}_{\hat{t}_i}$ will not be the highest labeled node, a contradiction. Thus no node with index higher than $\widetilde{s}_{\hat{t}_i}$ can reach node $\hat{t}_j$ in G.

(b.ii) By (b.i) we know there exists no path from node $\widetilde{s}_{\hat{t}_i}$ to any other node with higher index. Thus the shortest path in $H([1, \widetilde{s}_{\hat{t}_i}])$ corresponds to the shortest path in G. Thus $d_n(\widetilde{s}_{\hat{t}_i}) = d^*_n(\widetilde{s}_{\hat{t}_i})$. □

Finally, we demonstrate that procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ will correctly calculate all shortest path lengths for node pairs $(s, \hat{t}_i)$ satisfying $s \geq s_{\hat{t}_i}$. In particular, $G\_Backward(s_{\hat{t}_i}$, $\hat{t}_i)$ gives shortest path lengths for those requested OD pairs $(s, \hat{t}_i)$ whose shortest paths have some intermediate nodes with index higher than $\max\{s, \hat{t}_i\}$.

Theorem 5.3. *Suppose in the p^{th} iteration of procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, when the highest labeled node $\widetilde{s}^p_{\hat{t}_i}$ in LIST is removed , $d_n(\widetilde{s}^p_{\hat{t}_i}) = d^*_n(\widetilde{s}^p_{\hat{t}_i})$ is determined.*

Proof. By Corollary 5.2(c), $d_n(\widetilde{s}_{\hat{t}_i}) = d^*_n(\widetilde{s}_{\hat{t}_i})$ for the highest labeled node $\widetilde{s}_{\hat{t}_i}$ in the beginning of procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$.

When node $\widetilde{s}_{\hat{t}_i}$ is removed from $LIST$, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ update $d_n(s)$ for each node s that connects to node $s_{\hat{t}_i}$ by an arc $(s, \widetilde{s}_{\hat{t}_i})$ in G'_U. That is, $d_n(s) = \min\{d_n(s), c(a_{s\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})\}$. Let $\widetilde{s}'_{\hat{t}_i}$ be the current highest labeled node in $LIST$. $d_n(\widetilde{s}'_{\hat{t}_i})$ represents the shortest distance in $H([1, \widetilde{s}'_{\hat{t}_i}])$ from node $\widetilde{s}'_{\hat{t}_i}$ to node $\hat{t}_i$. If there exists no arc $(\widetilde{s}'_{\hat{t}_i}, \widetilde{s}_{\hat{t}_i})$ in G'_U, then obviously $d_n(\widetilde{s}'_{\hat{t}_i}) = d^*_n(\widetilde{s}'_{\hat{t}_i})$ since there exists no path from node $\widetilde{s}'_{\hat{t}_i}$ to node $\hat{t}_i$ that has intermediate node with index higher than $\widetilde{s}'_{\hat{t}_i}$.

If there exists arc $(\widetilde{s}'_{\hat{t}_i}, \widetilde{s}_{\hat{t}_i})$ in G'_U, then $c(a_{\widetilde{s}'_{\hat{t}_i}\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})$ represents the shortest distance in $H([1, \widetilde{s}'_{\hat{t}_i}] \cup \widetilde{s}_{\hat{t}_i})$. By Lemma 5.1(b) with $s = \widetilde{s}'_{\hat{t}_i}$, $t = \hat{t}_i$ and $r = \widetilde{s}'_{\hat{t}_i}$, we can conclude $d_n(\widetilde{s}'_{\hat{t}_i}) = \min\{d_n(\widetilde{s}'_{\hat{t}_i}), c(a_{\widetilde{s}'_{\hat{t}_i}\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})\} = d^*_n(\widetilde{s}'_{\hat{t}_i})$.

Using similar arguments, suppose in the p^{th} iteration, we remove the highest labeled node $\widetilde{s}^p_{\hat{t}_i}$ from $LIST$. $d_n(\widetilde{s}^p_{\hat{t}_i}) = \min\{d_n(\widetilde{s}^p_{\hat{t}_i}), c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}^{p-1}_{\hat{t}_i}}) + d^*_n(\widetilde{s}^{p-1}_{\hat{t}_i}), c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}^{p-2}_{\hat{t}_i}}) + d^*_n(\widetilde{s}^{p-2}_{\hat{t}_i}), \ldots,$ $c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}'_{\hat{t}_i}}) + d^*_n(\widetilde{s}'_{\hat{t}_i}), c(a_{\widetilde{s}^p_{\hat{t}_i}\widetilde{s}_{\hat{t}_i}}) + d^*_n(\widetilde{s}_{\hat{t}_i})\} = d^*_n(\widetilde{s}^p_{\hat{t}_i})$ by induction. □

Corollary 5.3. *(a) Procedure G_Backward($s_{\hat{t}_i}, \hat{t}_i$) terminates when node $s_{\hat{t}_i}$ is the only node in LIST, and correctly computes $d_n^*(s)$ for each node $s \geq s_{\hat{t}_i}$.*
(b) To trace the shortest path for OD pair $(s, \hat{t}_i)$, we have to initialize $s_{\hat{t}_i} := 1$ in the beginning of Algorithm SLU.

Proof. (a) By Theorem 5.3, if node $s \geq s_{\hat{t}_i}$ has been labeled by $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, its label $d_n(s) = d_n^*(s)$. For a unlabeled node $s \geq s_{\hat{t}_i}$, there exists no path from s to $\hat{t}_i$ in G', which means no path exists from s to $\hat{t}_i$ in G; thus its distance label remains M (i.e., infinity).

(b) The entries $succ_n(s)$ for each $s \geq s_{\hat{t}_i}$ are updated in all procedures whenever a better path from s to $\hat{t}_i$ is identified. To trace the shortest path for a particular OD pair $(s_i, \hat{t}_i)$, we need the entire $\hat{t}_i^{th}$ column of $[succ_{ij}^*]$ which contains information of the shortest path tree rooted at sink node $\hat{t}_i$. Thus we have to set $s_{\hat{t}_i} := 1$ so that procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ will update entries $succ_n(s)$ for each $s \geq 1$. □

Algorithm *SLU* can easily identify a negative cycle. In particular, any negative cycle will be identified in procedure *G_LU*0.

Theorem 5.4. *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_LU0 will identify it.*

Proof. Same as Theorem 4.4 in Section 4.2.5 □

To summarize, suppose the shortest path in G from $s_{\hat{t}_i}$ to $\hat{t}_i$ contains more than one intermediate node and let r be the highest intermediate node in that shortest path. If $s_{\hat{t}_i} > \hat{t}_i$, there are three cases: (1) $r < \hat{t}_i$ (2) $\hat{t}_i < r < s_{\hat{t}_i}$ and (3) $r > s_{\hat{t}_i}$. The first case will be solved by *G_LU*0, second case by $G\_Forward(\hat{t}_i)$, and third case by $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. If $s_{\hat{t}_i} < \hat{t}_i$, there are three cases: (1) $r < s_{\hat{t}_i}$ (2) $s < r < \hat{t}_i$ and (3) $r > \hat{t}_i > s$. The first case will be solved by *G_LU*0, and the second and third cases by $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. In particular, $G\_Forward(\hat{t}_i)$ only computes $d_n(k)$ for node $k > \hat{t}_i$. $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ sequentially computes $d_n^*(k)$ for node $k = n, \ldots, s_{\hat{t}_i}$.

To compute shortest path lengths for other OD pairs with the same destination $t_{\hat{i+1}} \neq \hat{t_i}$, we simply reset $d_n(k)$ and $succ_n(k)$ for each node k, and start $G\_Forward(t_{\hat{i+1}})$ and $G\_Backward(s_{t_{\hat{i+1}}}, t_{\hat{i+1}})$, without redoing G_LU0. Procedure G_LU0 only requires to be reapplied when arc lengths are changed.

When solving an APSP problem on an undirected graph, algorithm SLU becomes the same as Carré's algorithm [65], but it can save some storage and computational work compared with most shortest path algorithms. In particular, since the graph is symmetric, $ui(k) = do(k)$, and $uo(k) = di(k)$ for each node k. Therefore storing only the arcs (i, j) where $i > j$ in G' is sufficient. Special care may be required for paths using arcs in G'_U. For example, we need an additional m' dimensional successor vector, $succ'(a_{ij})$, to store the successor of arc (j, i) where $i > j$. In addition, Procedure G_LU0 can save half of its computation due to the symmetric structure.

For problems on acyclic graphs, we can reorder the nodes so that all the arcs after reordering point from higher nodes to lower nodes (or from lower nodes to higher nodes). Thus only the procedure $G\_Forward$ (or $G\_Backward$) is required, which is the same as the topological ordering method on acyclic graph.

Algorithm SLU is an efficient sparse implementation of Carré's algorithm. It depends on the topology of G'. The running time of these three procedures mainly depend on the m', the number of arcs in G'. In particular, the complexity of Algorithm SLU is $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|) + \sum_{\hat{t_i} \in Q} \sum_{k=\hat{t_i}}^{n-1} |di(k)| + \sum_{s_{\hat{t_i}} \in Q} \sum_{k=s_{\hat{t_i}}+1}^{n} |ui(k)|)$ which is $O(n^3)$ in the worst case. When solving an APSP problem on a complete graph, the three procedures G_LU0, $G\_Forward(k)$, and $G\_Backward(1, k)$ for $k = 1, \ldots, n$ will take $\frac{1}{3}$,$\frac{1}{6}$ and $\frac{1}{2}$ 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.

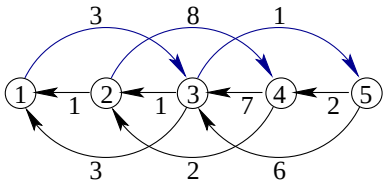
In some sense, computing the shortest distance for node pairs $(s, \hat{t_i})$ satisfying $s \geq s_{\hat{t_i}}$ can be viewed as computing the $(s, \hat{t_i})$ entry of the "inverse" of matrix $(I_n - C)$ in path algebra (see Section 3.4.1). After obtaining L and U from the LU decomposition (i.e., G_LU0), we can do the forward elimination (i.e., $G\_Forward(\hat{t_i})$) $L \otimes y_{\hat{t_i}} = b$, where the right hand side b is the $\hat{t_i}^{th}$ column of the identity matrix I_n. In particular, $y_{s\hat{t_i}}$ represents

the shortest distance in G'_L from node s to node $\hat{t}_i$, where $s > \hat{t}_i$. The backward substitution (i.e., $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$) $U \otimes x_{\hat{t}_i} = y_{\hat{t}_i}$ computes the shortest distance $x_{s\hat{t}_i}$ in G from node s to node $\hat{t}_i$, for $s = n, \ldots, s_{\hat{t}_i}$.

Carré's algorithm is algebraic and computes the ALL-ALL distance matrix. Algorithm SLU can be viewed as a truncated and decomposed sparse implementation of Carré's algorithm. In particular, SLU first "decomposes" Carré's algorithm columnwise by the requested destination nodes, and then the backward substitution is "truncated" as soon as all of the requested origins associated with the same destination are computed. Algorithm SLU terminates when all the requested OD pairs are computed; thus it skips many unnecessary triple comparisons that are required in Carré's algorithm, and is more efficient for solving MPSP problems.

Compared with algorithm $DLU2$ introduced in Section 4.3, algorithm SLU takes more advantage of the sparse topology of G'. Algorithm $DLU2$, on the other hand, requires more storage ($O(n^2)$ in general) than SLU ($O(m')$). Procedures G_LU and Get_D_L of algorithm $DLU2$ have the same number of total operations as the procedures G_LU0 and $G\_Forward$ of algorithm SLU. However, it is difficult to tell which of these two algorithms (i.e., algorithms $DLU2$ and SLU) is more efficient.

In particular, procedures Get_D_U and Min_add of algorithm $DLU2$ require operations on the dense matrix $[x_{ij}]$, but only do operations for requested OD entries. The procedure $G\_Backward$ of algorithm SLU requires operations on the sparse augmented graph G' only, but it requires the computation of shortest distances between higher-indexed node pairs before the computations on the requested OD entries. That is, to compute x^*_{st}, algorithm SLU must first compute x^*_{kt} for $k = n, \ldots, (s+1)$, but algorithm $DLU2$ can directly compute x^*_{st}. Another disadvantage of algorithm SLU is, when shortest paths have to be traced, it becomes "decomposed" like Carré's algorithm, which has to compute the whole shortest path tree like other SSSP algorithms. Algorithm $DLU2$, on the other hand, can trace any intermediate nodes very easily.



