CHAPTER III

SHORTEST PATH ALGORITHMS: 1-ALL, ALL-ALL, AND SOME-SOME

In this thesis, we focus on solving ODMCNF, a class of min-cost MCNF problems where the commodities represent origin-destination (OD) pairs.

As introduced in Section 2.3.2, when we use Dantzig-Wolfe decomposition and column generation to solve ODMCNF, we generate new columns by solving sequences of shortest path problems for specific commodities (OD pairs) using $c+\pi^*$ as the new arc lengths where c is the original arc cost vector and $-\pi^*$ is the optimal dual solution vector associated with the bundle constraint after we solve the RMP. Since the RMP changes at each iteration of the DW procedures, π^* will also change. Therefore, the shortest paths between specific OD pairs have to be repeatedly computed with different arc costs on the same network. Efficient shortest path algorithms will speed up the overall computational time needed to solve the ODMCNF.

In this chapter, we first survey shortest path algorithms that have appeared in the literature, and then discuss the advantages and disadvantages of these shortest path algorithms in solving multiple pairs shortest path problems. Finally, we introduce a new shortest path algorithm based on a new LP solution technique that follows the primal-dual algorithmic framework to solve sequences of nonnegative least squares (NNLS) problems, and show its connection to the well-known Dijkstra's algorithm.

3.1 Overview on shortest path algorithms

During the last four decades, many good shortest path algorithms have been developed. We can group shortest path algorithms into 3 classes:

- those that employ combinatorial or network traversal techniques such as *label-setting methods*, *label-correcting methods* and their hybrids

- those that employ Linear Programming (LP) based techniques like primal network simplex methods and dual ascent methods
- those that use algebraic or matrix techniques such as Floyd-Warshall [113, 304] and Carré's [64, 65] algorithms.

The first two groups of shortest path algorithms are mainly designed to solve the *Single Source (or Sink) Shortest Path* (SSSP) problem, which is the problem of computing the shortest path tree (SPT) for a specific source (or sink) node. Algebraic shortest path algorithms, on the other hand, are more suitable for solving the *All Pairs Shortest Paths* (APSP) problem, which is the problem of computing shortest paths for all the node pairs.

3.2 Notation and definition

For a *digraph* $G := (N, A)$ with $n = |N|$ nodes and $m = |A|$ arcs, a *measure matrix* C is the $n \times n$ matrix in which element c_{ij} denotes the length of arc (i, j) with *tail* j and *head* i. $c_{ij} := \infty$ if $(i, j) \notin A$. A *walk* is a sequence of r nodes $(n_1, n_2.., n_r)$ composed of $(r-1)$ arcs, (n_{k-1}, n_k), where $2 \leq k \leq r$ and $r \geq 2$. A *path* is a walk without repeated nodes so that all n_k are different. A *cycle* is a walk where all n_k are different except that the starting and ending nodes are the same; that is, $n_1 = n_r$. The length of a path (cycle) is the sum of lengths of its arcs. When we refer to a shortest path tree with root t, we mean a tree rooted at a sink node t where all the tree arcs point towards to t.

The *distance matrix* X is the $n \times n$ array with element x_{ij} as the length of the shortest path from i to j. Let $[succ_{ij}]$ denote the $n \times n$ *successor matrix*. That is, $succ_{ij}$ represents the node that immediately follows i in the shortest path from i to j. We could construct the shortest path from i to j by tracing the successor matrix. In particular, the shortest path from i to j is $i \rightarrow k_1 \rightarrow k_2 \rightarrow \ldots \rightarrow k_r \rightarrow j$, where $k_1 = succ_{ij}$, $k_2 = succ_{k_1 j}, \ldots,$ $k_r = succ_{k_{r-1} j}$, and $j = succ_{k_r j}$. If node i has a successor j, we say node i is the *predecessor* of node j. Let x^*_{ij} and $succ^*_{ij}$ denote the shortest distance and successor from i to j in G.

We say that node i is *higher* (*lower*) than node j if the index $i > j$ ($i < j$). A node i is said to be the *highest* (*lowest*) node in a node set $LIST$ if $i \geq k$ ($i \leq k$) $\forall\ k \in LIST$ (see Figure 4 in Section 4.1).

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

Given two arcs (s, k) and (k, t), a *triple comparison* $s \to k \to t$ compares $c_{sk} + c_{kt}$ with c_{st}. A *fill-in* happens when there is no arc (s, t) (i.e., $c_{st} = \infty$) but the triple comparison $s \to k \to t$ makes $c_{sk} + c_{kt} < \infty$.

Path algebra is an ordered semiring $(S, \oplus, \otimes, e, \emptyset, \preceq)$, with $S = \mathbb{R} \cup \{\infty\}$ and two binary operations defined as in Table 5.

Table 5: Path algebra operators

generalized addition ($\oplus$)	$a \oplus b$	$=$	$\min\{a, b\}$	$\forall a, b \in S$
generalized multiplication($\otimes$)	$a \otimes b$	$=$	$a + b$	
unit element e has value 0				
null element $\emptyset$ has value ∞				
the ordering $\preceq$ is the usual ordering ($\leq$) for real numbers.				

Path algebra obeys the commutative, associative and distributive axioms. (See [108, 65, 21, 63, 274, 275] for details.) We can also define generalized addition ($\oplus$) and generalized multiplication ($\otimes$) of matrices with elements in S as follows: Suppose $A, B, C, D \in S^{n \times n}$, where $A = [a_{ij}]$, $B = [b_{ij}]$, then $A \oplus B = C$, and $A \otimes B = D$ satisfy $c_{ij} = a_{ij} \oplus b_{ij}$ and $d_{ij} = \sum_{k=1}^{n} (a_{ik} \otimes b_{kj})$, where the symbol $\sum$ denotes a generalized addition.

Many shortest path algorithms in the literature can be viewed as methods for solving the linear systems defined on the path algebra, and will be reviewed in Section 3.4.

3.3 Single source shortest path (SSSP) algorithms

The Single Source (or Sink) Shortest Path (SSSP) problem determines the shortest path tree (SPT) for a specific source (or sink) node. SSSP algorithms in the literature can be roughly grouped into two groups: (1) Combinatorial algorithms which build the SPT based on combinatorial or graphical properties. (2) LP-based algorithms which view the SPT problem as a LP problem and solve it by specialized LP techniques.

In this section we also introduce a new algorithm of Thorup and give a summary of

previous computational experiments for many SSSP algorithms.

3.3.1 Combinatorial algorithms

The basic idea of the combinatorial SSSP algorithms is that, creating a node bucket (named LIST, usually initiated by the root node), the algorithms select a node from LIST, scan the node's outgoing arcs, update the distance labels for the endnodes of these outgoing arcs, put the updated nodes into LIST, and then choose another node from LIST. The procedures repeat until LIST becomes empty.

The differences between combinatorial shortest path algorithms are in the ways of maintaining node candidates to be scanned from LIST. Many sophisticated data structures have been used to improve the time bounds.

Label-setting methods, first proposed by Dijkstra [95] and Dantzig [85], choose the node with the smallest distance label. Methods that use special data structures to quickly choose the min-distance-label node have been developed. These data structures include the binary heap of Johnson [186], Fibonacci heap of Fredman and Tarjan [121], Dial's bucket [93], and radix heap of Ahuja et al. [4]. Many other complicated bucket-based algorithms have been suggested in the long computational survey paper by Cherkassky et al. [74].

On the other hand, label-correcting methods, first proposed by Ford [114], Moore [243], Bellman [40] and Ford and Fulkerson [117], have more flexibility in choosing a node from LIST. These data structures include the queue of Bellman [40], dequeue of Pape [264] and Levit [220], two-queue of Pallottino [260], dynamic bread-first-search of Goldfarb et al. [146], small-label-first (SLF) or large-label-last (LLL) of Bertsekas [44], and topological ordering of Goldberg and Radzik [140]. Yen [308] also gives another special node scanning procedure which eliminates approximately half of the node label updating operations of Bellman's algorithm.

Some efficient label-correcting algorithms [264, 220, 260, 140] are based on the following heuristic: suppose a node i in LIST has been scanned in previous iterations, and some of its successors, say, nodes $i_l, \ldots, i_k$, are also in LIST. Then it is better to choose node i to scan before choosing nodes $i_l, \ldots, i_k$. The rationale is, if node i_l, a successor of node i, is chosen

before node i to scan, then later when node i is updated and scanned, node i_l will have to be reupdated. Thus choosing node i rather than choosing any of its successors tends to save label updating operations. However, such heuristics may not guarantee a polynomial time algorithm. For example, the dequeue implementation of [264, 220] is very efficient in practice but has a pseudopolynomial theoretical running time. The other algorithms [260, 140] guarantee polynomial time bounds, but perform worse in practice.