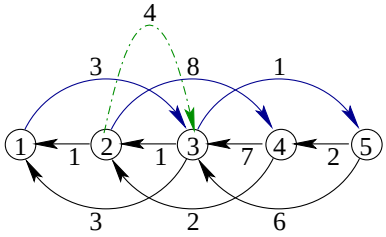
Original Graph G

0		3		
1	0		8	
3	1	0		1
	2	7	0	
		6	2	0

x_{ij}

0		3		
1	0		4	
1	2	0		5
	2	3	0	
		3	4	0

$succ_{ij}$



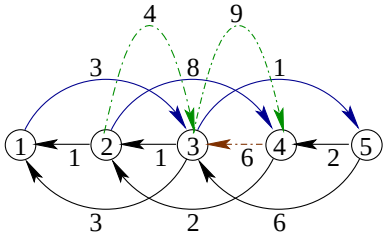
Augmented Graph G', $k = 1$

0		3		
1	0	4	8	
3	1	0		1
	2	7	0	
		6	2	0

x_{ij}

0		3		
1	0	3	4	
1	2	0		5
	2	3	0	
		3	4	0

$succ_{ij}$



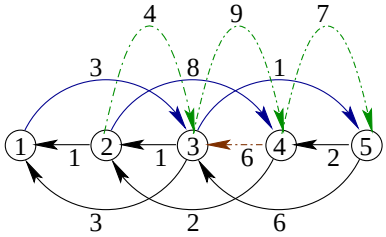
Augmented Graph G', $k = 2$

0		3		
1	0	4	8	
3	1	0	9	1
	2	6	0	
		6	2	0

x_{ij}

0		3		
1	0	3	4	
1	2	0	2	5
	2	2	0	
		3	4	0

$succ_{ij}$



Augmented Graph G', $k = 3$

0		3		
1	0	4	8	
3	1	0	9	1
	2	6	0	7
		6	2	0

x_{ij}

0		3		
1	0	3	4	
1	2	0	2	5
	2	2	0	2
		3	4	0

$succ_{ij}$

Dotted arcs have been added or modified

Modified entry in procedure G_LU

Infinity entry

Figure 8: Illustration of procedure G_LU on a small example

5.2.6 A small example

We cite a small example which computes x^*_{23} for a small network. Applying algorithm SLU, procedure G_LU0 creates the augmented graph G' which can be decomposed into G'_L and G'_U (see Figure 8). Procedure $G\_Forward(3)$ computes the shortest path length from each node $s > 3$ to node 3 in G'_L. Thus we obtain temporary distance label $d_n(2) = M$, $d_n(4) = 6$, and $d_n(5) = 6$. Note that $d^*_n(5) = d_n(5) = 6$ by Corollary 5.2(b.ii). Based on the current temporary distance labels, procedure $G\_Backward(2, 3)$ computes the shortest path length from each node $s \geq 2$ to node 3 in G'_U. In particular, $d^*_n(4) = \min\{d_n(4),\, c(a_{45}) + d^*_n(5)\} = \min\{6,\, 7+6\} = 6$, and $d^*_n(2) = \min\{d_n(2),\, c(a_{23}),\, c(a_{24}) + d^*_n(4),\, c(a_{25}) + d^*_n(5)\} = \min\{M,\, 4,\, 8+6,\, M+6\} = 4$, where $c(a_{ij})$ is the length of arc (i, j) in G'. Figure 9 illustrates the operations of procedures $G\_Forward(3)$ and $G\_Backward(2, 3)$.

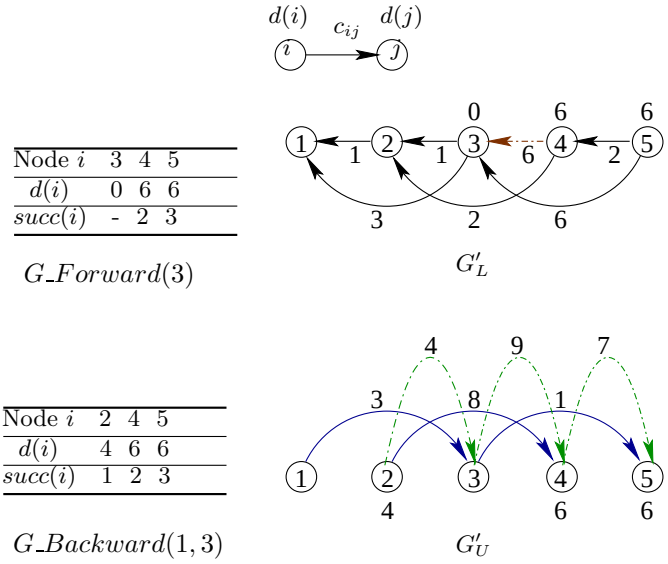


Figure 9: An example of all shortest paths to node 3

5.3 Implementation of algorithm SLU

Here we describe three efficient implementations of algorithm SLU.

5.3.1 Procedure *Preprocess*

Computing the optimal ordering that minimizes the fill-ins is NP-complete [272]. Many ordering techniques need to compute *degree production*, $i(k) \cdot o(k)$, where $i(k)$ ($o(k)$) denotes the in-degree (out-degree) of node k, for each node. The ordering techniques that minimize fill-ins have appeared in the literature of solving systems of linear equations and are applicable here since they are basically dealing with the same problem. In particular, we have implemented some of the following state-of-the-art ordering techniques:

1. **Natural ordering (NAT):**

   This ordering is the original one as read from the graph G (i.e., without any permutation).

2. **Dynamic Markowitz (DM):**

   This local minimum fill-in technique was first proposed by Markowitz [230], and has been proved very successful for general-purpose use.

   First we compute degree production for each node k. Choose the node $\hat{k}$ that has the smallest degree production from node 1 to node n, swap it with node 1, and update the degree production for the remaining $(n-1)$ nodes by removing arcs to/from $\hat{k}$. Then choose the node with smallest degree production among the remaining $(n-1)$ nodes, swap it with node 2, and update the degree production for the remaining $(n-2)$ nodes. Repeat the same procedure until finally no more node needs to be swapped.

   Note that if the graph is acyclic, the ordering obtained by Dynamic Markowitz rule will always make the nontrivial entries (i.e. finite entries) stored in the lower triangular part of the distance matrix. It is actually the so-called *topological ordering* for acyclic graph.

3. **Dynamic Markowitz with tie-breaking (DMT):**

   When using Markowitz criterion to choose node with the smallest degree production, we may often experience a tie. That is, two or more nodes may have the same degree production. Duff et al. [98] notice that a good tie-breaking strategy may substantially

affect the quality of orderings. However, how to get the best tie-breaking strategy still remains unclear. Here we implement a simple tie-breaking strategy that probes fill-ins created by the two node candidates that are tied, and chooses the one with fewer fill-ins. In particular, suppose both node r and s have the same degree production. Our strategy is to first choose r, compute its fill-ins, then reset, and choose s to compute its fill-ins as well. Finally, we choose the one with fewer fill-ins as the pivot.

In our computational results, the quality of the ordering obtained by this tie-breaking strategy is generally only a little better than the one without the tie-breaking strategy.

4. **Static Markowitz (SM):**

   This is another simple local minimum fill-in technique by Markowitz [230], which is easy to implement but usually creates more fill-ins than its dynamic variants.

   In particular, this ordering is obtained by sorting the degree production for all nodes in the ascending order, without recalculation after each assignment.

5. **METIS_NodeND (MNDn):**

   METIS [195] is a software package for partitioning unstructured graphs and computing fill-reducing orderings of sparse matrices. It is based on *multilevel nested dissection* [196] that identifies a vertex-separator, moves the separator to the end of the matrix, and then recursively applies a similar process for each one of the other two parts.

   METIS_NodeND is a stand-alone function in this package which uses a multilevel paradigm to directly find a vertex separator.

6. **METIS_EdgeND (MNDe):**

   METIS_EdgeND is another stand-alone function of METIS. It first computes an edge separator using a multilevel algorithm to find a vertex separator. The orderings produced by METIS_EdgeND generally incur more fill-ins than those produced by METIS_NodeND.

7. **Multiple Minimum Degree on C^TC (MMDm):**

To solve a sparse system of linear equations $CX = b$, Liu [223] proposes a method called *multiple minimum degree* which computes a symmetric ordering that reduces fill-ins in the Cholesky factorization of C^TC. SuperLU, a software package of Demmel et al. [91], implements this method. We use SuperLU for our tests.

8. **Multiple Minimum Degree on $C^T + C$ (MMTa):**

   This method, also proposed by Liu [223], computes a symmetric ordering that reduces fill-ins in the Cholesky factorization of C^T+C. SuperLU also implements this method.

9. **Approximate Minimum Degree Ordering (AMD):**

   The multiple minimum degree method computes a symmetric ordering on either C^TC or $C^T + C$ which may be much denser than C, and time-consuming as well. Davis et al. [89] propose this approximate minimum degree ordering method which computes a better ordering in shorter time. We also imported this function from SuperLU for our tests.

In our tests, usually the dynamic Markowitz rule with tie-breaking produces the fewest fill-ins. Dynamic Markowitz rule without tie-breaking strategy usually performs as well as the one with tie-breaking. The advanced ordering techniques of METIS and SuperLU may get a good ordering very quickly, but the quality of the ordering they compute is not as good as the dynamic Markowitz rules. Since the time to compute a good ordering is not a major concern of our research, the dynamic Markowitz rules fit our needs better since they produce better orderings in reasonable time.

5.3.2 Procedure G_LU0

G_LU0 is the procedure to construct the augmented graph G' using arc adjacency lists $di(k)$ and $uo(k)$ for $k = 1 \ldots n-1$ in the new ordering. In particular, when we scan node k, we scan each arc (i, k) in $di(k)$ and each arc (k, j) in $uo(k)$, then check whether the length of arc (i, j) need to be modified or not. To do so, we need fast access to the index of each arc (i, j) in G'.

If we use the forward/backward star [3] implementation, it may take $O(n)$ time to search $di(i)$ and $ui(i)$ for a specific arc (i, j). Another alternative is to store an extra $n \times n$ index matrix, $[a_{ij}]$, whose (i, j) entry stores the index of arc (i, j). This implementation only takes $O(1)$ time to retrieve the index given i and j, but requires extra storage.

Since modern computers have lots of storage and G_LU0 does access arcs very frequently (up to $\frac{n(n-1)(n-2)}{3}$ for a complete graph), we choose the latter implementation for our tests. In particular, we store an $n \times n$ matrix $[a_{ij}]$ to record all the arc index of the augmented graph.

This implementation of G_LU0 takes $O(\sum_{k=1}^{n-2}(|di(k)| \cdot |uo(k)|))$ time.

5.3.3 Procedures $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$

In terms of storage, $G\_Forward(\hat{t}_i)$ only requires $di(k)$, and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ requires $ui(k)$. We also use an indicator function $label(k)$ for each node k, where $label(k) = 1$ means node k is labeled and 0 otherwise. Let $N^F_{\hat{t}_i}$ denote the set of nodes that have ever entered $LIST$ in procedure $G\_Forward(\hat{t}_i)$, and $N^B_{s_{\hat{t}_i}, \hat{t}_i}$ be the set of nodes that have ever entered $LIST$ in procedure $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. Note that if $s_{\hat{t}_i} \leq \hat{t}_i$ then $N^F_{\hat{t}_i} \subseteq N^B_{s_{\hat{t}_i}, \hat{t}_i}$; otherwise $N^F_{\hat{t}_i} \supset N^B_{s_{\hat{t}_i}, \hat{t}_i}$.

When we *scan* a node k in $G\_Forward(\hat{t}_i)$ (or $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$), we are in fact doing triple comparison $s \rightarrow k \rightarrow \hat{t}_i$ for arcs (s, k) in $di(k)$ (or $ui(k)$). When we scan an arc (s, k), we mark its tail node s as labeled. The processes in these two procedures are very similar. That is, select a lowest (or highest) node k from $LIST$, scan and label the tails of arcs in $di(k)$ (or $ui(k)$), and then select the next lowest (or highest) node to scan.

Both of these two procedures contain a sorting process which selects the lowest (or highest) node from $LIST$ to scan. To efficiently select these lowest (or highest) nodes from $LIST$, we propose three different implementations. The first implementation uses $label(k)$ to check whether a node is labeled or not, which is similar to Dial's implementation [93] of Dijkstra's algorithm. The second one is to maintain a heap that stores nodes in ascending order in $G\_Forward(\hat{t}_i)$, and stores nodes in descending order in $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. The

last implementation is to maintain two heaps, a min-heap for $G\_Forward(\hat{t}_i)$ and a max-heap for $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$.

In our tests, both the first and the third implementations run faster than the second one. However, comparisons between the first implementation and the third one depend on the platforms and the compilers. In particular, the first implementation generally runs faster on a Sun Solaris workstation, while the result is reversed on a Intel PC running Linux.

5.3.3.1 Bucket implementation of $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$:

Although there is no data structure named 'bucket' in this implementation, we use this name since it resembles Dial's bucket implementation on Dijkstra's algorithm. In this implementation, we use an indicator function, $label(k)$ (i.e., a bucket), for each node k to indicate whether it is labeled or not (or in a sense, to indicate whether it is "in the bucket" or not). In the beginning, no nodes are labeled. That is, $label(k) = 0$ for each node k.

In $G\_Forward(\hat{t}_i)$, starting from the destination node $k = \hat{t}_i$, we scan arcs in $di(k)$, update the distance labels for their tails, and mark these tails as labeled. That is, $d_n(s) = \min\{d_n(s), c(a_{sk}) + d_n(k)\}$, and $label(tail(a_{sk})) = 1$ for each arc (s, k) in $di(k)$. After node k is scanned, we then search for the next lowest labeled node to scan. We repeat these processes until finally all the labeled nodes higher than $\hat{t}_i$ are scanned. At this moment, any node whose distance label $d_n(s)$ is finite must have been labeled. In particular, any labeled node has a finite distance label $d_n(s)$ to represent its shortest path length in G'_L to node $\hat{t}_i$. $G\_Forward(\hat{t}_i)$ starts from bucket $k = \hat{t}_i$, scans all its down-inward arcs (s, k) and puts each tail node s into its bucket if bucket s is still empty. It then searches for the next nonempty bucket (i.e., next labeled node) in ascending order. The operations are repeated until the last nonempty bucket $\widetilde{s}_{\hat{t}_i}$ (i.e., the highest labeled node) has been scanned.

In $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, starting from the highest labeled node $k = \widetilde{s}_{\hat{t}_i}$, we scan arcs in $ui(k)$, update the distance labels for their tails, and mark these tails as labeled if they are higher than $s_{\hat{t}_i}$. That is, $d_n(s) = \min\{d_n(s), c(a_{sk}) + d_n(k)\}$, and $label(tail(a_{sk})) = 1$ for each arc (s, k) in $ui(k)$ satisfying $s > s_{\hat{t}_i}$. Then we search for the next highest labeled node to scan. We repeat these operations until finally node $s_{\hat{t}_i}$ is scanned. At this moment, we

have finished the triple comparisons on all the arcs that are in some paths to $\hat{t}_i$ in G'_U; thus $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is terminated. $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ starts from bucket $k = \widetilde{s}_{\hat{t}_i}$ (i.e., the highest labeled node), scans all its up-inward arcs (s, k) and puts each tail node s into its bucket if bucket s is still empty. It then searches for the next nonempty bucket (i.e., next labeled node) in descending order. The operations are repeated until all the nonempty buckets with index higher than $s_{\hat{t}_i}$ (i.e., all the labeled nodes $s > s_{\hat{t}_i}$) have been scanned.

This bucket implementation has to check (1) each node $k = \hat{t}_i, \ldots, \widetilde{s}_{\hat{t}_i}$ in $G\_Forward(\hat{t}_i)$, and (2) each node $k = (s_{\hat{t}_i} + 1), \ldots, \widetilde{s}_{\hat{t}_i}$ in $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, at least once. Therefore, the complexity of this implementation is $O(\sum_{k=\hat{t}_i}^{\widetilde{s}_{\hat{t}_i}} (1) + \sum_{k \in N^F_{\hat{t}_i}} |di(k)|)$ for $G\_Forward(\hat{t}_i)$ and $O(\sum_{k=s_{\hat{t}_i}+1}^{\widetilde{s}_{\hat{t}_i}} (1) + \sum_{k \in N^B_{s_{\hat{t}_i}, \hat{t}_i}} |ui(k)|)$ for $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. This is an easy and very efficient implementation in our computational experiments. However, time spent in detecting unlabeled nodes (i.e., empty buckets) may be further saved. In particular, if we maintain all the labeled nodes in some sorted data structure such as a binary heap, then each iteration of $G\_Forward(\hat{t}_i)$ only removes the top node of a min-heap, and each iteration of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ only removes the top node of a max-heap. Such heaps avoid checking unlabeled nodes, but require overhead in heap sort.

Next, we propose the following two implementations based on heaps.

5.3.3.2 Single heap implementation of $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$:

In this implementation we use a single binary heap data structure to extract the lowest (or highest) labeled node. A node is inserted into the heap only when it is labeled. By maintaining the heap, first sorted as a min-heap in $G\_Forward(\hat{t}_i)$ and then as a max-heap in $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, we skip the scanning operations required by the bucket implementation for unlabeled nodes.

In $G\_Forward(\hat{t}_i)$, we maintain the heap as a min-heap (so that every node is lower than its children). Starting from the destination node $k = \hat{t}_i$, we scan arcs in $di(k)$, update the distance labels of their tails, mark these tails as labeled, and put them into the min-heap. Then we extract the top node from the min-heap and repeat these procedures until the

min-heap becomes empty, when we terminate $G\_Forward(\hat{t}_i)$.

To start $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, first we have to put all the labeled nodes higher than $s_{\hat{t}_i}$ into the max-heap. Then we extract the top node k (which corresponds to the highest labeled node), mark it as unlabeled, scan arcs in $ui(k)$, update the distance labels of their tails, mark these tails as labeled, and put them into the max-heap if they are higher than $s_{\hat{t}_i}$. We repeat the same procedures until finally the max-heap becomes empty. At this moment, we have finished the triple comparisons on all the arcs that are in some paths to $\hat{t}_i$ in G'_U; thus $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is terminated.

This implementation only checks labeled nodes, but its overhead in heap sorting may be time-consuming. In $G\_Forward(\hat{t}_i)$, each node in $N^F_{\hat{t}_i}$ is extracted and inserted into the min-heap exactly once. Both extracting and inserting a node on a min-heap may take $O(\log \left|N^F_{\hat{t}_i}\right|)$. Thus, an $O(\left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right|)$ time bound is obtained for all the min-heap operations in $G\_Forward(\hat{t}_i)$. This time bound is for the worst case. The average time should be much better since on average the depth of the heap is smaller than $\left|N^F_{\hat{t}_i}\right|$. Including the inevitable $\sum_{k \in N^F_{\hat{t}_i}} |di(k)|$ triple comparisons, the complexity of this single heap implementation on $G\_Forward(\hat{t}_i)$ is $O(\left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right| + \sum_{k \in N^F_{\hat{t}_i}} |di(k)|)$

In the beginning of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, we need a loop to identify labeled nodes either from node $\hat{t}_i$ to the highest labeled node $\widetilde{s}_{\hat{t}_i}$ (if $s_{\hat{t}_i} < \hat{t}_i$), or from node $s_{\hat{t}_i}$ to $\widetilde{s}_{\hat{t}_i}$ (if $s_{\hat{t}_i} > \hat{t}_i$), which will take $O(\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\})$ time. Inserting all the labeled nodes that are higher than $s_{\hat{t}_i}$ into the max-heap will take $O(\left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right|)$ time. Each node in $N^B_{s_{\hat{t}_i}, \hat{t}_i}$ is extracted exactly once, for a total of $O(\left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right|)$ time. Including the inevitable $\sum_{k \in N^B_{s_{\hat{t}_i}, \hat{t}_i}} |ui(k)|$ triple comparisons, the complexity of this single heap implementation on $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ is $O(\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\} + \left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right| + \sum_{k \in N^B_{s_{\hat{t}_i}, \hat{t}_i}} |ui(k)|)$. Again this is for the worst case. On average, extracting a node from or inserting a node into the max-heap may take time shorter than $O(\log \left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right|)$ since the depth of the max-heap is usually smaller than $\left|N^B_{s_{\hat{t}_i}, \hat{t}_i}\right|$.

This implementation only maintains a single heap which becomes empty in the beginning

of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. To insert those nodes that are labeled by $G\_Forward(\hat{t}_i)$ into the max-heap in the beginning of $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$, an overhead that checks $\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\}$ nodes has to be made. If only a few of these $\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\}$ nodes are labeled, this overhead will become inefficient. Next, we propose another way to avoid this overhead, at the cost of maintaining an additional heap data structure.

5.3.3.3 Two-heap implementation of $G\_Forward(\hat{t}_i)$ and $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$:

Instead of using a single heap, this implementation maintains two heaps, one min-heap for $G\_Forward(\hat{t}_i)$, and one max-heap for $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$.

In particular, after we extract the lowest labeled node k from the min-heap in $G\_Forward$ $(\hat{t}_i)$, we insert that node into the max-heap right away, if $k \geq s_{\hat{t}_i}$. Therefore, after $G\_Forward(\hat{t}_i)$, the max-heap is ready to start $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$. $G\_Forward(\hat{t}_i)$ spends $O(\ \min\{\ \left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right|, \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right|\ \})$ time inserting nodes into the max-heap, and $O(\left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right|)$ time inserting and extracting nodes on the min-heap. Including the $O(\sum_{k \in N^F_{\hat{t}_i}} |di(k)|)$ triple comparisons, the overall complexity of $G\_Forward(\hat{t}_i)$ is $O(\ \left|N^F_{\hat{t}_i}\right| \log \left|N^F_{\hat{t}_i}\right| + \sum_{k \in N^F_{\hat{t}_i}} |di(k)|\ )$. Similarly, $G\_Backward(s_{\hat{t}_i}, \hat{t}_i)$ will take $O(\ \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| \log \left|N^B_{s_{\hat{t}_i},\hat{t}_i}\right| + \sum_{k \in N^B_{s_{\hat{t}_i},\hat{t}_i}} |ui(k)|\ )$ time.

Therefore this two-heap implementation has a better time bound but needs more storage than the single heap implementation.

5.3.4 Summary on different SLU implementations

We give three implementations of algorithm SLU: a bucket implementation ($SLU1$), a single heap implementation ($SLU2$), and a two-heap implementation ($SLU3$). All of these three implementation have the same procedure G_LU0. They differ by techniques of implementing procedures $G\_Forward$ and $G\_Backward$.

Table 7 compares the running time for these three implementations.

It is difficult to tell which implementation is theoretically better than the others. However, the bucket implementation seems to be faster in practice, according to our experiments in Section 5.6.2.

Table 7: Running time of different SLU implementations

	G_LU0	**G_Forward**$(\hat{t}_i)$	**G_Backward**$(\mathbf{s}_{\hat{t}_i}, \hat{t}_i)$
bucket	$\sum_{k=1}^{n-2}(\lvert di(k)\rvert \cdot \lvert uo(k)\rvert)$	$\sum_{k=\hat{t}_i}^{\widetilde{s}_{\hat{t}_i}}(1) + \sum_{k\in N^F_{\hat{t}_i}} \lvert di(k)\rvert$	$\sum_{k=s_{\hat{t}_i}+1}^{\widetilde{s}_{\hat{t}_i}}(1) + \sum_{k\in N^B_{s_{\hat{t}_i},\hat{t}_i}} \lvert ui(k)\rvert$
single heap	$\sum_{k=1}^{n-2}(\lvert di(k)\rvert \cdot \lvert uo(k)\rvert)$	$\left\lvert N^F_{\hat{t}_i}\right\rvert \log \left\lvert N^F_{\hat{t}_i}\right\rvert + \sum_{k\in N^F_{\hat{t}_i}} \lvert di(k)\rvert$	$\widetilde{s}_{\hat{t}_i} - \max\{s_{\hat{t}_i}, \hat{t}_i\}$ $+\left\lvert N^B_{s_{\hat{t}_i},\hat{t}_i}\right\rvert \log \left\lvert N^B_{s_{\hat{t}_i},\hat{t}_i}\right\rvert + \sum_{k\in N^B_{s_{\hat{t}_i},\hat{t}_i}} \lvert ui(k)\rvert$
two-heap	$\sum_{k=1}^{n-2}(\lvert di(k)\rvert \cdot \lvert uo(k)\rvert)$	$\left\lvert N^F_{\hat{t}_i}\right\rvert \log \left\lvert N^F_{\hat{t}_i}\right\rvert + \sum_{k\in N^F_{\hat{t}_i}} \lvert di(k)\rvert$	$\left\lvert N^B_{s_{\hat{t}_i},\hat{t}_i}\right\rvert \log \left\lvert N^B_{s_{\hat{t}_i},\hat{t}_i}\right\rvert + \sum_{k\in N^B_{s_{\hat{t}_i},\hat{t}_i}} \lvert ui(k)\rvert$

5.4 Implementation of algorithm $DLU2$

Algorithm $DLU2$ is one of our new MPSP algorithms proposed in Section 4.3. Here we describe two efficient implementations of algorithm $DLU2$. Algorithm $DLU2$ (see Section 4.3) contains three major procedures: $Preprocess$, G_LU (see Section 4.2.1) and $Get\_D(s_i, t_i)$ (see Section 4.3.1). The preprocessing ($Preprocess$) and LU decomposition (G_LU) procedures are similar to those of algorithm SLU.

All the operations of SLU are based on the arc adjacency lists of G' (i.e., $di(k)$, $uo(k)$, and $ui(k)$ for each node k). Each iteration of SLU updates only two $n \times 1$ arrays $d_n(k)$ and $succ_n(k)$ to report the shortest path. On the other hand, although algorithm $DLU2$ performs most of its operations based on the arc adjacency lists of G' (i.e., $di(k)$ and $uo(k)$ for each node k), it also updates two $n \times n$ arrays $[x_{ij}]$ and $[succ_{ij}]$.