The threshold algorithm of Glover et al. [134] combines techniques from both label-setting and label-correcting algorithms. In particular, the algorithm maintains two queues: NEXT and NOW. Nodes are put into NOW from NEXT if their distance label are less than the threshold, and the algorithm scans nodes in NOW and puts new nodes to NEXT. After NOW is empty, a new threshold is set and the algorithm iterates until finally NEXT becomes empty. The performance of this algorithm is sensitive to the way of adjusting threshold value; thus a fine-tuning procedure may be required to achieve better efficiency.

Although label-setting methods have more favorable theoretical time bounds than label-correcting methods, their overhead in sorting the node candidates may worsen their empirical performance, especially for sparse networks. The current best time bound of combinatorial algorithms to solve SSSP is $O(\min\{m + n\log n,\ m\log\log C,\ m + n\sqrt{\log C}\})$ obtained by [121],[184] and [4] for cases with nonnegative arc lengths, and $O(mn)$ by most label-correcting algorithms for cases with general arc lengths and no negative cycles (see Chapter 4 and Chapter 5 of [3]).

3.3.2 LP-based algorithms

Suppose we are solving an ALL-1 shortest path problem which determines a SPT rooted at sink node t. This ALL-1 SSSP can be formulated as the following LP:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} = Z^{P*}_{ALL-1}(x) \tag{SSSP}$$

$$s.t. \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} = b_i = \begin{cases} 1 & , \text{if } i \neq t \\ -(n-1) & , \text{if } i = t \end{cases} \quad \forall i \in N \tag{3.1}$$

$$x_{ij} \geq 0 \quad \forall (i,j) \in A$$

In particular, the problem can be viewed as if every node other than t (a total of $n-1$ nodes) sends one unit of flow to satisfy the demand $(n-1)$ of the root node t. The constraint coefficient matrix of (3.1) is the node-arc incidence matrix $\widetilde{N}$ introduced in Section 1.2. It is totally unimodual and guarantees that the LP solution of SSSP will be integral due to the integral right hand side. Moreover, because there exists a redundant constraint in SSSP, we can remove the last row of (3.1) to get a new coefficient matrix $\overline{N}$ and obtain the following new LP:

$$\min cx = Z^{P*}_{ALL-1}(x) \qquad \text{(ALL-1-Primal)}$$

$$s.t.\ \ \overline{N}x = 1\ \forall i \in N \setminus t \qquad (3.2)$$

$$x \geq 0$$

whose dual is

$$\max \sum_{i \in N \setminus t} \pi_i = Z^{D*}_{ALL-1}(\pi) \qquad \text{(ALL-1-Dual)}$$

$$s.t.\ \ \pi_i - \pi_j \leq c_{ij}\ \forall (i,j) \in A, i, j \neq t \qquad (3.3)$$

$$\pi_i \leq c_{it}\ \forall (i,t) \in A \qquad (3.4)$$

$$-\pi_j \leq c_{tj}\ \forall (t,j) \in A \qquad (3.5)$$

This LP thus can be solved by a specialized simplex method, usually called the network simplex method.

In network simplex method, a tree corresponds to a basis, and each dual variable π_i corresponds to a distance label associated with node i (thus $\pi_t = 0$). Let the reduced cost associated with an arc (i,j) be $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$. Then starting from the root r, we can set π such that every tree arc has zero reduced cost.

Primal network simplex starts with a spanning tree $T(t)$ rooted at t which can easily be constructed by any tree-search algorithm, and then identifies some non-tree arc (u,v) with $c^{\pi}_{uv} < 0$. Adding (pivoting in) the non-tree arc (u,v) to the tree $T(t)$ will create a cycle with negative reduced cost. The algorithm sends flow along the cycle, identifies (pivots out) a tree arc (u,w) to become a non-tree arc [145], and updates the duals associated

with the subtree rooted at node u so that the reduced cost for all tree arcs remains 0. The algorithm iterates these procedures until no more non-tree arc has negative reduced cost, which means optimality has been achieved. Notice that in each iteration, the objective is strictly improved, which means each iteration is a nondegenerate pivot.

In fact, the label-setting and label-correcting methods in Section 3.3.1 can be viewed as variants of primal network simplex methods [94]. In particular, if we add artificial arcs which have a very large cost M from all the other nodes to the root t, the primal network simplex method which chooses the most negative reduced cost non-tree arc will correspond to the Dijkstra's algorithm. The label-correcting methods may be viewed as "quasi-simplex" algorithms since they identify a profitable non-tree arc, but only update the dual for one node instead of all the nodes in its rooted subtree. Dial et al. [94] suggests that the superiority of the "quasi-simplex" methods to the "full-simplex" methods is caused by the extra overhead involved in maintaining and updating the data structures in the "full-simplex" method. Indeed, the label-correcting methods have more pivots but each pivot is cheap to accomplish, while the primal network simplex methods have fewer pivots but each pivot involves more operations.

Goldfarb et al. [145] propose an $O(n^3)$ primal network simplex algorithm. The current fastest network simplex based SSSP algorithm is by Goldfarb and Jin [148] which has time bound $O(mn)$ and is as fast as the best label-correcting algorithms.

Several dual ascent algorithms [261] such as auction algorithms [47, 45, 71] and relaxation algorithms [45] have been proposed. The Dijkstra's algorithm can also be viewed as a primal-dual algorithm (See Section 5.4 and Section 6.4 of [263]). We will discuss in more detail the connection between three algorithms, Dijkstra's algorithm, primal-dual algorithm, and a new method called least squares primal-dual method (LSPD), in Section 3.6 later on.

In practice, the LP-based algorithms tend to perform worse than the combinatorial algorithms for solving SSSP [94, 58, 212].

3.3.3 New SSSP algorithm

Recently, Thorup [295, 296] proposed a deterministic linear time and space algorithm to solve the undirected SSSP for graphs with either integral or floating point nonnegative arc lengths on a random access machine (RAM). Based on Thorup's algorithm, Pettie and Ramachandran [265] use the minimum spanning tree structure to create a new algorithm on a pointer machine for the undirected SSSP with nonnegative floating point arc lengths. Their algorithm will have a better time bound of $O(m + n \log\log n)$ as long as the maximal ratio of any two arc lengths is small enough. These new algorithms, although shown to have good worst-case complexity, may be practically inefficient [14].

3.3.4 Computational experiments on SSSP algorithms

Extensive computational survey papers on SSSP algorithms have been written comparing the efficiency of the label-correcting, label-setting and hybrid methods on both artificially generated networks [260, 134, 135, 173, 96, 241, 74] and real road networks [312].

According to Cherkassky et al. [74], no single best algorithm exists for all classes of shortest path problems. They observed a double bucket implementation of Dijkstra's algorithm named DIKB and a label-correcting method named GOR1 which uses a *topological-scan* idea to be robust for most of their test cases. Zhan and Noon [312], using the same set of codes as Cherkassky et al., concluded that PAPE and TWO_Q, two variants of label-correcting methods, perform best on real road networks. They also recommend DIKA, a Dijkstra approximate buckets implementation, for cases with smaller arc lengths, and DIKB, which is more favorable for cases with larger arc lengths.

3.4 All pairs shortest path (APSP) algorithms

The All Pairs Shortest Path (APSP) problem determines the shortest paths between every pair of nodes. Obviously, it can be solved by n SSSP algorithms, one for each node as the root. Also, it can be solved by the LP reoptimization techniques which we will discuss in more detail in Section 3.5.2. In this section, we focus on algebraic algorithms based on the path algebra introduced in Section 3.2.

The optimality conditions of APSP are

$$x_{ij} = \begin{cases} \min\limits_{k \neq i,j}(c_{ik} + x_{kj}) & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases}$$

That is, suppose we know x_{kj}, the shortest distance from all other nodes k to j. Then, the shortest distance from i to j, x_{ij}, can be computed by choosing the shortest path among all possible paths from i to j via any node k. In fact, this is the well-known *Bellman's equation*. Using path algebra, Bellman's equation for solving APSP can be rewritten as

$$X = CX \oplus I_n \tag{3.6}$$

where I_n denotes an $n \times n$ identity matrix with 0 as the diagonal entries and ∞ as the off-diagonal entries.

It can be shown (see [65]) that (3.6) has a unique solution X^* if G has no cycles of zero length. An *extremal solution* X^* for (3.6) can be obtained recursively as

$$\begin{aligned} X^* &= I_n \oplus CX \\ &= I_n \oplus C(I_n \oplus CX) = I_n \oplus C \oplus C^2(I_n \oplus CX) \\ &= I_n \oplus C \oplus C^2 \oplus .... \oplus C^{n-1} \oplus C^n \oplus ... \\ &= I_n \oplus (I_n \oplus C) \oplus (I_n \oplus C^2) \oplus .... \oplus (I_n \oplus C^{n-1}) \oplus (I_n \oplus C^n) \oplus ... \\ &= I_n \oplus (I_n \oplus C) \oplus (I_n \oplus C^2) \oplus .... \oplus (I_n \oplus C^{n-1}) \\ &= I_n \oplus C \oplus C^2 \oplus .... \oplus C^{n-1} \\ &= (I_n \oplus C)^{n-1} \end{aligned} \tag{3.7}$$

In particular, the (i, j) entry of C^k, represents the shortest distance from i to j using paths containing at most k arcs. That is, the shortest distance from node i to node j can be obtained by the length of a shortest directed path from i to j using at most $n - 1$ arcs assuming no cycles with negative length [291].