In particular, the procedure G_LU and subprocedures $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$ update $[x_{ij}]$ and $[succ_{ij}]$ by operations on arcs $di(k)$ and $uo(k)$ for each node k in G'. We implement $Min\_add(s_i, t_i)$ algebraically since the updated $[x_{ij}]$ and $[succ_{ij}]$ become denser and make "graphical" implementation of $Min\_add(s_i, t_i)$ less efficient.

To speed up the acyclic operations of subprocedures $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, we propose two implementations: bucket implementation and heap implementation.

5.4.1 Bucket implementation of $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$

This is similar to the bucket implementation of $G\_Forward(t_i)$ in Section 5.3.3.1.

In $Get\_D\_L(t_i)$, starting from node $k = t_i$, we scan arcs in $di(k)$, update the distance

Algorithm 8 DLU2($\mathbf{Q} := \{(\mathbf{s_i}, \mathbf{t_i}) : \mathbf{i} = \mathbf{1}, \ldots, \mathbf{q}\}$)

begin
  Preprocess;
  Initialize:
    $opt_{ij} := 0 \; \forall \; i = 1, \ldots, n, \; j = 1, \ldots, n$
    $subrow_i := 0$; $subcol_i := 0 \; \forall \; i = 1, \ldots, n$
  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 $x_{s_i t_i} \neq \infty$ ***then***
        $Get\_P(s_i, t_i)$;
      else there exists no path from s_i to t_i
end

Procedure Preprocess
begin
  Decide a node ordering $perm(k)$ for each node k;
  Symbolic execution of procedure *G_LU* to determine the arc adjacency list $di(k)$, and $uo(k)$, for each node k of the augmented graph G' ;
  Initialize: $\forall (s,t) \in G'$, $x_{st} := c_{st}$; $succ_{st} := t$
  Initialize: $\forall (s,t) \notin G'$, $x_{st} := M$; $succ_{st} := 0$
    $opt_{ij} := 0 \; \forall \; i = 1, \ldots, n, \; j = 1, \ldots, n$
    $subrow_i := 0$; $subcol_i := 0 \; \forall \; i = 1, \ldots, n$
end

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 $di(t)$
            if $s < t$ ***then***
              add a new arc (s, t) to $uo(s)$
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Procedure Get_D(s_i, t_i)
begin
  if $opt_{s_i t_i} = 0$ ***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 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 $(s, k) \in di(k)$ ***do***
      if $s \notin LIST$, put s into $LIST$
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
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***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$
end

Subprocedure Min_add(s_i, t_i)
begin
  $r_i := \max\{s_i, t_i\}$
  for $k = n$ down to $r_i + 1$ ***do***
    if $x_{s_i t_i} > x_{s_i k} + x_{k t_i}$ ***then***
      $x_{s_i t_i} := x_{s_i k} + x_{k t_i}$; $succ_{s_i t_i} := succ_{s_i k_i}$
  $opt_{s_i t_i} := 1$
end

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 $k := succ_{k t_i}$
end

labels for their tails, and mark these tails as labeled. That is, $x_{st_i} = \min\{x_{st_i}, x_{sk} + x_{kt_i}\}$, and $label(s) = 1$ for each arc (s, k) in $di(k)$. After node k is scanned, we then search for the next lowest labeled node to scan and repeat these procedures until finally all the labeled nodes higher than t_i are scanned. Then we reset $label(s) = 0$ for each node s.

In $Get\_D\_U(s_i)$, starting from node $k = s_i$, we scan arcs in $uo(k)$, update the distance labels for their heads, and mark these heads as labeled. That is, $x_{s_i t} = \min\{x_{s_i t}, x_{s_i k} + x_{kt}\}$, and $label(t) = 1$ for each arc (k, t) in $uo(k)$. After node k is scanned, we then search for the next lowest labeled node to scan and repeat these procedures until finally all the labeled nodes higher than s_i are scanned. Then we reset $label(t) = 0$ for each node t.

Let $\widetilde{s}_{t_i}$ and $\widetilde{t}_{s_i}$ denote the highest labeled node obtained by $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, respectively. Let $N^L_{t_i}$ and $N^U_{s_i}$ be the set of labeled nodes by $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, respectively. The complexity of the bucket implementation is $O(\sum_{k=t_i}^{\widetilde{s}_{t_i}} (1) + \sum_{k \in N^L_{t_i}} |di(k)|)$ for $Get\_D\_L(t_i)$ and $O(\sum_{k=s_i}^{\widetilde{t}_{s_i}} (1) + \sum_{k \in N^U_{s_i}} |uo(k)|)$ for $Get\_D\_U(s_i)$. Note that we use the indicator arrays $subcol_{t_i}$ and $subrow_{s_i}$ for each distinct s_i and t_i to avoid redundant operations.

5.4.2 Heap implementation of $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$

This is similar to the bucket implementation of $G\_Forward(t_i)$ in Section 5.3.3.2.

In $Get\_D\_L(t_i)$, we maintain a min-heap where any of its tree node is lower than its children. Starting from the destination node $k = t_i$, we scan arcs in $di(k)$, update the distance labels for their tails, mark these tails as labeled, and put them into the min-heap. Then we extract the top node from the min-heap and repeat these procedures until the min-heap becomes empty.

In $Get\_D\_U(s_i)$, we use the same min-heap that has become empty after $Get\_D\_L(t_i)$.

Starting from the origin node $k = s_i$, we scan arcs in $uo(k)$, update the distance labels for their heads, mark these heads as labeled, and put them into the min-heap. Then we extract the top node from the min-heap and repeat these procedures until the min-heap becomes empty.

Let $N_{t_i}^L$ and $N_{s_i}^U$ be the set of nodes labeled by $Get\_D\_L(t_i)$ and $Get\_D\_U(s_i)$, respectively. The complexity of the bucket implementation is $O(\left|N_{t_i}^L\right| \log \left|N_{t_i}^L\right| + \sum_{k \in N_{t_i}^L} |di(k)|)$ for $Get\_D\_L(t_i)$ where $\sum_{k \in N_{t_i}^L} |di(k)|$ is the total number of scans and $\left|N_{t_i}^L\right| \log \left|N_{t_i}^L\right|$ comes from inserting a node into and extracting a node from a binary heap. Similarly, $Get\_D\_U(s_i)$ has an $O(\left|N_{s_i}^U\right| \log \left|N_{s_i}^U\right| + \sum_{k \in N_{s_i}^U} |uo(k)|)$ time bound.

5.5 Settings of computational experiments

In this Section, we present the test conditions for our computational experiments on solving MPSP problems.

5.5.1 Artificial networks and real flight network

We use four network generators which produce artificial networks: SPGRID, SPRAND and SPACYC are designed by Cherkassky et al. [74]; NETGEN is by Klingman et al. [207, 185]. We also test our algorithms on a real Asia-Pacific flight network.

5.5.1.1 Asia-Pacific flight network (AP-NET):

As shown in Chapter 1, the AP-NET contains 112 nodes (48 center nodes in Table 2, and 64 rim nodes in Table 3) and 1038 arcs, where each node represents a chosen city and each arc represents a chosen flight. Among the 1038 arcs, 480 arcs connect center cities to center cities; 277 arcs connect rim cities to center cities; and 281 arcs connect center cities to rim cities. We use the great circle distance between the departure and arrival cities of each flight leg as the arc length.

5.5.1.2 SPGRID:

SPGRID generates a grid network defined by $XY + 1$ nodes. In particular, consider a plane with XY integer coordinates $[x, y]$, $1 \leq x \leq X$, $1 \leq y \leq Y$. An arc with tail $[x, y]$ is called

"forward" if its head is $[x+1, y]$, "upward" if its head is $[x, (y+1) \bmod Y]$, and "downward" if its head is $[x, (y-1) \bmod Y]$ for $1 \leq x \leq X$, $1 \leq y \leq Y$. A "super source" node is connected to nodes $[1, y]$ for $1 \leq y \leq Y$. Let each "layer" be the subgraph induced by the nodes of the same x. For each layer, SPGRID can specify each layer to be either doubly connected (i.e. each layer is a doubly connected cycle) or singly connected. Among all of the adjustable parameters, we choose to adjust X, Y, layer connectivity, and the range of arc lengths for arcs inside each layer ($[c_l, c_u]$) and arcs between different layers ($[\hat{c}_l, \hat{c}_u]$). The arc lengths will be uniformly chosen in the range.

Following the settings used in [74], we generate the following four SPGRID families:

- SPGRID-SQ: (SQ stands for square grid)

  $X = Y \in \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$; each layer is double cycle;

  $[c_l, c_u] = [10^3, 10^4]$; $[\hat{c}_l, \hat{c}_u] = [10^3, 10^4]$

- SPGRID-WL: (WL stands for wide or long grid)

  Wide: $X = 16, Y \in \{64, 128, 256, 512\}$; Long: $Y = 16, X \in \{64, 128, 256, 512\}$;

  each layer is double cycle; $[c_l, c_u] = [10^3, 10^4]$; $[\hat{c}_l, \hat{c}_u] = [10^3, 10^4]$

- SPGRID-PH: (PH stands for positive arc lengths and hard problems)

  $Y = 32, X \in \{16, 32, 64, 128, 256\}$; each layer is single cycle;

  $[c_l, c_u] = [1, 1]$; $[\hat{c}_l, \hat{c}_u] = [10^3, 10^4]$; randomly assign 64 additional arcs within each layer

- SPGRID-NH: (NH stands for negative arc lengths and hard problems)

  $Y = 32, X \in \{16, 32, 64, 128\}$; each layer is single cycle;

  $[c_l, c_u] = [1, 1]$; $[\hat{c}_l, \hat{c}_u] = [-10^4, -10^3]$; randomly assign 64 additional arcs within each layer

5.5.1.3 SPRAND:

SPRAND first constructs a hamiltonian cycle, and then adds arcs with distinct random end points. Except for the SPRAND-LENS and SPRAND-LEND family, we set the length

of the arcs in the hamiltonian cycle to be 1 and others to be uniformly chosen from the interval $[0, 10^4]$.

By adjusting the parameters, we generate the following six SPRAND families:

- SPRAND-S: (S stands for sparse graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$
- SPRAND-D: (D stands for dense graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$
- SPRAND-LENS: (LEN stands for different arc lengths; S stands for sparse graphs)
  $|N| \in \{256, 1024\}$; average degree = 4; range of arc lengths $[L, U] \in \{[1, 1]$, $[0, 10]$, $[0, 10^2]$, $[0, 10^4]$, $[0, 10^6]\}$
- SPRAND-LEND: (LEN stands for different arc lengths; D stands for dense graphs)
  $|N| \in \{256, 1024\}$; $|A| = \frac{|N|(|N|-1)}{4}$; range of arc lengths $[L, U] \in \{[1, 1]$, $[0, 10]$, $[0, 10^2]$, $[0, 10^4]$, $[0, 10^6]\}$
- SPRAND-PS: (P stands for changeable node potential; S stands for sparse graphs)
  $|N| \in \{256, 1024\}$; average degree = 4; lower bound of node potential = 0; upper bound of node potential $\in \{0, 10^4, 10^5, 10^6\}$
- SPRAND-PD: (P stands for changeable node potential; D stands for dense graphs)
  $|N| \in \{256, 1024\}$; $|A| = \frac{|N|(|N|-1)}{4}$; lower bound of node potential = 0; upper bound of node potential $\in \{0, 10^4, 10^5, 10^6\}$

5.5.1.4 NETGEN:

NETGEN is a network generator developed by Klingman et al. [207]. We use its C version to create our testing categories. Among many adjustable parameters, we choose to adjust $|N|$, $|A|$ and $[L, U]$ where L and U represent the lower and upper bounds on arc lengths.

By adjusting the parameters, we generate the following four NETGEN families:

- NETGEN-S: (S stands for sparse graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$; $[L, U] = [0, 10^3]$

- NETGEN-D: (D stands for dense graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; $[L, U] = [0, 10^3]$

- NETGEN-LENS: (LEN stands for different arc lengths; S stands for sparse graphs)
  $|N| \in \{256, 1024\}$; average degree = 4; $L = 0$; $U \in \{0, 10^2, 10^4, 10^6\}$

- NETGEN-LEND: (LEN stands for different arc lengths; D stands for dense graphs)
  $|N| \in \{256, 1024\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; $L = 0$; $U \in \{0, 10^2, 10^4, 10^6\}$

5.5.1.5 SPACYC:

SPACYC generates acyclic networks. It first constructs a central path starting from node 1 that visits every other node exactly once, and then randomly connects nodes. All arcs are oriented from nodes of smaller index to nodes of larger index. Among many adjustable parameters, we choose to adjust $|N|$, $|A|$ and $[L, U]$ where L and U represent the lower and upper bounds on arc lengths.