Since we are only interested in the shortest path (not cycle) lengths, the generalized addition $(I_n \oplus C^k)$ will zero all the diagonal entries of C^k while retaining all the off-diagonal ones. The properties $(I_n \oplus C^k) = (I_n \oplus C^{n-1})$ for $k \geq n$ and $A \oplus A = A$ are used in the

above derivation. They hold because we assume there are no negative cycles, and any path in G contains at most $n-1$ arcs.

Now we will review APSP algorithms that solve Bellman's equations (3.6), and then summarize APSP methods that involve matrix multiplications.

3.4.1 Methods for solving Bellman's equations

The APSP problem can be interpreted as determining the shortest distance matrix X that satisfies Bellman's equations (3.6). Techniques analogous to the direct or iterative methods of solving systems of linear equations thus could be used to solve the APSP problem. In particular, direct methods such as the *Gauss-Jordan* and *Gaussian elimination* correspond to the well-known *Floyd-Warshall* [113, 304] and *Carré's* [64, 65] algorithms, respectively. Iterative methods like the *Jacobi* and *Gauss-Seidel* methods actually correspond to the SSSP algorithms by Bellman [40] and Ford [117], respectively (see [65] for proofs of their equivalence). The relaxation method of Bertsekas [45] can also be interpreted as a Gauss-Seidel technique (see [261]).

Since the same problem can also be viewed as inverting the matrix $(I_n - C)$, the *escalator method* [244] for inverting a matrix corresponds to an inductive APSP algorithm proposed by Dantzig [87]. Finally, the *decomposition algorithm* proposed by Mill [240] (also, Hu [172]) decomposes a huge graph into parts, solves APSP for each part separately, and then reunites the parts. This resembles the *nested dissection method* (see Chapter 8 in [98]), a partitioning or tearing technique to determine a good elimination ordering for maintaining sparsity, when solving a huge system of linear equations. All of these methods (except the iterative methods) have $O(n^3)$ time bounds and are believed to be efficient for dense graphs.

Here, we review the algorithm proposed by Carré [64], which corresponds to Gaussian elimination. This algorithm is seldom referred to in the literature. We are more interested in this algorithm because it inspires us to develop our new MPSP algorithms in Chapter 4.

3.4.1.1 Carré's algorithm

The Gaussian elimination-like APSP algorithm proposed by Carré contains three phases: one LU decomposition procedure and n successive forward/backward operations. Suppose

we first solve an ALL-1 shortest path tree rooted at a sink node t, and then repeat the same procedures for different sink nodes to solve the APSP problem. The reason for such unconventional notation (solving ALL-1 instead of 1-ALL shortest path problems) is that we usually solve for a column vector (corresponding to an ALL-1 distance vector) when we solve a system of linear equations. Here the t^{th}column in the distance matrix represents an ALL-1 shortest path tree rooted at a sink node t.

Algorithm 1 Carré

begin
  Initialize: $\forall$ s, $x_{ss} := 0$ and $succ_{ss} := s$;
    $\forall (s,t)$, ***if*** $(s,t) \in A$ ***then***
      $x_{st} := c_{st}$; $succ_{st} := t$;
    else $x_{st} := \infty$; $succ_{st} := 0$;
  LU;
  for each sink node t ***do***
    $Forward(t)$;
    $Backward(t)$;
end

There are three procedures: LU, $Forward(t)$, and $Backward(t)$.

Procedure LU
begin
  for $k = 1$ to $n-2$ ***do***
    for $s = k+1$ to n ***do***
      for $t = k+1$ to n ***do***
        if $s \neq t$ and $x_{st} > x_{sk} + x_{kt}$ ***then***
          $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

Procedure LU **:** The LU procedure is analogous to the LU decomposition in Gaussian elimination. That is, using node k, whose index is smaller than both s and t, as an intermediate node, if the current path from s to t is longer than the joint paths from s to k and k to t, we update x_{st} and its associated successor index. In particular, a shorter path from s to t is obtained by joining the two paths from s to k and from k to t.

In terms of algebraic operations, let l_{st} and u_{st} denote the (s,t) entry in the lower and upper triangle of X, respectively. The LU procedure is simply the variant of LU decomposition in Gaussian elimination. In particular, first we initialize them as $l_{st} := c_{st}$ $\forall$

$s > t$, $u_{st} := c_{st}$ $\forall$ $s < t$. Then the procedures compute

$$\begin{aligned} l_{st} &:= l_{st} \oplus \sum_{k=1}^{t-1} (l_{sk} \otimes u_{kt}) \\ u_{ts} &:= u_{ts} \oplus \sum_{k=1}^{t-1} (l_{tk} \otimes u_{ks}) \end{aligned} \quad \text{for } t = 2, \ldots, (n-1) \text{ and } s = (t+1), \ldots, n$$

The complexity of LU is $O(n^3)$. For a complete graph with n nodes, this procedure will perform exactly $\sum_{k=1}^{n-2} \sum_{s=k+1}^{n} \sum_{\substack{t=k+1 \\ j \neq i}}^{n} (1) = \frac{n(n-1)(n-2)}{3}$ triple comparisons.

Procedure Forward(t)
begin
  for $s = t + 2$ to n ***do***
    for $k = t + 1$ to $s - 1$ ***do***
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

Procedure $Forward(t)$ **:** For any sink node t, $Forward(t)$ is a procedure to obtain shortest paths from higher nodes to node t on G'_L, the induced subgraph representing the lower triangular part of X obtained after LU.

In terms of algebraic operations, this procedure is the same as solving $L \otimes y = b$, where the right hand side b is the t^{th} column of the identity matrix I_n. First we initialize $y_s := b_s$ $\forall$ s. Then

$$y_s := b_s \oplus \sum_{k=t}^{s-1} (l_{sk} \otimes y_k), \text{ for } s = (t+1), \ldots, n$$

The complexity of $Forward(t)$ is $O(n^2)$. For a complete graph with n nodes, this procedure will perform exactly $\sum_{t=1}^{n-2} \sum_{s=t+2}^{n} \sum_{k=t+1}^{s-1} (1) = \frac{n(n-1)(n-2)}{6}$ triple comparisons in solving APSP.

Procedure Backward(t)
begin
  for $s = n - 1$ down to 1 ***do***
    for $k = s + 1$ to n ***do***
      if $x_{st} > x_{sk} + x_{kt}$ ***then***
        $x_{st} := x_{sk} + x_{kt}$; $succ_{st} := succ_{sk}$;
end

Procedure *Backward*(t) **:** For any sink node t, *Backward*(t) is a procedure that computes shortest paths from all nodes to node t on G'_U, the induced subgraph representing the upper triangular part of X obtained after LU.

In terms of algebraic operations, this procedure is the same as solving $U \otimes x = y$, where the right hand side y is the solution to the previous operation, $L \otimes y = b$. First we initialize $x_s := y_s \ \forall \ s$. Then

$$x_{n-s} := y_{n-s} \oplus \sum_{k=n-s+1}^{n} (u_{n-s,k} \otimes x_k), \text{ for } s = 1, \ldots, (n-1)$$

The complexity of *Backward*(t) is $O(n^2)$. For a complete graph with n nodes, this procedure will perform exactly $\sum_{t=1}^{n} \sum_{\substack{s=1 \\ s \neq t}}^{n-1} \sum_{\substack{k=s+1 \\ k \neq t}}^{n} (1) = \frac{n(n-1)(n-2)}{2}$ triple comparisons in solving APSP.

More about Carré's algorithm : Table 6 is an example that illustrates how Carré's algorithm performs the triple comparisons $(s \rightarrow k \rightarrow t)$ to solve an APSP problem for a 4-node complete graph.

Table 6: Triple comparisons of Carré's algorithm on a 4-node complete graph

LU	
$s \rightarrow k \rightarrow t,\ k < s, t$	
$k = 1$	$k = 2$
2→1→3	
2→1→4	
3→1→2	
3→1→4	3→2→4
4→1→2	
4→1→3	4→2→3

Forward(t)			
$s \rightarrow k \rightarrow t,\ s > k > t$			
$t = 1$	$t = 2$	$t = 3$	$t = 4$
3→2→1	4→3→2		
4→3→1			
4→2→1			
Backward(t)			
$s \rightarrow k \rightarrow t,\ k > s$			
$t = 1$	$t = 2$	$t = 3$	$t = 4$
3→4→1	3→4→2	2→4→3	2→3→4
2→4→1	1→3→2	1→2→3	1→2→4
2→3→1	1→4→2	1→4→3	1→3→4

Carré gave an algebraic proof of the correctness and convergence of his algorithm. He also gave a graphical interpretation of his algorithm in [64].