By adjusting the parameters, we generate the following five SPACYC families:

- SPACYC-PS: (P stands for positive arc length; S stands for sparse graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$; arcs in the central path have length = 1, all other arcs have lengths uniformly distributed in $[L, U] = [0, 10^4]$

- SPACYC-NS: (N stands for negative arc length; S stands for sparse graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; average degree $\in \{4, 16\}$; arcs in the central path have length $= -1$, all other arcs have lengths uniformly distributed in $[L, U] = [-10^4, 0]$

- SPACYC-PD: (P stands for positive arc length; D stands for dense graphs)
  $|N| \in \{128, 256, 512, 1024, 2048\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; arcs in the central path have length = 1, all other arcs have lengths uniformly distributed in $[L, U] = [0, 10^4]$

- SPACYC-NS: (N stands for negative arc length; D stands for dense graphs)
  $|N| \in \{128, 256, 512, 1024\}$; $|A| = \frac{|N|(|N|-1)}{k}$ where $k \in \{4, 2\}$; arcs in the central

path have length $= -1$, all other arcs have lengths uniformly distributed in $[L, U] = [-10^4, 0]$

- SPACYC-P2N: (P2N stands for changing the arc lengths from positive to negative) $|N| \in \{128, 1024\}$; average degree $= 16$; arcs in the central path have length $= 1$, all other arcs have lengths uniformly distributed in $[L, U] \in 10^3 \times \{[0, 10], [-1, 9], [-2, 8], [-3, 7], [-4, 6], [-5, 5], [-6, 4], [-10, 0]\}$

5.5.2 Shortest path codes

Using different node selection techniques, we have several implementations of our MPSP algorithms SLU and $DLU2$. We also implement a "combinatorial" (or graphical) Floyd-Warshall (FW) algorithm which is much faster than its naive algebraic implementation. All versions of SLU, DLU, and FW share the same preprocessing procedure which determines a sparse node ordering.

For other SSSP algorithms, we modify the label correcting and label setting codes written by Cherkassky et al. [74] for solving the MPSP problems.

5.5.2.1 SLU codes:

All versions $SLU1$, $SLU2$, and $SLU3$ have the same LU factorization subroutines. They differ by techniques of implementing the forward elimination and backward substitution procedures. In particular, $SLU1$ uses a technique similar to Dial's bucket implementation as described in Section 5.3.3.1; $SLU2$ uses single heap as introduced in Section 5.3.3.2; $SLU3$ uses two heaps as introduced in Section 5.3.3.3.

5.5.2.2 DLU2 codes:

Both $DLU21$ and $DLU22$ have the same LU factorization and min-addition subroutines. They differ by techniques of implementing the acyclic operations in *Acyclic_L* and *Acyclic_U* procedures. In particular, $DLU21$ uses a technique similar to Dial's bucket implementation as described in Section 5.4.1; $DLU22$ uses single heap as introduced in Section 5.4.2.

5.5.2.3 *Floyd-Warshall code:*

The graphical Floyd-Warshall algorithm FWG takes advantage of the sparse node ordering to avoid more trivial triple comparisons than its naive algebraic implementation FWA. In particular, in the k^{th} iteration of FWG, the nontrivial triple comparison $i \rightarrow k \rightarrow j$ computes $\min\{x_{ij}, x_{ik}+x_{kj}\}$ for $x_{ik} \neq \infty$, and $x_{kj} \neq \infty$. These nontrivial triple comparisons can be efficiently implemented if we maintain a $n \times 1$ outgoing arc list (k, l) and a $n \times 1$ incoming arc list (l, k) for $l = 1, \ldots, n$ for each node k. In the procedure, whenever $x_{ij} = \infty$ and $x_{ik}+x_{kj} < \infty$, we add a new arc (i, j) to the outgoing arc list of node i and incoming arc list of node j. Thus this implementation avoids trivial triple comparisons where $x_{ik} = \infty$ or $x_{kj} = \infty$ and is much faster.

5.5.2.4 *Label correcting SSSP codes: $GOR1$, BFP, $THRESH$, $PAPE$, $TWOQ$*

These five label correcting SSSP codes are chosen from Cherkassky et al. [74] due to their better performance than other implementations of label correcting algorithms. They differ with each other in the order of node selection for scanning.

In particular, $GOR1$, proposed by Goldberg and Radzik [140], does a topological ordering on the candidate nodes so that the node with more descendants will be scanned earlier. BFP is a modified Bellman-Ford algorithm [40] which scans a node only if its parent is not in the queue. $THRESH$ is the threshold algorithm by Glover et al. [134]. $PAPE$ is a dequeue implementation by Pape [264] and Levit [220] which maintains two candidate lists: one is a stack and the other is a queue. It always selects a node to scan from the stack, if it is not empty; otherwise, it scans a node from the queue. When it scans a node, that is, checks the heads of the outgoing arcs from that node, if the head has not been scanned yet, it will be put into the queue; otherwise, it will be put into the stack. $PAPE$ has been shown to have an exponential time bound [203, 290], but is practically efficient in most real-world networks. $TWOQ$ by Pallottino [260] is a similar algorithm which maintains two queues instead of one stack and one queue as $PAPE$.

When solving SSSP problems, $GOR1$ and BFP have complexity $O(nm)$, $THRESH$ has complexity $O(nm)$ for problems with nonnegative arc lengths, and $TWOQ$ has complexity

$O(n^2m)$.

5.5.2.5 Label setting SSSP codes: DIKH, DIKBD, DIKR, DIKBA

These four label setting SSSP codes are chosen from Cherkassky et al. [74] due to their better performance than other implementations of label setting algorithms. All of them are variants of Dijkstra's algorithm. They differ with each other in the way of selecting the node with smallest distance label.

In particular, $DIKH$ is the most common binary heap implementation. $DIKBD$, $DIKR$, and $DIKBA$ can be viewed as variants of Dial's bucket implementation [93]. $DIKBD$ is a double bucket implementation, $DIKBA$ is an approximate bucket implementation, and $DIKR$ is the radix-heap implementation. See Cherkassky et al. [74] for more detailed introduction on these implementations and their complexities.

5.5.2.6 Acyclic code: (ACC)

ACC is also written by Cherkassky et al. [74] to test the performance of different algorithms on acyclic graphs. It is based on topological ordering which has complexity $O(m)$ for solving SSSP problems.

5.5.3 Requested OD pairs

The indices of the requested OD pairs may affect the efficiency of our algorithms $DLU2$ and SLU, but will not affect the SSSP algorithms. In particular, algorithm $DLU2$ and SLU compute the shortest path lengths for node pairs with larger index earlier or faster than node pairs with smaller index since both $DLU2$ and SLU use the LU factorization, which does more triple comparisons for higher node pairs. The SSSP algorithms, on the other hand, find the shortest path tree at each iteration for different roots, and thus does not depend on the indices of the requested OD pairs.

To do a fair computational comparison, we first randomly choose several destination nodes (columns in the OD matrix). For each destination node, we randomly choose its associated origin node (row). We produce four sets: OD_{25}, OD_{50}, OD_{75}, and OD_{100} of OD pairs for each test family, in which OD_k means that the requested OD pairs cover $k\% * |N|$

Table 8: number of fill-ins created by different pivot rules

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

destination nodes (columns). Using these four random OD sets for each test family, we can observe how our algorithms $DLU2$ and SLU perform compared with the repeated SSSP algorithms.

Intuitively, algorithm $DLU2$ and SLU should be advantageous for the OD_{100} case. The 100% random requested OD pairs will cover all the columns and make the MPSP problem an APSP problem for repeated SSSP algorithms. However, algorithm $DLU2$ and SLU save some computational work since they terminate when all the $|N|$ requested OD pairs are computed, instead of all $\frac{|N|(|N|-1)}{2}$ OD pairs.

5.6 Computational results

This section summarizes our comprehensive computational experiments. First we compare the different preprocessing rules, and then we compare different implementations of SLU, $DLU2$ and how they perform compared with the Floyd-Warshall algorithm. Finally we compare SLU and $DLU2$ with many SSSP algorithms on many different problem families using different percentages of covered columns in the set of requested OD pairs.

5.6.1 Best sparsity pivoting rule

Using the Asia-Pacific flight network (AP-NET) as an example, Table 8 gives the number of fill-ins induced by different pivot rules (see Section 5.3.1). None of these 9 pivoting rules is always superior to the others. However in our experience, usually the dynamic Markowitz rule (with or without tie breaking) produces fewer fill-ins than others. Thus in all of our experiments, we compare the number of fill-ins induced by NAT, DM, and DMT, then choose the smallest of these three rules as our pivoting rule.

5.6.2 Best SLU implementations

$SLU1$, $SLU2$, and $SLU3$ all use the same node ordering produced by the preprocessing procedure. In our tests (see Tables 35,...,91 in the appendix), $SLU1$ is always faster than $SLU2$ and $SLU3$. $SLU2$ performs a little better than $SLU3$, but in general they perform very similarly. We also have an interesting finding which indicates that the heap-oriented codes will perform better on Intel machines than Sun machines. In particular, the relative performance of the heap-oriented codes such as $SLU2$, and $SLU3$ will improve on Intel machines using Linux (Mandrake 8.2) or Windows (cygwin on Win2000). Nevertheless, $SLU1$ is still more efficient overall.

5.6.3 Best $DLU2$ implementations

Both $DLU21$ and $DLU22$ use the same node ordering produced by the preprocessing procedure. In our tests (see Tables 35,...,91 in the appendix), $DLU21$ is usually faster than $DLU22$. As discussed previously, the relative performance of the heap-oriented codes such as $DLU22$ will improve on Intel machines. Nevertheless, $DLU21$ is still more efficient asymptotically.

5.6.4 SLU and $DLU2$ vs. Floyd-Warshall algorithm

Algorithms SLU and $DLU2$ always perform much faster than the Floyd-Warshall algorithm. Here we use the AP-NET as an example. Suppose there are 112 requested OD pairs whose destination nodes span the whole node set (i.e. 112 nodes). Table 9 shows the relative running times of the naive algebraic implementation of Floyd-Warshall algorithm (FWA), the graphical implementation (FWG), and the SLU and $DLU2$ implementations on different platforms.[1] In this specific example, $DLU21$ is the fastest code. Although FWG significantly improves upon FWA, it is still worse than $SLU1$ and $DLU21$. We also observe that when the network becomes larger, the Floyd-Warshall algorithm becomes more inefficient. The reason may be due to more memory accessing operations since it is an algebraic algorithm and requires $O(n^2)$ storage.

[1]Sun is a Sun workstation; Mdk is Mandrake Linux on an Intel machine; Win is Win2000 on an Intel machine.

Table 9: Floyd-Warshall algorithms vs. SLU and $DLU2$

	FWA	FWG	SLU1	SLU2	SLU3	DLU21	DLU22
Sun	18.77	8.21	2.10	5.49	5.38	1.00	1.37
Mdk	12.25	5.62	3.87	6.25	5.06	1.00	1.29
Win	13.37	6.17	2.63	5.77	5.63	1.00	1.29

5.6.5 SLU and $DLU2$ vs. SSSP algorithms

In our computational experiments, we produce four sets: OD_{25}, OD_{50}, OD_{75}, and OD_{100} of OD pairs for each problem family. For the same problem family, the SSSP algorithms have consistent performance for different numbers of OD pairs. This is because the SSSP algorithms solve ALL-1 shortest path trees for each destination; thus increasing the number of distinct destinations simply increases the number of applications of these algorithms. On the other hand, our MPSP algorithms SLU and $DLU2$ will have better relative performance when number of distinct destinations increases. In particular, for the same problem set, SLU and $DLU2$ will perform relatively better on cases of $100\%\,|N|$ distinct destinations than on cases of $25\%\,|N|$ distinct destinations. The computationally burdensome LU factorization procedure of SLU and $DLU2$ only needs to be done once, after which the overall effort for solving APSP problems is less. Thus, for problems requesting OD pairs with more distinct destinations, the overhead in the LU factorization will not be wasted.

5.6.5.1 Flight networks

Table 10 lists the computational results of 15 algorithms on the AP-NET.

We observe that all versions of SLU and $DLU2$ perform better for cases with more distinct destinations. In this specific example, $DLU21$ is one of the fastest codes, especially for cases where more than 75% distinct requested destinations. Most label correcting methods like $PAPE$, $TWOQ$, $THRESH$, and BFP also perform well. $SLU1$ is not as fast as $DLU2$, but is still faster than all the Dijkstra codes. FWG, our modified Floyd-Warshall algorithm, is the slowest one even when all the SSSP codes are solving an APSP problem in the case with 100% distinct requested destinations.