Carré's algorithm has several good properties compared with other algebraic APSP algorithms. First, it performs the least number of triple comparisons, $n(n-1)(n-2)$, for complete graphs as shown by Nakamori [251]. The Floyd-Warshall algorithm performs the

same number of triple comparisons but uses a different ordering. Second, it decomposes the APSP into n shortest path trees, one for each sink node. Therefore, it can save many operations depending on the number of sink nodes in the MPSP problem. The Floyd-Warshall algorithm, on the other hand, has to do triple comparisons over the entire distance matrix.

3.4.1.2 Direct methods vs. iterative methods

Sparsity and stability are the two major concerns in solving linear systems. It is a well-known fact in numerical algebra that iterative methods usually converge faster but are more numerically unstable (i.e. affected by rounding errors) than direct methods. However, the operators in path algebra only involve comparison and addition; thus the numerical instability is not a concern in solving Bellman's equations in path algebra. This may explain why iterative methods such as label-correcting methods, instead of direct methods such as Floyd-Warshall or Carré's algorithms, are still the most popular methods used to solve either SSSP or APSP problems.

In this thesis, we investigate ways of improving the direct methods to exploit sparsity and make them competitive with the popular label-correcting methods.

3.4.2 Methods of matrix multiplication

As derived in equation (3.7), Shimbel [291] suggests a naive algorithm using $\log(n)$ matrix squarings of $(I_n \oplus C)$ to solve the APSP problem. To avoid many distance matrix squarings, some $O(n^3)$ distance matrix multiplication methods such as the *revised matrix* [171, 306] and *cascade* [108, 211, 306] algorithms perform only two or three successive distance matrix squarings. However, Farbey et al. [108] show that these methods are still inferior to the Floyd-Warshall algorithm which only needs a single distance matrix squaring procedure.

Aho et al. (see [2], pp.202-206) show that computing $(I_n \oplus C)^{n-1}$ is as hard as a single distance matrix squaring, which takes $O(n^3)$ time. Fredman [122] proposes an $O(n^{2.5})$ algorithm to compute a single distance matrix squaring, but it requires a program of exponential size. Its practical implementation, improved by Takaoka [293], still takes $O(n^3((\log\log n)/\log n)^{\frac{1}{2}})$ which is just slightly better. Recently, much work has been

done in using block decomposition and fast matrix multiplication techniques to solve the APSP problem. These new methods, although they have better subcubic time bounds, usually require the arc lengths to be either integers of small absolute value [12, 294, 313] or can only be applied to unweighted, undirected graphs [284, 126, 125]. All of these matrix multiplication algorithms seem to be more suitable for dense graphs since they do not exploit sparsity. Their practical efficiency remains to be evaluated.

3.4.3 Computational experiments on APSP algorithms

To date, no thorough computational analysis on solving the APSP problem has been reported. It is generally believed, however, that the algebraic algorithms require more computational storage and are more suitable for dense graphs, but are unattractive for graphs with large size or sparse structure (as is the case in most real world applications).

None of those matrix-multiplication algorithms has been implemented, despite their better complexity. The Floyd-Warshall algorithm is the most common algebraic APSP algorithm, but it can not take advantage of sparse graphs. On the other hand, Gaussian elimination is shown to be more advantageous than Gauss-Jordan elimination in dealing with sparse matrix [22]. This suggests that a sparse implementation of Carré's algorithm, rather than a sparse Floyd-Warshall algorithm, may be practically attractive.

Goto et al. [150] implement Carré's algorithm [65] sparsely using code generation techniques. The problem they faced is similar to ours. In particular, for a graph with fixed topology, shortest paths between all pairs must be repeatedly computed with different numerical values of arc lengths. To take advantage of the fixed sparse structure, they used a preprocessing procedure which first identifies a good node pivoting order so that the fill-ins in the LU decomposition phase are decreased. They then run Carré's algorithm once to record all the nontrivial triple comparisons in the LU decomposition, forward elimination and backward substitution phases. Based on the triple comparisons recorded in the preprocessing procedure, they generate an ad hoc shortest path code specifically for the original problem. Excluding the time spent in the preprocessing and code-compiling phases, this ad hoc code seems to perform very well, up to several times faster than other SSSP algorithms

they tested, on the randomly generated grid graphs.

In fact, the attractive performance of Goto's implementation may be misleading. First, the largest graph they tested, a 100-node 180-arc grid graph, is not as large as many real-world problems. For larger graphs, the code generated may be too large to be stored or compiled. Second, their experiments were conducted in 1976, since which time many good SSSP algorithms and efficient implementations have been proposed. However, their sparse implementations intrigued us and convinced us to further investigate ways of improving algebraic shortest path algorithms for solving shortest paths between multiple node pairs.

3.5 Multiple pairs shortest path algorithms

The ODMCNF usually contains multiple OD pairs. Thus we will focus on solving the *Multiple Pairs Shortest Path* (MPSP) problem, which is to compute the shortest paths for q specific OD pairs (s_i, t_i), $i = 1 \dots q$. Let $|s|$ ($|t|$) denote the number of distinct source (sink) nodes.

Obviously the MPSP problem can be solved by simply applying an SSSP algorithm $\hat{q}$ times, where $\hat{q} = \min\{|s|, |t|\}$ (we call such methods repeated SSSP algorithms), or by applying an APSP algorithm once and extracting the desired OD entries. Both of these methods can involve more computation than necessary. To cite an extreme example, suppose that we want to obtain shortest paths for n OD pairs, (s_i, t_i), $i = 1 \dots n$, which correspond to a matching. That is, each node appears exactly once in the source and sink node set but not the same time. For this specific example, we must apply an SSSP algorithm exactly n times, which is as hard as solving an APSP problem. Such "overkill" operations may be avoided if we use label-setting methods, since it suffices to terminate once all the destination nodes are permanently labeled. To do this, extra storage and book-keeping procedures are required. On the other hand, using the Floyd-Warshall algorithm, we still need to run until the last iteration to get all n OD entries. Either way we waste some time finding the shortest paths of many unwanted OD pairs.

In this section, we first review related methods appearing in the literature, and then briefly introduce our methods.

3.5.1 Repeated SSSP algorithms

Due to the simplicity and efficiency of the state-of-the-art SSSP algorithms, the MPSP problem is usually solved by repeatedly applying an SSSP algorithm $\hat{q}$ times, once for each source (or sink) node.

Theoretically, label-setting algorithms might be preferred due to their better time bounds. For cases with general arc lengths, we can (see [252, 183, 127]) first use a label-correcting algorithm to obtain a shortest path tree rooted at same node r, and then for each arc (i, j) we transform the original arc length c_{ij} to $c_{ij}^{\pi} = c_{ij} + d_i - d_j$, where d_i denotes the shortest distance from r to i. The nonnegative transformed arc length c_{ij}^{π} corresponds to the *reduced cost* in the LP. We then are able to repeatedly use the label-setting algorithm for the remaining $(\hat{q} - 1)$ SSSP problems.

Although this method has better theoretical time bounds, as argued in section 3.3.1, there is no absolute empirical superiority between label-setting and label-correcting methods. Therefore, whether it pays to do the extra arc length transformation, or simply apply a label-correcting method $\hat{q}$ times, is still quite debatable.

3.5.2 Reoptimization algorithms

The concept of reoptimization can be useful in two aspects: (1) when a MPSP problem must be solved with different arc lengths, and (2) when a shortest path tree must be calculated based on a previous shortest path tree.

After we have obtained a shortest path tree, suppose there are k arcs whose lengths have been changed. For cases involving only a decreasing-length-modification of some arcs, Goto and Sangiovanni-Vincentelli [151] proposed an $O(kn^2)$ algebraic algorithm that resembles the Householder formula for inverting modified matrices. Fujishige [123] gave another algorithm based on Dijkstra's algorithm which requires less initial storage than Goto's, but can not deal with cases that have negative arc lengths. For more general cases in which some arc lengths might increase, only LP algorithms such as primal network simplex methods have been proposed to attack such problems.

LP reoptimization techniques may also be advantageous for solving the MPSP problem.

For example, suppose we have obtained a shortest path tree rooted at r, and now try to obtain the next shortest path tree rooted at s. Dual feasibility will be maintained when switching roots, since the optimality conditions guarantee that the reduced cost, c_{ij}^{π}, remains nonnegative for each arc (i, j). We may thus use the current shortest path tree as an advanced basis to start with, and apply any dual method (e.g. the dual simplex method by Florian et al.[112], or the dual ascent method by Nguyen et al. [253]) to solve the newly rooted SSSP problem. The same problem may be solved by adding an artificial arc with long length from s to r while deleting the previous tree arc pointing toward s. Then, a primal network simplex method can be applied since primal feasibility is maintained.

These reoptimization methods exploit the advanced initial basis obtained from the previous iteration. Also, if the new root s is close to the old root r, many of the previous shortest path tree arcs might remain in the newly rooted shortest path tree.

Florian et al. [112] reported up to a 50% running time reduction in several test problems when compared with Dial's and Pape's algorithms (implemented in [94]). More recently, Burton [58] reported that Florian's algorithm runs faster than Johnson's algorithm (a repeated SSSP algorithm, see [183]) *only* when s is reachable by at most two arcs from r. In all other cases Johnson's algorithm outperforms Florian's.

More computational analysis of the MPSP problem is still required to draw a conclusion on the empirical dominance between these reoptimization algorithms and the repeated state-of-the-art SSSP algorithms.

3.5.3 Proposed new MPSP algorithms

It is easy to see that repeated SSSP algorithms are more efficient for MPSP problems with few sources (i.e. $\hat{q} \ll n$). For cases with many sources, are the repeated SSSP algorithms still superior in general? Can we modify an APSP algorithm so that it runs faster on MPSP problems, is competitive in solving MPSP problems on sparse graphs, and takes advantage of special properties of MPSP problems such as fixed topology or requested OD pairs? Chapter 4 and Chapter 5 try to answer these questions.

We propose two new algebraic shortest path algorithms based on Carré's algorithm

[64, 65] in Chapter 4. Like other algebraic APSP algorithms, our algorithms can deal with negative arc lengths. Even better, our algorithms can save half of the storage and running time for undirected or acyclic graphs, and avoid unnecessary operations that other algebraic APSP algorithms must do when solving MPSP problems.

To verify their empirical performance, in Chapter 5, we compare our codes with the state-of-the-art SSSP codes written by Cherkassky et al. [74] on artificial networks from four random network generators.

3.6 On least squares primal-dual shortest path algorithms

The *least squares primal-dual* (LSPD) algorithm [28] is a primal-dual algorithm for solving LPs. Instead of minimizing the sum of the absolute infeasibility to the constraints when solving the restricted primal problem (RPP) as does the conventional primal-dual algorithm, LSPD tries to minimize the sum of the squares of the infeasibility.

In particular, Leichner et al. [216] propose a LP phase I algorithm that strictly improves the infeasibility in each iteration in order to solve the following feasibility problem:

$$Ex = b,\ x \geq 0.$$

In stead of using the conventional simplex phase I method that solves

$$\begin{aligned} &\min |b - Ex| \\ &s.t.\ \ x \geq 0, \end{aligned}$$

they seek solutions to a special case of the *bounded least-squares problem* (BLS) called the *non-negative least-squares problem* (NNLS) formulated as follows:

$$\begin{aligned} &\min \|b - Ex\|^2 \\ &s.t.\ \ x \geq 0. \end{aligned} \tag{NNLS}$$

Similar algorithms have been proposed by Dantzig [86], Björck [54, 55], Van de Panne and Whinston [301], Lawson and Hanson [214], and Goldfarb and Idanani [147]. Recently, Gopalakrishnan et al. [149, 28, 30, 29] give variants of LSPD algorithms to solve several

classes of LP and network problems. Since the algorithm is impervious to degeneracy, it is considered to be efficient in solving highly degenerate problems such as assignment problems. Their computational experiments show that LSPD terminates in much fewer iterations than CPLEX network simplex and Hungarian method in solving the assignment problems.

The bottleneck of LSPD lies in solving NNLS efficiently, which in turn depends on solving a *least-squares problem* (LS) efficiently. When LSPD is applied to solve SCNF problems [149, 30], special properties of the node-arc incidence matrix can be exploited to derive a special combinatorial implementation which solves LS very efficiently. In this section we will give a specialized LSPD algorithm to solve the ALL-1 and 1-1 shortest paths on networks with nonnegative arc lengths, and discuss its connection to the original primal-dual algorithm and the well-known Dijkstra's algorithm.

3.6.1 LSPD algorithm for the ALL-1 shortest path problem

Given an initial feasible dual solution π (we can use $\pi = 0$ because $c_{ij} \geq 0$) for the dual problem ALL-1-Dual in Section 3.3.2, first we identify those arcs $(i, j) \in A$ satisfying $\pi_i - \pi_j = c_{ij}$, which we call *admissible arcs*. Let $\hat{A}$ be the set that contains all the admissible arcs. Let $\hat{G} = (N, \hat{A})$ denote the *admissible graph*, which contains all the nodes in N but only arcs in $\hat{A}$. By defining $c^{\pi}_{ij} = c_{ij} - \pi_i + \pi_j$ as the *reduced cost* for arc (i, j), the admissible arc set $\hat{A}$ contains all the arcs $(i, j) \in A$ such that $c^{\pi}_{ij} = 0$. We call a node i an *admissible node* if there exists a path from i to t in $\hat{G}$. Assuming the sink t is initially admissible, the admissible node set $\hat{N}$ is the connected component of $\hat{G}$ that contains t.

We can solve the restricted primal problem (RPP), which seeks the flow assignment x^* on admissible graph $\hat{G}$ that minimizes the sum of squares of the node imbalance (or slackness vector) $\delta = b - \hat{E}x_{\hat{A}}$:

$$\min \sum_{i \in N \setminus t} \delta_i^2 = \sum_{i \in N \setminus t} (b_i - \hat{E}_{i.} x_{\hat{A}})^2 \tag{3.8}$$

$$s.t. \;\; x_a \geq 0 \;\forall\; a \in \hat{A},$$

where $\hat{E}$ is the column subset of the node-arc incidence matrix (with the row corresponding

to node t removed) that corresponds to the admissible arcs $\hat{A}$. All the non-admissible arcs have zero flows.

Problem (3.8) is a NNLS problem and can be solved by the algorithm of Leichner et al. [216]. The optimal imbalance δ^* can be used as a dual improving direction to improve π (see Gopalakrishnan et al. [149, 28] for the proof) in the LSPD algorithm. Here we give a special implementation (Algorithm 2) of Gopalakrishnan et al.'s algorithm, *LSPD-ALL-1*, for solving the ALL-1 shortest path problem. It contains a procedure *NNLS-ALL-1* to solve the RPP (3.8).

Algorithm 2 LSPD-ALL-1

begin

Initialize: $\forall$ node i, $\pi_i := 0$; $\delta_t^* := 0$; add node t to $\hat{N}$;

Identify admissible arc set $\hat{A}$ and admissible node set $\hat{N}$;

while $\left|\hat{N}\right| < n$ ***do***

$\delta^* = NNLS\text{-}ALL\text{-}1(\hat{G}, \hat{N})$;

$\theta = \min_{(i,j)\in A, \delta_i^* > \delta_j^*} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta_i^* - \delta_j^*}\}$; $\pi = \pi + \theta\delta^*$;

Identify admissible arc set $\hat{A}$ and admissible node set $\hat{N}$;

end

Procedure LSPD-ALL-1$(\hat{\mathbf{G}}, \hat{\mathbf{N}})$

begin

for $i = 1$ to n ***do***

if node $i \in \hat{N}$ ***then***

$\delta_i^* = 0$;

else

$\delta_i^* = 1$;

return δ^*;

end

Applying the algorithm *LSPD-ALL-1*, we obtain the following observations:

1. $\delta_i^* = 0$, $\forall i \in \hat{N}$ and $\delta_i^* = 1$, $\forall i \in N \setminus \hat{N}$.

2. Let $\hat{N}^k$ denote the admissible nodes set obtained in the beginning of iteration k, then $\hat{N}^k \subseteq \hat{N}^{k+1}$ and $\left|\hat{N}^{k+1}\right| \geq \left|\hat{N}^k\right| + 1$

3. In at most $n-1$ major iterations, the algorithm *LSPD-ALL-1* terminates with $\hat{N} = N$.

Now we show that algorithm *LSPD-ALL-1* will correctly compute an ALL-1 shortest tree.

Theorem 3.1. *The δ^* computed by the procedure* NNLS-ALL-1 *solves problem (3.8).*

Proof. Because no non-admissible node has a path of admissible arcs to t, no non-admissible node can ship any of its imbalance (initialized as $\delta_i^* = 1 \; \forall i \in N \setminus t$) to t via admissible arcs, and thus its optimal imbalance remains 1. On the other hand, each admissible node can always ship its imbalance to t via uncapacitated admissible arcs so that its optimal imbalance becomes zero. Therefore the δ^* computed by procedure *NNLS-ALL-1* corresponds to the optimal imbalance δ_i^* for the RPP (3.8). □

Lemma 3.1. *Algorithm* LSPD-ALL-1 *solves the ALL-1 shortest path problem.*

Proof. Algorithm *LSPD-ALL-1* is a specialized LSPD algorithm for ALL-1 shortest path problem. By Theorem 3.1, the δ^* solves quadratic RPP (3.8). δ^* is a dual ascent direction [149, 28]. The algorithm *LSPD-ALL-1* iteratively computes the step length θ to update dual variables π, identifies admissible arcs (i.e., columns), and solves the quadratic RPP (3.8) until $\sum_{i \in N \setminus t} \delta_i^{*2}$ vanishes, which means the primal feasibility is attained. Since the dual feasibility and complementary slackness conditions are maintained during the whole procedure, *LSPD-ALL-1* solves the ALL-1 shortest path problem. □

Now we compare algorithm *LSPD-ALL-1* with the original primal-dual algorithm for solving the ALL-1 shortest path problem.