It is also interesting that those codes which require more memory access (such as FWG,

Table 10: Relative performance of different algorithms on AP-NET

		FWG	SLU 1	SLU 2	SLU 3	DLU 21	DLU 22	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
25%	Sun	57.09	4.30	10.22	10.52	2.22	4.35	3.78	1.09	2.04	1.09	1.00	6.00	4.35	8.78	10.57
	Mdk	21.16	4.26	5.95	5.00	1.42	1.79	2.84	1.05	1.63	1.00	1.05	3.53	3.32	5.32	4.05
	Win	19.12	2.30	4.75	4.46	1.20	1.35	2.90	1.00	1.55	1.00	1.05	3.30	3.40	5.71	6.80
50%	Sun	27.52	3.71	9.46	9.12	2.33	3.06	3.40	1.08	1.96	1.02	1.00	5.67	4.27	8.35	10.31
	Mdk	10.92	3.81	5.92	4.81	1.03	1.70	2.70	1.11	1.65	1.00	1.05	3.49	3.27	5.35	4.51
	Win	9.54	2.10	5.00	4.38	1.12	1.30	2.95	1.02	1.62	1.05	1.00	3.56	3.50	5.83	6.58
75%	Sun	18.21	3.44	8.84	8.86	1.77	2.37	3.33	1.07	1.97	1.05	1.00	5.67	4.21	8.36	9.52
	Mdk	9.02	4.36	7.13	5.71	1.00	1.71	3.47	1.36	2.09	1.24	1.33	4.40	4.00	6.69	5.69
	Win	7.50	2.47	5.24	5.03	1.00	1.51	3.44	1.22	1.88	1.22	1.16	4.08	4.02	6.87	7.07
100%	Sun	13.63	3.49	9.12	8.93	1.66	2.27	3.55	1.08	1.99	1.05	1.00	5.66	4.43	8.52	10.29
	Mdk	5.62	3.87	6.25	5.06	1.00	1.29	3.01	1.17	1.78	1.09	1.14	3.78	3.48	5.75	4.80
	Win	6.17	2.63	5.77	5.63	1.00	1.29	3.72	1.35	2.23	1.39	1.27	4.21	4.55	7.26	5.58

Table 11: 75% SPGRID-SQ

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
10x10/3	5.00	6.20	5.50	1.30	3.30	1.10	1.00	7.30	6.10	10.90	26.10
20x20/3	5.54	3.73	5.04	1.18	2.58	1.08	1.00	8.01	4.40	9.43	11.84
30x30/3	4.97	3.79	4.27	1.13	2.01	1.05	1.00	6.82	3.27	7.15	6.52
40x40/3	5.85	10.74	4.56	1.12	2.15	1.05	1.00	7.18	3.24	7.14	5.22
50x50/3	5.55	5.05	5.11	1.13	2.04	1.04	1.00	7.27	3.09	6.81	4.56
60x60/3	6.00	5.07	5.24	1.13	1.95	1.03	1.00	6.92	2.91	6.37	3.88
70x70/3	6.86	5.84	4.67	1.14	2.13	1.05	1.00	7.35	3.04	6.62	3.73
80x80/3	8.64	9.91	5.65	1.14	2.14	1.05	1.00	7.54	3.05	6.55	3.54
90x90/3	9.62	13.71	5.16	1.22	2.02	1.03	1.00	6.60	2.69	5.72	2.86
100x100/3	12.62	9.09	4.82	1.14	2.26	1.04	1.00	7.83	3.04	6.49	3.22

$SLU2$, $SLU3$, $DLU22$, $DIKH$, $DIKR$, and $DIKBA$) perform better on the Intel platform (Mdk and Win) than on Sun platform.

5.6.5.2 Random network generators

In this section, we choose the cases whose requested OD pairs have 75% $|N|$ distinct destinations for our discussion. Results for other cases using 25% $|N|$, 50% $|N|$, and 100% $|N|$ distinct destinations are listed in the appendix. All of these experiments are run on the Sun machine. The performances of $SLU2$ and $SLU3$ are always worse than $SLU1$. Similarly, $DLU22$ is worse than $DLU21$. Thus we only include $SLU1$ and $DLU21$ here for comparison.

SPGRID-SQ family: Table 11 shows that label-correcting codes $TWOQ$, $PAPE$ and BFP perform the best in this SPGRID-SQ family. Dijkstra-based codes such as $DIKBD$,

Table 12: 75% SPGRID-WL

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
16x64/3	5.66	3.50	4.38	1.09	2.05	1.05	1.00	6.22	3.88	8.42	9.69
16x128/3	5.70	3.59	5.15	1.12	1.99	1.05	1.00	5.82	3.61	8.20	8.12
16x256/3	5.65	4.46	4.90	1.10	2.03	1.05	1.00	5.76	3.70	8.49	7.83
16x512/3	7.11	3.18	5.60	1.09	2.04	1.03	1.00	5.41	3.53	8.33	7.24
64x16/3	3.93	3.69	4.81	1.13	2.22	1.06	1.00	8.39	3.25	6.78	5.60
128x16/3	3.38	3.76	4.97	1.15	2.02	1.03	1.00	8.46	2.90	5.78	3.68
256x16/3	3.39	4.85	4.86	1.12	1.93	1.03	1.00	9.61	2.98	5.77	3.27
512x16/3	3.51	5.00	5.06	1.15	1.80	1.04	1.00	10.62	2.97	5.64	3.02

Table 13: 75% SPGRID-PH

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
32x32/6	4.11	11.70	1.28	1.37	1.12	3.94	2.32	1.87	4.10	2.49	1.00
16x32/5	5.47	9.00	1.91	1.14	1.00	2.03	1.48	3.12	6.22	4.51	2.03
64x32/7	6.26	16.39	1.59	3.05	2.24	10.78	5.34	1.96	3.68	2.43	1.00
128x32/8	9.48	19.82	1.98	6.72	3.14	20.74	9.85	1.93	3.48	2.34	1.00
256x32/8	13.95	24.60	2.23	13.26	3.46	25.35	11.65	1.88	3.27	2.25	1.00

$DIKR$, and $DIKBA$ perform relatively worse for smaller networks. $DIKBD$ perform slightly worse than $THRESH$, but is the fastest Dijkstra's code. $SLU1$ and $DLU21$ perform similarly to $GOR1$ but become worse for larger networks.

SPGRID-WL family: Table 12 shows that label-correcting codes $TWOQ$, $PAPE$ and BFP perform the best in this SPGRID-WL family. $THRESH$ is only slightly worse than BFP. $DIKBD$ is the fastest Dijkstra's code, but $DIKBA$ catches up for larger LONG cases. $SLU1$ is faster in the LONG cases, $DLU21$ is faster in the WIDE cases, and they both are slightly better than $GOR1$. $DIKR$ performs the worst for the WIDE cases, but $DIKH$ performs the worst for the LONG cases.

SPGRID-PH family: Table 13 shows that $DIKBA$, $DIKH$, and $GOR1$ perform the best. $DIKR$, $DIKBD$ and $THRESH$ perform slightly worse but are still relatively better than the remaining codes. BFP, $TWOQ$, $SLU1$ and $PAPE$ perform worse when the network become larger. $DLU21$ perform the worst overall.

Table 14: 75% SPGRID-NH

Grid/deg	SLU 1	DLU 21	GOR 1	BFP	THRESH	PAPE	TWOQ	DIK H	DIK BD	DIK R	DIK BA
16x32/5	7.89	13.07	2.56	1.01	1.21	1.01	1.00	4.39	2.07	4.11	3.54
32x32/6	8.39	23.66	2.06	1.04	1.00	1.07	1.07	3.50	1.45	2.76	1.90
64x32/7	10.92	29.59	1.93	1.18	1.00	1.24	1.23	3.20	1.21	2.18	1.35
128x32/8	15.79	33.39	1.87	1.19	1.00	1.20	1.20	3.04	1.06	1.85	1.13

Table 15: 75% SPRAND-S4 and SPRAND-S16

$\|N\|$/deg	SLU 1	DLU 21	GOR 1	BFP	THRESH	PAPE	TWOQ	DIK H	DIK BD	DIK R	DIK BA
128/4	9.00	7.00	4.00	1.50	1.50	1.00	1.00	6.50	5.50	10.50	12.00
256/4	12.00	12.60	3.90	1.00	1.30	1.00	1.00	5.90	5.10	8.80	4.10
512/4	16.37	22.27	3.40	1.21	1.00	1.19	1.23	4.45	3.79	6.16	2.73
1024/4	36.87	66.55	3.81	1.77	1.00	1.77	1.81	4.10	4.48	5.23	1.77
2048/4	103.60	144.95	3.11	1.83	1.00	2.06	2.09	2.98	3.96	3.46	1.03
128/16	10.43	14.14	3.00	1.00	1.00	1.14	1.00	2.57	2.14	3.57	4.29
256/16	25.42	26.36	3.06	1.24	1.00	1.27	1.27	2.64	2.33	3.36	2.30
512/16	78.31	59.22	3.21	1.38	1.00	1.59	1.59	2.49	2.24	3.04	1.63
1024/16	159.01	147.26	2.82	1.43	1.00	2.01	1.92	1.98	2.01	2.31	1.19
2048/16	270.46	262.17	2.80	1.68	1.22	2.89	2.73	1.73	1.95	1.90	1.00

SPGRID-NH family: Table 14 shows that all label-correcting codes perform the best, followed by the label-setting codes. $SLU1$ and $DLU21$ perform the worst, especially for larger networks.

SPRAND-S family: Table 15 shows that label-correcting codes perform the best. Label-setting codes are slightly worse than label-correcting codes, and tend to perform relatively better for cases with larger degree. $SLU1$ and $DLU21$ perform the worst, especially for networks with more nodes and larger degrees.

SPRAND-D family: Table 16 shows that label-setting codes perform the best. Label-correcting methods such as $THRESH$, $GOR1$, and BFP perform slightly worse than the label-setting codes. For smaller networks (e.g. $|N| \leq 512$), $SLU1$ and $DLU21$ perform the worst. $PAPE$ and $TWOQ$ perform very well for smaller networks, but become the worst for larger and denser cases.

SPRAND-LENS family: Table 17 shows that label-setting codes perform the best for cases with smaller arc length. For larger arc length, label-correcting codes perform the best.

Table 16: 75% SPRAND-D4 and SPRAND-D2

$\|N\|$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	7.64	10.07	2.50	1.21	1.00	1.29	1.29	1.79	1.71	2.21	2.36
128/64	4.38	5.34	2.22	1.09	1.00	1.22	3.28	1.12	1.06	1.28	1.16
256/64	12.61	11.53	2.47	1.28	1.00	1.60	1.56	1.12	1.07	1.30	1.06
256/128	8.34	7.56	2.78	1.49	1.30	2.03	1.97	1.02	1.01	1.12	1.00
512/128	27.97	15.53	3.01	1.80	1.45	3.13	2.97	1.06	1.08	1.13	1.00
512/256	15.70	8.89	3.40	2.36	1.99	4.66	4.33	1.00	1.03	1.03	1.01
1024/256	30.46	22.95	3.81	3.14	2.61	8.41	7.02	1.00	1.10	1.05	1.05
1024/512	17.01	13.19	4.33	4.05	3.33	12.17	9.81	1.00	1.13	1.06	1.12
2048/512	19.60	15.53	4.35	5.29	4.01	19.17	13.31	1.00	1.07	1.01	1.11
2048/1024	10.30	8.10	4.99	7.62	5.46	30.65	19.84	1.00	1.06	1.00	1.15

Table 17: 75% SPRAND-LENS4

$\|N\|$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[1,1]$	4.03	4.34	2.72	5.79	5.10	11.31	9.24	2.41	1.00	2.14	1.86
	$[0,10]$	3.53	3.94	2.03	2.61	2.33	2.75	2.75	1.94	1.00	1.86	1.39
	$[0,10^2]$	5.17	5.39	2.43	1.30	1.00	1.61	1.48	2.87	1.83	3.22	1.96
	$[0,10^4]$	12.80	13.70	3.60	1.00	1.30	1.00	1.00	5.80	5.20	8.70	4.60
	$[0,10^6]$	14.00	14.89	4.44	1.11	1.44	1.00	1.00	6.56	9.89	12.67	7.78
1024/4	$[1,1]$	20.00	35.44	4.67	35.38	22.67	49.16	57.99	2.59	1.00	2.05	3.55
	$[0,10]$	18.74	31.19	3.81	22.92	16.26	33.66	31.79	2.22	1.00	1.91	1.13
	$[0,10^2]$	21.19	36.03	3.52	4.98	3.38	6.61	5.07	2.63	1.65	2.58	1.00
	$[0,10^4]$	36.58	60.04	3.63	1.68	1.00	1.72	1.72	4.04	4.20	5.06	1.61
	$[0,10^6]$	40.03	68.74	3.60	1.14	1.10	1.01	1.00	4.76	4.86	7.42	1.96

Both $SLU1$ and $DLU21$ perform the worst, especially for cases with more nodes and larger ranges of arc length.