3.6.2 LSPD vs. original PD algorithm for the ALL-1 shortest path problem

The only difference between algorithm LSPD and the original PD algorithm is that they solve different RPP. The original PD algorithm solves the following RPP:

$$\min \sum_{i \in N \setminus t} \delta_i \qquad \text{(RPP-ALL-1)}$$

$$s.t. \;\; \hat{E}_{i\cdot} x_{\hat{A}} + \delta_i = 1, \; i \in N \setminus t$$

$$x_{\hat{A}}, s \geq 0$$

whose dual is

$$\max \sum_{i \in N \setminus t} \rho_i \qquad \text{(DRPP-ALL-1)}$$

$$s.t. \;\; \rho_i \leq \rho_j, \, \forall (i,j) \in \hat{A}, i, j \neq t$$

$$\rho_i \leq 0, \, \forall (i,t) \in \hat{A}$$

$$\rho_j \geq 0, \, \forall (t,j) \in \hat{A}$$

$$\rho_i \leq 1, \, \forall i \in N \setminus t$$

The optimal dual solution ρ^* of DRPP-ALL-1 will be used as a dual-ascent direction in the PD process. It is easy to observe that $\rho_i^* = 1$ for each node i that can not reach t along admissible arcs in $\hat{A}$ (i.e. i is non-admissible). Also, if node i is admissible, that is, there exists a path from i to t with intermediate nodes $\{i_1, i_2, i_3, ..., i_k\}$, then $\rho_{i_1}^* = \rho_{i_2}^* = \rho_{i_3}^* = ... = \rho_{i_k}^* = 0$. In other words, the original PD algorithm will have $\rho^* = 0$ for all the admissible node, and $\rho^* = 1$ for all the non-admissible nodes. Thus the improving direction ρ^* obtained by the original PD algorithm is identical to the one obtained by LSPD introduced in previous section.

Therefore we can say that algorithm LSPD and the original PD algorithm are identical to each other in solving the ALL-1 shortest path problem since they produce the same improving direction and step length and construct the same restricted network $\hat{G}$.

Next we will compare these two algorithms with the famous Dijkstra's algorithm.

3.6.3 LSPD vs. Dijkstra's algorithm for the ALL-1 shortest path problem

First we review the Dijkstra's algorithm. For our convenience, we construct a new graph $G'' = (N, A'')$ by reversing all the arc direction of A so that the original ALL-1 shortest path problem on G to sink t becomes a 1-ALL shortest problem from source t with nonnegative arc length on G''. Initialize a node set V as empty and its complement $\overline{V}$ as the whole node set N. The distance label for each node i, denoted as $d(i)$, represents the distance from t to i in G''. Define $pred(j) = i$ if node i is the predecessor of node j.

We say a node is *permanently labeled* if it is put into V. A node is *labeled* if its distance label is finite. A node is *temporarily labeled* if it is labeled but not permanently labeled.

Algorithm 3 Dijkstra(G'')

begin

Initialize: $\forall$ node $i \in N \setminus t$, $d(i) := \infty$, $pred(i) = -1$;
$d(t) := 0$, $pred(t) := 0$; $V := \emptyset, \overline{V} := N$;

while $|V| < n$ ***do***

let $i \in \overline{V}$ be a node such that $d(i) = \min\{d(j) : j \in \overline{V}\}$
$V := V \cup \{i\}; \overline{V} := \overline{V} \setminus \{i\}$
for each $(i, j) \in A$ ***do***
if $d(j) > d(i) + c_{ij}$ **then**
$d(j) := d(i) + c_{ij}$; $pred(j) := i$;

end

Dijkstra's algorithm starts by labeling t, and then iteratively labels temporary nodes with arcs from permanently labeled nodes. This is identical to the *LSPD-ALL-1* which grows admissible nodes only from admissible nodes. In fact, in every major iteration, the set of admissible nodes in *LSPD-ALL-1* is the same as the set of permanently labeled nodes in Dijkstra. To show this, we only need to show that both algorithms will choose the same nodes in every major iteration.

Theorem 3.2. *Both algorithm Dijkstra and* LSPD-ALL-1 *choose the same node to become permanently labeled (in Dijkstra) or admissible (in LSPD-ALL-1) in each major iteration.*

Proof. We already know that both algorithms start at the same node t. In *LSPD-ALL-1*, we will identify an admissible node j_1 by identifying the admissible arc (j_1, t) such that $(j_1, t) = \arg \min_{(j,t)\in A, s_j^* > s_t^*} \{\frac{c_{jt} - \pi_i + \pi_j}{s_j^* - s_t^*}\} = \arg \min_{(j,t)\in A} \{c_{jt}\}$. The second equality holds because $s_j^* = 1$, $s_t^* = 0$, and $\pi = 0$ in the first iteration. This is the same as Dijkstra's algorithm in the first iteration.

Assume both algorithms have the same set of admissible (or permanently labeled) nodes V^k in the beginning of the k^{th} major iteration. Algorithm *LSPD-ALL-1* will choose an admissible arc (i_k, j_r) such that $(i_k, j_r) = \arg \min_{(i,j)\in A, \delta_i^* > s_j^*} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta_i^* - s_j^*}\} = \arg \min_{(i,j)\in A, j\in V^k, i\notin V^k} \{c_{ij} + \pi_j\}$. Again, the second equality holds because $s_j^* = 0$ for each $j \in V^k$, and $\delta_i^* = 1$, $\pi_i = 0$ for each $i \notin V^k$. If node j is admissible in the k^{th} iteration, let $j \to j_p \to ... \to j_2 \to j_1 \to t$ denote the path from j to t. Then we can calculate $\pi_j = c_{jj_p} + ... + c_{j_2 j_1} + c_{j_1 t}$ since $\pi_t = 0$ and all the arcs along this path are admissible thus having zero reduced cost. So, $(i_k, j_r) =$

$\arg \min_{(i,j)\in A, j\in V^k, i\notin V^k} \{c_{ij} + \pi_j\} = \arg \min_{(i,j)\in A, j\in V^k, i\notin V^k} \{ \sum_{(p,q)\in path\{i\to j\to j_p\to ...\to j_1\to t\}} c_{pq}\}$. Therefore, in the beginning of the $(k+1)^{st}$ major iteration, node i_k becomes admissible with π_{i_k} $= \sum_{(p,q)\in path\{i_k\to j_r\to j_{r-1}\to ...\to j_1\to t\}} c_{pq}$.

Dijkstra's algorithm in the k^{th} iteration will choose the node reachable from V^k with the minimum distance label. That is, choose node i_k reachable from a permanent labeled node j_r such that $d(i_k) = \min_{(j,i)\in A, j\in V^k} d(i) = \min_{(j,i)\in A, j\in V^k} \{d(j) + c_{ji}\}$. Let $t \to j_1 \to j_2 \to ... \to j_p \to j$ denote the path from t to a permanently labeled node j. Since j is permanently labeled, $d(j) = \sum_{(p,q)\in path\{j\to j_p\to ...\to j_2\to j_1\to t\}} c_{pq}$. Therefore, node i_k will become permanently labeled in the $(k+1)^{st}$ major iteration with distance label $d(i_k) = \sum_{(p,q)\in path\{t\to j_1\to j_2\to ...\to j_r\to i_k\}} c_{pq}$.

Therefore, these two algorithms perform the same operation to get the same shortest distance label for the newly permanently labeled (or admissible) node. □

From these discussion, we conclude that when solving the ALL-1 shortest path problem with nonnegative arc length, all the three algorithms, Dijkstra, *LSPD-ALL-1*, and the original PD algorithm, will perform the same operations in each iteration. In fact, this result is due to the nondegeneracy of the problem structure. Remember that the basis corresponds to a spanning tree in the network problem. In this ALL-1 shortest path problem, each node other than t has supply 1 to send to t. In each iteration of these algorithms, the primal infeasibility will strictly decrease, hence each pivot is always nondegenerate.

LSPD is an algorithm designed to take advantage of doing nondegenerate pivots in each iteration. Therefore, in this special case, it just performs as efficiently as the other two algorithms. Next we will see that because the 1-1 shortest path problem does not have the nondegenerate property, thus *LSPD-1-1* does do a better job than the original PD algorithm in some sense.

3.6.4 LP formulation for the 1-1 shortest path problem

Unlike the ALL-1 shortest path problem which searches for a shortest path tree (SPT), the 1-1 shortest path problem only needs part of the SPT, namely, the shortest path between certain two nodes, s and t. It can be viewed as sending a unit flow from s to t with the minimal cost via uncapacitated arcs. Its linear programming formulation is similar to the

ALL-1 formulation in Section 3.3.2 except now the node imbalance vector b only has two nonzero entries: $+1$ for s, -1 for t, and 0 for all the other nodes.

Since the node-arc incidence matrix has one redundant row, we remove the row corresponding to t, to get the following primal and dual formulations:

$$\min cx = Z^{P*}_{1-1}(x) \tag{1-1-Primal}$$

$$s.t.\ \ \overline{N}x = \begin{cases} 1 & \text{, if } i = s \\ 0 & \text{, if } i \in N \setminus \{s,t\} \end{cases} \text{, } i \in N \setminus t \tag{3.9}$$

$$x \geq 0,$$

whose dual is

$$\max \pi_s = Z^{D*}_{1-1}(\pi) \tag{1-1-Dual}$$

$$s.t.\ \ \pi_i - \pi_j \leq c_{ij}\ \forall (i,j) \in A, i, j \neq t \tag{3.10}$$

$$\pi_i \leq c_{it}\ \forall (i,t) \in A \tag{3.11}$$

$$-\pi_j \leq c_{tj}\ \forall (t,j) \in A \tag{3.12}$$

Here the right-hand-side of (3.9) only has one nonzero entry ($+1$ for node s). This makes the dual objective $Z^{D*}_{1-1}(\pi)$ differ from that of ALL-1, $Z^{D*}_{ALL-1}(\pi)$, in which $Z^{D*}_{1-1}(\pi)$ maximizes only π_s while $Z^{D*}_{ALL-1}(\pi)$ maximize the sum $\sum_{i \in N \setminus t} \pi_i$. Therefore, we give a new procedure *NNLS-1-1* to solve the nonnegative least-squares problem in our 1-1 shortest path algorithm, *LSPD-1-1*. We will also illustrate the difference between solving the ALL-1 and 1-1 shortest path problem when the original PD algorithm is applied. Finally we will explain the connections between the Dijkstra, *LSPD-1-1*, and original PD algorithms when they are used to solve the 1-1 shortest path problem.

3.6.5 LSPD algorithm for the 1-1 shortest path problem

Before we give the new *LSPD-1-1* shortest path algorithm, the *admissible node* set $\hat{N}$ has to be redefined as follows: a node i is *admissible* if it is reachable from s only via admissible arcs. All the other definitions such as admissible arcs $\hat{A}$ and the admissible graph $\hat{G}$ remain the same as in Section 3.6.1. With a procedure *NNLS-1-1* that solves the RPP (3.8), the algorithm *LSPD-1-1* is shown below as Algorithm 4.

Algorithm 4 LSPD-1-1

begin

Initialize: $\forall$ node i, $\pi_i := 0$; $\delta_s^* := 0$; add node s to $\hat{N}$;

Identify admissible arc set $\hat{A}$ and admissible node set $\hat{N}$;

while node $t \notin \hat{N}$ ***do***

$\delta^* = NNLS\text{-}1\text{-}1(\hat{G}, \hat{N})$;

$\theta = \min_{(i,j)\in A, \delta_i^* > \delta_j^*} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta_i^* - \delta_j^*}\}$; $\pi = \pi + \theta\delta^*$;

Identify admissible arc set $\hat{A}$ and admissible node set $\hat{N}$;

end

Procedure LSPD-1-1$(\hat{\mathbf{G}}, \hat{\mathbf{N}})$

begin

for $i = 1$ to n ***do***

if node $i \in \hat{N}$ ***then***

$\delta_i^* = \frac{1}{|\hat{N}|}$;

else

$\delta_i^* = 0$;

return δ^*;

end

Applying algorithm *LSPD-1-1*, we obtain the following observations:

1. $\delta_i^* = \frac{1}{|\hat{N}|}$, $\forall i \in \hat{N}$ and $\delta_i^* = 0$, $\forall i \in N \setminus \hat{N}$.
2. Let $\hat{N}^k$ denote the admissible nodes set obtained in the beginning of iteration k. Then $\hat{N}^k \subseteq \hat{N}^{k+1}$ and $\left|\hat{N}^{k+1}\right| \geq \left|\hat{N}^k\right| + 1$.
3. In at most $n - 1$ major iterations, the algorithm *LSPD-1-1* terminates when node t becomes admissible. Then, s can send its unit imbalance to t via some path composed only by admissible arcs so that the total imbalance over all nodes becomes 0.

Now we show that algorithm *LSPD-1-1* will correctly the shortest path from s to t.

Theorem 3.3. *The δ^* computed by the procedure* NNLS-1-1 *solves problem (3.8).*

Proof. The RPP (3.8) is a quadratic programing problem. If we relax the nonnegativity constraints, it is a least-squares problem which can be solved by solving the normal equation $\hat{E}^T\hat{E}x^* = \hat{E}^Tb$. In other words, $\hat{E}^T\delta^* = \hat{E}^T(b - \hat{E}x^*) = 0$. Note that each row of $\hat{E}^T$ contains only two nonzero entries (i.e., $+1$ and -1) which represent an admissible arc. In

other words, $\hat{E}^T\delta^* = 0$ implies $\delta_i^* = s_j^*$ for each admissible arc (i,j) which implies all admissible nodes have the same optimal imbalance δ^*. Since the total system imbalance is 1 (from the source s), the optimal least-squares solution δ_i^* for the RPP (3.8) will be $\frac{1}{|\hat{N}|}$ for each admissible node i. Using the optimal imbalance δ^*, it is easy to compute the unique optimal arc flow x^* and verify that $x^* \geq 0$ by traversing nodes on the component that contains the source node s (For more details in application of LSPD on network problems, see [149, 30].). Thus the optimal imbalance δ^* by the procedure *NNLS-1-1* solves (3.8) □

Lemma 3.2. *Algorithm* LSPD-1-1 *solves the 1-1 shortest path problem from s to t.*

Proof. Algorithm *LSPD-1-1* is a specialized LSPD algorithm for 1-1 shortest path problem. By Theorem 3.3, the δ^* solves quadratic RPP (3.8). δ^* is a dual ascent direction [149, 28]. The algorithm *LSPD-1-1* iteratively computes the step length θ to update dual variables π, identifies admissible arcs (i.e., columns), and solves the quadratic RPP (3.8) until $\sum_{i \in N \setminus t} \delta_i^{*2}$ vanishes, which means the primal feasibility is attained. Since the dual feasibility and complementary slackness conditions are maintained during the whole procedure, *LSPD-1-1* solves the 1-1 shortest path problem from s to t. □

Intuitively, we can view this algorithm as the following: starting from the source s, *LSPD-1-1* tries to reach t by growing the set of admissible nodes. The algorithm keeps propagating the unit imbalance along all the admissible arcs so that the unit imbalance will be equally distributed to each admissible node before t becomes admissible. Once t becomes admissible, all the imbalance flows to t so that the optimal system imbalance δ^* becomes 0. Then the algorithm is finished.

To further speed up algorithm *LSPD-1-1*, we observe that for each admissible node k, $s_k^* = \frac{1}{|\hat{N}|}$ and $\theta s_k^* = \min_{(i,j) \in A, \delta_i^* > s_j^*} \{\frac{c_{ij} - \pi_i + \pi_j}{\delta_i^* - s_j^*}\} \cdot \frac{1}{|\hat{N}|} = \min_{(i,j) \in A, \delta_i^* = \frac{1}{|\hat{N}|}, s_j^* = 0} \{\frac{c_{ij} - \pi_i + \pi_j}{\frac{1}{|\hat{N}|}}\} \cdot \frac{1}{|\hat{N}|} = \min_{(i,j) \in A, \delta_i^* > s_j^*} \{c_{ij} - \pi_i + \pi_j\} \cdot 1$. Thus we can speed up the algorithm *LSPD-1-1* using $\hat{\theta} = \min_{(i,j) \in A, \delta_i^* > s_j^*} \{c_{ij} - \pi_i + \pi_j\}$ and $\hat{s}_k = 1$ instead of the original θ and s_k^*. This modification will not affect the selection of new admissible arc and node; thus the algorithm achieves the same objective using simpler computations.

For our convenience, we will use this modified version of the *LSPD-1-1* algorithm in later sections.

3.6.6 LSPD vs. original PD algorithm for the 1-1 shortest path problem

When the original PD algorithm solves the 1-1 shortest path problem, the primal RPP formulation is as follows:

$$\min \sum_{i \in N \setminus t} \delta_i \tag{RPP-1-1}$$

$$s.t.\ \ \hat{E}_i . x_{\hat{A}} + \delta_i = \begin{cases} 1 & \text{, if } i = s \\ 0 & \text{, if } i \in N \setminus \{s, t\} \end{cases} \text{, } i \in N \setminus t$$

$$x_{\hat{A}}, s \geq 0$$

whose dual is

$$\max \rho_s \tag{DRPP-1-1}$$

$$s.t.\ \ \rho_i \leq \rho_j,\ \forall (i,j) \in \hat{A}, i, j \neq t$$

$$\rho_i \leq 0,\ \forall (i,t) \in \hat{A}$$

$$\rho_j \geq 0,\ \forall (t,j) \in \hat{A}$$

$$\rho_i \leq 1,\ \forall i \in N \setminus t$$

Unlike when solving the ALL-1 shortest path problem, the original PD algorithm will have degenerate pivots when solving RPP-1-1, which is a major difference from the LSPD algorithm since the LSPD algorithm guarantees nondegenerate pivots at every iteration.