SPRAND-LEND family: Table 18 shows that label-setting codes perform the best. $GOR1$ performs slightly worse than label-setting codes. Label-correcting codes except $GOR1$ perform well for cases with larger range of arc length, but become the worst for

Table 18: 75% SPRAND-LEND4

$\|N\|$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	$[1,1]$	8.43	7.57	3.19	7.57	7.33	36.57	24.05	1.00	1.10	1.05	1.76
	$[0,10]$	8.43	7.62	2.67	4.24	3.71	18.67	11.48	1.00	1.10	1.05	1.29
	$[0,10^2]$	9.16	8.42	2.37	2.05	1.74	6.16	4.47	1.00	1.00	1.05	1.21
	$[0,10^4]$	13.54	12.31	2.54	1.31	1.00	1.62	1.54	1.23	1.15	1.38	1.15
	$[0,10^6]$	12.29	11.36	2.43	1.21	1.14	1.57	1.57	1.14	1.50	1.43	1.00
1024/256	$[1,1]$	20.13	15.87	4.88	39.11	37.08	344.86	235.18	1.00	1.28	1.01	17.69
	$[0,10]$	21.49	16.24	4.49	20.73	17.26	226.33	122.45	1.00	1.31	1.01	4.36
	$[0,10^2]$	22.29	17.68	4.02	9.16	7.43	86.06	43.06	1.00	1.24	1.02	1.52
	$[0,10^4]$	30.57	23.29	3.84	3.29	2.66	8.18	6.87	1.00	1.12	1.05	1.07
	$[0,10^6]$	34.98	27.26	3.82	1.98	2.11	2.91	2.86	1.07	1.22	1.16	1.00

Table 19: 75% SPRAND-PS4

$\|N\|$/deg	P	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	0	11.70	12.60	3.70	1.00	1.30	1.00	1.00	5.70	5.20	8.70	4.60
	10^4	13.22	14.11	4.00	1.11	1.44	1.11	1.00	7.00	5.11	9.33	5.56
	10^5	12.80	13.50	3.90	1.10	2.20	1.00	1.00	13.00	5.00	8.80	7.20
	10^6	12.60	13.70	3.80	1.00	3.20	1.00	1.00	21.70	8.00	10.30	7.10
1024/4	0	37.17	65.28	3.64	1.65	1.00	1.64	1.65	4.00	4.18	5.04	1.71
	10^4	36.95	60.61	3.62	1.67	1.00	1.69	1.70	4.07	3.88	4.76	1.66
	10^5	28.90	47.06	2.69	1.24	1.00	1.27	1.27	4.63	3.36	3.45	1.56
	10^6	19.78	34.31	2.19	1.00	1.94	1.02	1.02	11.32	3.49	4.40	2.83

Table 20: 75% SPRAND-PD4

$\|N\|$/deg	P	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/64	0	13.62	11.92	2.54	1.23	1.00	1.54	1.54	1.15	1.15	1.38	1.15
	10^4	10.75	9.81	2.12	1.06	1.00	1.38	1.31	1.50	1.25	1.50	1.31
	10^5	10.29	9.24	1.94	1.00	1.65	1.24	1.18	2.65	1.53	1.88	1.65
	10^6	10.24	9.24	2.00	1.00	1.82	1.24	1.24	3.24	1.88	2.06	1.88
1024/256	0	28.76	22.75	3.75	3.21	2.66	8.31	6.97	1.00	1.10	1.05	1.06
	10^4	17.02	13.59	2.26	1.85	1.62	4.78	4.13	1.07	1.02	1.01	1.00
	10^5	9.26	6.98	1.18	1.00	1.56	2.49	2.11	2.07	1.70	1.68	1.66
	10^6	9.07	6.98	1.18	1.00	2.15	2.52	2.15	2.81	2.06	2.05	2.00

cases with smaller arc length. $SLU1$ and $DLU21$ perform worse for cases with more nodes and larger range of arc length.

SPRAND-PS family: Table 19 shows that the label-correcting codes perform the best, and are "potential-invariant" [74]. When the range of node potential P increases, there will be more arcs with negative lengths but not negative cycles which slow down the label-setting codes. Although $SLU1$ and $DLU21$ perform the worst overall, they perform relatively better when P increases.

SPRAND-PD family: Table 20 shows that both label-setting and label-correcting codes perform best for these dense families. $SLU1$ and $DLU21$ perform the worst, although their performances become relatively better when P increases.

NETGEN-S family: Table 21 shows that label-correcting codes are the best, followed by the label-setting codes. $SLU1$ and $DLU21$ perform better than label-setting codes for small networks with smaller degrees; they become significantly worse for larger cases.

Table 21: 75% NETGEN-S4 and NETGEN-S16

$\|N\|$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	8.00	6.50	4.50	1.00	2.00	1.00	1.00	6.50	5.50	10.50	13.00
256/4	13.44	14.44	4.11	1.00	1.33	1.11	1.00	6.67	4.44	9.11	6.33
512/4	15.60	20.75	3.68	1.12	1.07	1.00	1.00	4.82	2.93	6.14	3.21
1024/4	44.18	71.50	3.79	1.18	1.04	1.00	1.00	4.50	2.43	5.21	2.00
2048/4	152.32	190.10	3.75	1.21	1.09	1.01	1.00	4.53	2.00	4.74	1.50
128/16	7.62	8.00	2.25	1.00	1.00	1.12	1.12	2.25	2.00	3.00	2.62
256/16	16.53	17.72	3.03	1.36	1.00	1.44	1.44	2.44	1.75	2.86	1.97
512/16	42.73	39.44	3.30	1.53	1.00	1.67	1.67	2.44	1.56	2.65	1.50
1024/16	141.41	136.49	3.36	1.59	1.00	1.80	1.82	2.32	1.30	2.33	1.28
2048/16	318.70	305.86	3.52	1.64	1.00	1.90	1.89	2.39	1.23	2.22	1.16

Table 22: 75% NETGEN-D4 and NETGEN-D2

$\|N\|$/deg	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	5.24	6.06	2.35	1.12	1.00	1.24	1.24	1.47	1.24	1.82	1.65
256/64	11.84	10.71	2.75	2.31	1.52	2.07	1.99	1.27	1.00	1.32	1.13
512/128	30.63	18.02	3.55	1.93	2.12	2.97	3.88	1.20	1.00	1.18	1.06
1024/256	33.91	26.59	4.04	2.39	2.66	4.01	3.89	1.09	1.00	1.09	1.03
2048/512	22.19	18.20	4.31	2.64	3.08	3.97	3.86	1.03	1.00	1.03	1.01
128/64	3.55	4.27	2.24	1.18	1.21	1.48	1.52	1.06	1.00	1.24	1.12
256/128	9.33	8.08	3.38	1.92	2.14	2.91	4.81	1.15	1.00	1.20	1.13
512/256	17.46	10.30	3.87	2.21	2.55	3.71	3.57	1.11	1.00	1.09	1.02
1024/512	16.59	12.62	4.25	2.65	3.09	4.85	4.65	1.00	1.01	1.05	1.03

NETGEN-D family: Table 22 shows that label-setting codes perform the best, followed by the label-correcting codes. $SLU1$ and $DLU21$ perform the worst, but are relatively better for denser cases.

NETGEN-LENS family: Table 23 shows that label-correcting codes perform the best, followed by the label-setting codes. $SLU1$ and $DLU21$ perform the worst, especially for larger networks. Label-setting codes perform worse when the range of arc lengths becomes

Table 23: 75% NETGEN-LENS4

$\|N\|$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
256/4	$[0,10]$	13.67	13.44	4.44	1.11	1.44	1.00	1.00	6.22	1.89	3.56	1.67
	$[0,10^2]$	15.25	16.62	5.12	1.25	1.62	1.00	1.12	7.25	2.75	6.38	2.25
	$[0,10^4]$	14.44	15.11	4.33	1.11	1.44	1.00	1.00	6.33	4.33	9.33	7.44
	$[0,10^6]$	13.22	13.56	4.22	1.11	1.33	1.00	1.00	6.67	10.44	12.67	9.00
1024/4	$[0,10]$	47.88	72.66	3.60	1.21	1.02	1.00	1.00	4.24	1.23	2.11	1.16
	$[0,10^2]$	43.86	71.94	3.64	1.18	1.05	1.01	1.00	4.40	1.48	2.99	1.28
	$[0,10^4]$	46.57	75.32	3.65	1.18	1.06	1.01	1.00	4.51	2.40	5.13	1.96
	$[0,10^6]$	61.76	94.82	3.64	1.19	1.03	1.00	1.01	4.31	4.27	6.70	2.10

Table 24: 75% NETGEN-LEND4

$\|N\|$/deg	$[L,U]$	SLU 1	DLU 21	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
256/64	$[0,10]$	14.40	13.50	2.50	1.20	1.30	1.20	1.20	1.30	1.10	1.20	1.00
	$[0,10^2]$	13.18	12.18	3.09	1.73	1.73	2.18	2.27	1.36	1.00	1.18	1.00
	$[0,10^4]$	11.46	10.46	2.69	1.38	1.46	2.00	1.92	1.23	1.00	1.23	1.08
	$[0,10^6]$	9.53	8.87	2.40	1.27	1.27	1.67	1.60	1.00	1.60	1.27	1.00
1024/256	$[0,10]$	35.28	27.80	2.22	1.03	1.12	1.04	1.04	1.00	1.00	1.03	1.00
	$[0,10^2]$	33.84	27.00	3.77	2.40	2.78	2.90	2.90	1.06	1.01	1.04	1.00
	$[0,10^4]$	34.23	26.56	4.11	2.41	2.72	4.60	3.95	1.07	1.00	1.08	1.03
	$[0,10^6]$	33.04	25.63	3.88	2.23	2.51	3.77	3.65	1.04	1.10	1.13	1.00

Table 25: 75% SPACYC-PS4 and SPACYC-PS16

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THRESH	PAPE	TWOQ	DIKH	DIKBD	DIKR	DIKBA
128/4	$[0,10^4]$	1.50	1.00	2.50	1.00	2.00	2.50	1.50	2.00	4.00	1.50	4.00	7.50
256/4	$[0,10^4]$	1.00	1.22	2.22	1.11	1.89	2.00	1.56	1.78	3.67	1.67	3.33	3.89
512/4	$[0,10^4]$	1.00	1.38	2.90	1.36	2.74	2.17	2.21	2.50	3.50	1.29	2.74	2.26
1024/4	$[0,10^4]$	1.00	1.47	4.18	1.41	3.06	2.18	2.47	2.82	3.82	1.24	2.59	1.71
2048/4	$[0,10^4]$	1.00	1.31	4.29	1.30	3.38	2.27	2.78	2.95	3.61	1.06	2.27	1.18
128/16	$[0,10^4]$	1.67	1.00	3.67	2.00	2.67	2.33	2.33	2.67	3.00	2.33	3.33	7.00
256/16	$[0,10^4]$	1.11	1.61	3.00	1.00	2.17	1.72	2.06	2.17	2.28	1.28	2.00	2.50
512/16	$[0,10^4]$	1.00	1.50	3.85	1.15	2.69	1.97	2.87	2.86	2.28	1.17	1.81	1.63
1024/16	$[0,10^4]$	1.00	1.51	5.94	1.14	3.20	2.17	3.43	3.54	2.40	1.03	1.71	1.37
2048/16	$[0,10^4]$	1.13	1.45	6.90	1.24	3.77	2.45	4.07	4.19	2.37	1.00	1.57	1.11

larger. It it not clear how the range of arc lengths affects $SLU1$ and $DLU21$.

NETGEN-LEND family: Table 24 shows that label-setting codes perform the best, followed by the label-correcting codes. $SLU1$ and $DLU21$ perform the worst, especially for larger networks. When the range of arc lengths becomes larger, $SLU1$ and $DLU21$ perform slightly better.

SPACYC-PS family: Table 25 shows that $SLU1$ and $GOR1$ perform the best. $DLU21$ performs relatively worse for cases with more nodes and larger degrees. On the other hand, label-setting codes perform relatively worse for cases with fewer nodes and smaller degrees.

SPACYC-NS family: Table 26 shows that $SLU1$ and $GOR1$ perform the best, followed by the $DLI21$. Other label-correcting algorithms perform much worse. All label-setting codes are even worse than the label-correcting algorithms.