If s and t are not adjacent and all the arc costs are strictly positive, we start the algorithm with $\pi = 0$ which makes $\hat{A}$ empty in the first iteration. Then the optimal solution for DRPP-1-1 in the first iteration will be $\rho_s^* = 1$, $\rho_i^* \leq 1\ \forall i \in N \setminus \{s, t\}$. That is, we are free to choose any ρ_i^* as long as it does not exceed 1. This property of multiple optimal dual solutions is due to the degeneracy of RPP-1-1. When we have multiple choices to improve the dual solution, there is no guarantee of improving the objective of RPP-1-1 at any iteration. In fact, we may end up cycling or take a long time to move out the degenerate primal solution.

If we are very lucky, by choosing the "right" dual improving direction, we may even solve this problem much faster.

To eliminate the uncertainty caused by primal degeneracy when solving RPP-1-1, we have to choose the dual improving direction in a smart way. One way is to choose $\rho_i^* = 0$ for non-admissible nodes. Then, by the first constraint in DRPP-1-1, admissible nodes will be forced to have $\rho_i^* = 1$. This is because we want to maximize ρ_s, and the best we can do is $\rho_s^* = 1$. By doing so, we force all the nodes reachable from s (i.e., admissible nodes) to have $\rho_i^* = 1$. Then the original PD algorithm chooses the same admissible arcs and nodes as *LSPD-1-1*. Next, we will show that this specific PD algorithm performs the same operations as Dijkstra's algorithm in each iteration.

3.6.7 LSPD vs. Dijkstra's algorithm for the 1-1 shortest path problem

The Dijkstra's algorithm for the 1-1 shortest path problem is the same as the ALL-1 case in Section 3.6.3, except that it terminates as soon as the sink t is permanently labeled. In this section, we show that algorithm *LSPD-1-1* performs the same operations as Dijkstra's algorithm does.

Algorithm *LSPD-1-1* starts at source node s, and then identifies admissible arcs to grow the set of admissible nodes. This is the same as Dijkstra's algorithm. If both algorithms choose the same node in each iteration, the admissible node set $\hat{N}$ in the *LSPD-1-1* algorithm will be equivalent to the permanently labeled node set V in Dijkstra's algorithm.

The following proposition explains that both algorithms choose the same nodes in every major iteration.

Theorem 3.4. *Both Dijkstra and* LSPD-1-1 *choose the same node to become permanently labeled (in Dijkstra) or admissible (in* LSPD-1-1*) in each major iteration.*

Proof. We already know that both algorithms start at s. In *LSPD-1-1*, we will identify an admissible node i_1 by identifying the admissible arc (s, i_1) such that $(s, i_1) = \arg \min_{(s,i)\in A, s_s^* > \delta_i^*} \{\frac{c_{si} - \pi_s + \pi_i}{s_s^* - \delta_i^*}\} = \arg \min_{(s,i)\in A} \{c_{si}\}$.The second equality holds because $s_s^* = 1$, $\delta_i^* = 0$, and $\pi = 0$ in the first iteration. This is the same as Dijkstra's algorithm in the first iteration.

Assume both algorithms have the same set of admissible (or permanently labeled) nodes V^k in the beginning of the k^{th} major iteration. Algorithm *LSPD-1-1* will choose an arc (i_r, j_k) such that $(i_r, j_k) = \arg \min_{(i,j)\in A, \delta_i^* > s_j^*} \{\frac{c_{ij}-\pi_i+\pi_j}{\delta_i^*-s_j^*}\} = \arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{c_{ij} - \pi_i\}$. Again, the second equality holds because $\delta_i^* = 1$ for each $i \in V^k$, and $s_j^* = 0$, $\pi_j = 0$ for each $j \notin V^k$ If node i is admissible in the k^{th} iteration, let $s \to i_1 \to i_2 \to ... \to i_p \to i$ denote the path from s to i. Then we can calculate $\pi_s = c_{si_1} + c_{i_1i_2} + ... + c_{i_pi} + \pi_i$ since all the arcs along this path are admissible and thus have zero reduced cost. That is, $-\pi_i = \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i\}} c_{pq} - \pi_s$ for each admissible node i. So, (i_r, j_k) $= \arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{c_{ij} - \pi_i\} = \arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{ \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i\to j\}} c_{pq} - \pi_s\} =$ $\arg \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{ \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i\to j\}} c_{pq}\}$. The last equality holds since π_s is fixed when we compare all the arcs $(i, j) \in A, i \in V^k, j \notin V^k$.

Therefore, in the beginning of the $(k+1)^{st}$ major iteration, node j_k becomes admissible with $\pi_{j_k} = 0$, and $\pi_s = \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i_r\to j_k\}} c_{pq}$.

The criterion to choose the new admissible arc (i_r, j_k) is the same as Dijkstra's algorithm which we will explain next.

Dijkstra's algorithm in the k^{th} iteration will choose a node reachable from V^k with minimum distance label. That is, it chooses a node j_k reachable from some permanent node i_r such that $d(j_k) = \min_{(i,j)\in A, i\in V^k, j\notin V^k} d(j) = \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{d(i) + c_{ij}\}$. Let $s \to i_1 \to i_2 \to ... \to i_p \to i$ denote the path from s to a permanently labeled node i. Since i is permanently labeled, $d(i) = \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i_p\to i\}} c_{pq}$. Therefore, node j_k will become permanently labeled because $d(j_k) = \min_{(i,j)\in A, i\in V^k, j\notin V^k} \{d(i) + c_{ij}\} = \min_{(i,j)\in A, i\in V^k, j\notin V^k}$ $\{ \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i\to j\}} c_{pq}\}$. That is, node j_k will become permanently labeled in the $(k+1)^{st}$ major iteration and will have distance label $d(j_k) = \sum_{(p,q)\in path\{s\to i_1\to i_2...\to i_r\to j_k\}} c_{pq}$.

It is easy to see that these two algorithms perform the same operation to identify the same newly permanently labeled (or admissible) node. □

From this proposition, we observe that in the *LSPD-1-1* algorithm, π_i represents the shortest distance between an admissible node i and the most recent admissible node in V^k, while in Dijkstra's algorithm $d(i)$ represents the shortest distance between s and i. In other

words, $d(i) = \pi_s - \pi_i$ for any admissible node i. Therefore, when t becomes admissible, $d(t) = \pi_s$.

Here we use a physical example to illustrate these two algorithms. Consider each node as a ball, and each arc as a string connecting two balls. Here we assume all the arc length is positive.

For the *LSPD-1-1* algorithm, we can think as follows: in the beginning we put all the balls on the floor, and then we pick up ball s and raise it. We keep raising s until the string (s, i_1) becomes tight; then if we raise ball s further more, ball i_1 will be raised. We keep increasing the height of s until finally ball t is raised. At this moment, the height of s (i.e. π_s) is the shortest path between s and t, and the shortest path consists of all arcs on the path to t whose strings are tight.

For Dijkstra's algorithm, we use a plate which has one hole for these n balls to fall through. In the beginning we put the plate on the floor, then put ball s in the hole and all the other balls on the plate. We will put ball s on the floor so that when we raise the plate, s will stay on the floor. Then we begin to raise the plate until the string (s, i_1) is so tight that ball i_1 begins to pass through the hole. We keep raising the plate until finally ball t is about to fall through the hole. At that time, the height of ball t (i.e. $d(t)$) is the shortest path length between s and t, and the shortest path consists of all arcs on the path to t whose strings are tight.

From these discussion, we conclude that when solving the 1-1 shortest path problem with nonnegative arc lengths, Dijkstra and *LSPD-1-1* algorithm are, in fact, identical to each other. The original P-D algorithm will face the problem of primal degeneracy when solving the RPP-1-1. However, if we choose the improving dual direction intelligently (i.e., $\rho^* = 0$ for all non-admissible nodes and $\rho^* = 1$ for all admissible nodes), the original PD algorithm will perform same operations as the *LSPD-1-1* algorithm.

3.6.8 Summary

In summary, the LSPD algorithm is identical to Dijkstra's algorithm for solving both ALL-1 and 1-1 shortest path problems. The original PD algorithm, on the other hand, is identical

to the Dijkstra's algorithm for solving the ALL-1 shortest path problem, but needs to choose a specific dual improving direction (there are multiple ones) so that it will be identical to the Dijkstra's algorithm.

For the general arc cost cases, both LSPD and the original PD algorithms can be applied, but Dijkstra's algorithm is not applicable. The theoretical and practical performances of LSPD and the original PD algorithms for solving shortest path problems with general arc costs remain to be investigated.