Table 26: 75% SPACYC-NS4 and SPACYC-NS16

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/4	$[-10^4, 0]$	1.00	4.00	5.00	6.00	5.00	9.00	7.00	6.00	80.00	11.00	14.00	12.00
256/4	$[-10^4, 0]$	1.00	1.20	2.60	1.20	2.30	4.50	3.40	3.20	41.50	6.80	8.90	6.90
512/4	$[-10^4, 0]$	1.00	1.19	2.65	1.15	5.98	9.25	7.94	8.42	132.42	12.27	14.88	12.44
1024/4	$[-10^4, 0]$	1.00	1.60	4.80	1.47	16.93	26.13	17.40	22.13	363.00	34.20	40.60	34.67
2048/4	$[-10^4, 0]$	1.00	1.23	4.24	1.24	21.89	36.01	36.21	46.62	848.86	46.51	55.58	47.21
128/16	$[-10^4, 0]$	1.00	1.40	1.80	1.40	3.00	6.40	7.20	6.20	43.80	7.00	8.40	7.60
256/16	$[-10^4, 0]$	1.00	1.61	3.00	1.22	4.78	10.67	16.39	11.78	131.44	10.89	13.00	11.39
512/16	$[-10^4, 0]$	1.00	1.44	3.82	1.22	11.55	31.16	57.77	47.94	521.17	28.78	32.91	29.39
1024/16	$[-10^4, 0]$	1.00	1.42	5.95	1.05	17.84	44.16	72.13	61.92	569.24	41.95	47.55	42.76
2048/16	$[-10^4, 0]$	1.00	1.33	6.72	1.13	30.49	82.06	148.13	135.75	941.90	72.38	82.54	74.28

Table 27: 75% SPACYC-PD4 and SPACYC-PD2

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[0, 10^4]$	1.00	1.38	2.50	1.12	1.75	1.75	1.62	1.75	1.38	1.25	1.62	2.25
256/64	$[0, 10^4]$	1.20	1.55	3.29	1.12	2.51	2.00	2.78	2.82	1.29	1.00	1.24	1.49
512/128	$[0, 10^4]$	1.51	1.95	6.01	1.51	3.54	2.68	4.69	4.64	1.28	1.00	1.16	1.17
1024/256	$[0, 10^4]$	1.77	3.84	12.65	1.67	4.17	3.14	5.89	5.87	1.15	1.00	1.11	1.06
2048/512	$[0, 10^4]$	1.76	4.04	12.06	1.66	4.40	3.28	6.06	5.90	1.06	1.00	1.03	1.04
128/64	$[0, 10^4]$	1.00	1.36	2.27	1.27	1.91	1.73	2.00	1.91	1.36	1.09	1.36	2.18
256/128	$[0, 10^4]$	1.49	1.58	3.45	1.45	3.03	2.46	3.61	3.58	1.17	1.00	1.15	1.27
512/256	$[0, 10^4]$	1.75	1.81	5.61	1.64	3.61	3.00	4.73	4.64	1.15	1.00	1.09	1.12
1024/512	$[0, 10^4]$	1.88	3.87	11.15	1.73	3.83	3.73	5.56	5.35	1.06	1.00	1.04	1.06
2048/1024	$[0, 10^4]$	1.75	2.80	8.54	1.66	3.99	3.53	5.31	5.13	1.01	1.00	1.02	1.04

SPACYC-PD family: Table 27 shows that all label-setting codes and $GOR1$ perform the best, followed by the $SLU1$ which is slightly worse for larger cases. Label-correcting codes also perform well, less than 5 times slower than ACC. $DLU21$ performs worse when the network become larger. Both $SLU1$ and $DLU21$ perform relatively better for denser cases.

Table 28: 75% SPACYC-ND4 and SPACYC-ND2

$\|N\|$/deg	$[L,U]$	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/32	$[-10^4, 0]$	1.00	1.38	2.38	1.12	3.25	8.00	12.50	7.25	62.00	8.12	9.38	8.38
256/64	$[-10^4, 0]$	1.00	1.24	2.71	1.06	5.96	16.13	44.00	19.93	250.69	14.38	15.76	14.82
512/128	$[-10^4, 0]$	1.00	1.26	3.85	1.05	11.63	32.61	179.11	58.14	943.02	27.44	28.58	27.74
1024/256	$[-10^4, 0]$	1.00	2.11	7.31	1.03	21.87	66.24	708.88	183.48	3312.01	55.39	56.49	55.65
128/64	$[-10^4, 0]$	1.00	1.21	2.07	1.14	3.79	10.07	15.07	8.71	66.00	8.36	9.43	8.71
256/128	$[-10^4, 0]$	1.00	1.00	2.25	1.04	6.24	17.86	49.41	20.86	177.84	15.42	16.36	15.80
512/256	$[-10^4, 0]$	1.00	1.01	3.10	1.04	12.66	40.90	267.88	82.04	992.15	33.17	33.93	33.40
1024/512	$[-10^4, 0]$	1.00	2.06	6.26	1.04	23.91	75.98	703.17	233.57	4211.82	64.92	65.05	64.90

Table 29: 75% SPACYC-P2N128 and SPACYC-P2N1024

$\|N\|$/deg	$[L,U]$ ×1000	ACC	SLU 1	DLU 21	GOR 1	BFP	THR ESH	PA PE	TWO Q	DIK H	DIK BD	DIK R	DIK BA
128/16	$[0,10]$	1.78	2.75	4.59	1.84	1.00	1.34	1.00	3.44	3.22	2.69	4.84	7.66
	$[-1,9]$	1.40	2.12	3.70	1.57	1.02	1.30	1.05	1.00	2.83	2.02	3.42	5.65
	$[-2,8]$	1.00	1.51	2.58	1.14	1.03	1.37	1.08	1.07	3.61	1.97	2.66	4.92
	$[-3,7]$	1.00	1.49	2.38	1.03	1.54	2.27	2.06	1.92	6.83	3.00	3.79	5.70
	$[-4,6]$	1.00	1.47	2.44	1.00	2.67	4.69	5.83	4.47	26.88	5.73	7.00	8.03
	$[-5,5]$	1.00	1.39	2.36	1.11	3.56	7.38	10.68	6.58	64.11	8.35	10.23	10.70
	$[-6,4]$	1.00	1.44	2.44	1.06	3.84	7.73	12.06	7.42	66.14	9.38	11.34	11.64
	$[-10,0]$	1.00	1.41	2.60	1.14	5.73	13.51	21.16	16.16	189.81	15.10	17.95	15.79
1024/16	$[0,10]$	1.76	2.31	8.55	1.66	1.03	1.03	1.00	1.00	2.83	1.90	3.48	2.00
	$[-1,9]$	1.29	1.97	7.35	1.44	1.09	1.00	1.09	1.06	2.85	1.53	2.62	1.71
	$[-2,8]$	1.00	1.39	5.16	1.04	1.22	1.29	1.37	1.39	4.69	1.67	2.37	2.12
	$[-3,7]$	1.00	1.35	4.96	1.04	2.73	3.84	3.82	4.18	24.27	4.90	5.98	5.39
	$[-4,6]$	1.00	1.45	5.55	1.26	7.72	14.04	16.51	14.40	90.43	17.43	20.72	18.51
	$[-5,5]$	1.00	1.44	6.23	1.10	16.69	39.44	58.19	57.38	479.52	45.75	52.42	46.75
	$[-6,4]$	1.00	1.39	5.41	1.10	28.39	72.41	137.86	113.71	959.76	82.71	95.02	85.14
	$[-10,0]$	1.00	1.42	5.48	1.15	39.25	97.81	162.62	88.94	877.62	118.56	135.33	121.50

SPACYC-ND family: Table 28 shows that $GOR1$ and $SLU1$ perform the best, followed by $DLU21$, which is a little worse for larger cases. All other codes (especially $DIKH$ and $PAPE$) perform significantly worse, especially for larger cases.

SPACYC-P2N family: Table 29 shows that $GOR1$ and $SLU1$ perform the best. When $[L,U]$ is larger than $[-3000,7000]$, the other label-correcting and label-setting codes perform well and slightly better than $DLU21$. However, when $[L,U]$ is smaller than $[-3000,7000]$, these SSSP codes (except $GOR1$) perform significantly worse. The more negative the arc lengths are, the worse they become.

5.7 Summary

After these comprehensive computational experiments, we have several observations.

1. Among the eight different pivoting rules for sparsity, the dynamic Markowitz and its variant that employs simple tie-breaking technique usually produce fewer fill-ins than other pivoting rules.

2. Among the three different SLU implementations, the bucket implementation $SLU1$ outperforms the other two implementations that are based on binary heaps, for all

cases. Similarly, the bucket implementation of $DLU2$, $DLU21$, has better performance than the heap implementation.

For general larger networks, $SLU1$ and $DLU21$ can not compete with the best of the state-of-the-art SSSP codes , except for acyclic networks. Nevertheless, $SLU1$ and $DLU21$ do not perform worst for all cases.

For dense graphs, usually $DLU21$ performs better than $SLU1$; for sparse graphs, usually $SLU1$ is better than $DLU21$. Unlike other SSSP algorithms which perform consistently regardless the number of distinct destinations, $SLU1$ and $DLU21$ perform relatively better for MPSP problems of more distinct requested destinations than for problems of fewer distinct requested destinations.

3. The Asia-Pacific flight network is the only real-world network tested in this thesis. Although it is sparse (112 nodes, 1038 arcs), $DLU21$ performs better than $SLU1$, which in turn outperforms all of the implemented label-setting codes. The label-correcting codes perform similarly to $SLU1$ and $DLU21$. The Floyd-Warshall algorithm is in no case competitive with other SSSP algorithms or our proposed MPSP algorithms.

4. In most SPGRID families, the label-correcting codes usually perform the best, except for the SPGRID-PH family for which the label-setting codes perform the best. $SLU1$ and $DLU21$ have better performance on the SPGRID-SQ and SPGRID-WL families than other SPGRID families.

   For each SPGRID family, label-setting codes usually perform relatively worse on smaller networks.

5. $SLU1$ and $DLU21$ are usually slower than other label-correcting and label-setting codes for most SPRAND families. Label-correcting codes perform better for most of the sparse SPRAND families, but label-setting codes perform better for most dense SPRAND families.

   When the range of arc lengths decreases (e.g., ≤ 10), label-correcting codes tend to perform much worse. When the range of arc lengths increases on sparse graphs, label-setting codes will perform only slightly worse.

6. $SLU1$ and $DLU21$ are usually slower than other label-correcting and label-setting codes for most NETGEN families. Label-correcting codes perform better for most sparse NETGEN families, but label-setting codes perform better for most dense NETGEN families.

   When the range of arc lengths increases on sparse graphs, label-setting codes tend to perform a little worse.

7. $SLU1$ and $GOR1$ usually outperform other codes for all SPACYC families except the SPACYC-PD family, for which the label-setting codes perform asymptotically the best. $DLU21$ performs asymptotically the worst for cases whose arc lengths are all positive. However, all label-correcting codes (except $GOR1$) and label-setting codes perform significantly worse for cases with negative arc lengths.

5.7.1 Conclusion and future research

Although our computational experiments are already extensive, more thorough tests may still be required to draw more solid conclusions. There are too many factors that may affect the performance of MPSP algorithms, such as requested OD pairs, arc lengths, network topologies, and node orderings.

In our experiments, for each test case, we generate only one set of requested OD pairs. Different requested OD pairs may affect the performance since the distribution of requested pairs in the OD matrix will not affect the SSSP algorithms but may affect our MPSP algorithms.

By specifying the same numbers of nodes and arcs but different random seeds, we may generate several different test cases with the same topology but different arc lengths. A more thorough experiment could generate several such networks and test all of the algorithms on these cases to compute their average performance. Due to time constraints, in this chapter we use only one random seed for each topology.

Different node orderings will significantly affect our MPSP algorithms. In this chapter, we choose a node ordering aimed at reducing the total fill-ins in the LU factorization. As discussed in Section 4.4, another ordering consideration is to group the requested OD pairs

and assign them higher indices. This is difficult to test for randomly generated networks and requested OD pairs. It may be worth trying for some real-world applications where both the network topology and requested OD pairs are fixed.

Our limited results show that our MPSP algorithms seem to perform worse for general larger networks. This may make sense since after all SLU and DLU are algebraic algorithms, although we create graphical implementations for testing. In particular, SLU and DLU are more suitable for cases whose arc lengths are randomly distributed for all cases since they are designed to work independent of arc lengths. They are based on the ordering of triple comparisons. The sequence of triple comparisons is fixed and unavoidable in our MPSP algorithms. Thus, even if there exists some arc with a very large length, our algorithms could not avoid some trivial triple comparisons involving this arc, although intuitively we know in advance these operations are wasteful. Conventional SSSP algorithms, on the other hand, are based more on the comparison of arc lengths. For example, the greedy algorithm (Dijkstra's) only does triple comparisons on nodes with the smallest distance label in each iteration. Adding some "greedy" ideas to our MPSP algorithms might help them avoid many wasteful operations. However, it is still not clear how to integrate these two different methodologies.

If we suspect our MPSP algorithms may perform worse than conventional SSSP algorithms for most cases, we can design an experiment by generating the requested OD pairs most favorable to our MPSP algorithms. In particular, since both SLU and DLU can compute optimal distance faster for OD pairs with larger indices, the most favorable settings for our MPSP algorithms are: (a) find an optimal (or a very good) sparse node ordering, and (b) in that ordering, generate k requested OD pairs that span the k entries in the OD matrix in a line perpendicular to the diagonal entries. That is, entries $(n, n-k+1)$, $(n-1, n-k+2)$, $\ldots$, $(n-k+2, n-1)$, $(n-k+1, n)$ will be the set of requested OD pairs. Such a setting will force the conventional SSSP to solve k SSSP problems, but will be most advantageous for our MPSP algorithms since they are not only in the sparse ordering but are also the closest to the "southeastern" corner of the OD matrix in the new ordering.

If experiments on many random graphs with such settings still show that our MPSP algorithms do not outperform the conventional SSSP algorithms, we would conclude that our MPSP algorithms are indeed less efficient.

The experiments in this chapter are just a first step in evaluating algorithmic efficiency when solving general MPSP problems. More thorough experiments will be done in the future.