http://pubsonline.informs.org/journal/ijoc

INFORMS JOURNAL ON COMPUTING
Vol. 32, No. 2, Spring 2020, pp. 279–288
ISSN 1091-9856 (print), ISSN 1526-5528 (online)

Solving the Nearly Symmetric All-Pairs Shortest-Path Problem

Gerald G. Brown,[a] **W. Matthew Carlyle**[a]

[a] Naval Postgraduate School, Monterey, California 93943

Contact: ggbrown@nps.edu, http://orcid.org/0000-0002-2974-7162 (GGB); mcarlyle@nps.edu, http://orcid.org/0000-0003-4806-6634 (WMC)

Received: May 12, 2017
Revised: June 12, 2018; October 4, 2018
Accepted: October 9, 2018
Published Online in Articles in Advance: November 1, 2019

https://doi.org/10.1287/ijoc.2018.0873

Abstract. We introduce a simple modification to the repeated shortest-path algorithm for the all-pairs shortest-path problem that adds a cumulative distance label update at each iteration based on the shortest-path tree from the prior iteration. We have implemented and tested our update using several shortest-path algorithms on a range of test networks of varying size, degree, and "skewness" (i.e., asymmetry) of costs on antisymmetric arcs, and we find that it provides a significant speedup to any such algorithm, except for cases either in which the underlying graph is extremely sparsely connected (or even disconnected) or when the arc costs are highly nonsymmetric. An added charm is that our best-modified method preserves the polynomial worst case runtime of its label-correcting antecedent. As with other repeated shortest-path algorithms, it is significantly faster than the Floyd–Warshall algorithm on sparsely connected networks and even some fairly densely connected networks.

History: Accepted by S. Raghavan, Area Editor for Network Optimization: Algorithms & Applications.
Funding: This work was funded by the Office of Naval Research.
Supplemental Material: The online appendix is available at https://doi.org/10.1287/ijoc.2018.0873.

Keywords: all-pairs shortest paths • repeated shortest-path algorithm • label-correcting algorithms • two-queue • symmetric costs

1. Introduction

Given a network specified by a directed graph $G = (N, A)$, where $n = |N|$ is the number of nodes and $m = |A|$ is the number of arcs, and an integral cost c_{ij} on each arc $(i, j) \in A$, the *all-pairs shortest-path problem* seeks a minimum length directed path between each pair of nodes i and j in N if such a path exists, where the length of a directed path is the sum of the costs of the arcs on the path. The all-pairs shortest-path problem can be solved using a specialized method, such as the Floyd–Warshall algorithm (Floyd 1962, Ingerman 1962, Warshall 1962), or using a *repeated shortest-path* algorithm, which solves a sequence of single-source shortest-path problems: one from each possible source node to all other nodes. The repeated shortest-path algorithm is usually recommended for sparse networks, whereas the concise exposition of the Floyd–Warshall algorithm is preferred for dense problems.

We introduce a distance label processing step for repeated shortest-path algorithms that improves their performance over a wide range of test problems (including many that would not traditionally be called "sparse") using a custom update that provides, after each shortest-path solve, a very effective "warm start" for subsequent solves.

Our motivating application is a new method to suggest for ships the shortest navigable path between any two ocean-navigable points on a spherical Earth that avoids obstacles (Washburn and Brown 2016). Obstacles are represented by nodes defining corners of spherical polygons, and the method benefits from defining a network with arc lengths between each pair of nodes for which there is an unobstructed great circle connecting them and then precomputing all of the shortest-path distances between all pairs of nodes. Other applications arise from air route planning and vehicle routing.

In each of these cases, the underlying directed graph is *symmetric* (i.e., for each arc $(i, j) \in A$, there is a corresponding *antisymmetric* arc $(j, i) \in A$), although the arc costs themselves might not be exactly symmetric: because of oceanic currents, winds aloft, or slight differences in directional node-to-node distances, altitudes, or times, we might have that $c_{ij} \neq c_{ji}$ for some (possibly all) arcs $(i, j) \in A$. We do not assume symmetry in the arc costs, but our algorithms expect symmetric arcs in the underlying graph; this can be accommodated by adding antisymmetric arcs where none exist with sufficiently high costs. If we define $C = \max_{(i,j)\in A}\{|c_{ij}|\}$ to be the maximum absolute arc cost, then no simple directed path in a network with n nodes can have length nC or greater, and therefore, we can use nC to represent such an "infinite" arc cost. As is typically done in shortest-path algorithms, we will also use nC as a distance label to represent a path that has not yet been found (or does not exist) between a particular pair of nodes.

Our key idea is to apply a shortest-path algorithm from each source node to all other (destination) nodes and take advantage of the *principle of optimality* for shortest-path problems: each application of a shortest-path algorithm yields a *tree of shortest paths* from the source to all other nodes, and every subpath in that tree must also be a shortest path. See Gallo (1980) for a thorough exposition of this idea. Any such subpath's length can be calculated quickly, and the collection of all of these values can be used to warm start subsequent applications of label correcting from other source nodes. When the underlying graph is symmetric, each shortest subpath in the tree has a corresponding feasible, but possibly suboptimal, reverse path between the same pair of nodes. If the network also has *symmetric costs* (i.e., $c_{ij} = c_{ji}$ for each arc (i,j) in a symmetric graph), then these reverse paths are also shortest paths. Regardless of whether these reverse paths are shortest paths, they still allow for an effective warm start of subsequent iterations. The additional computing required for this modification is not without cost, but our empirical analysis suggests that it is almost always worthwhile, except for very sparsely connected (and especially, disconnected) networks. We find that, even in cases with highly asymmetric costs, the computation improves the runtime of the underlying algorithm but that the benefit diminishes as the costs become increasingly asymmetric.

2. Solving the All-Pairs Shortest-Path Problem

For any network, the all-pairs shortest-path problem has an optimal solution that can be specified by (1) a set of distance labels, d_{kj}, giving for each node k and each node j the length of a shortest path from node k to node j and (2) the arcs in one of the shortest paths from each k to each j. The paths themselves are typically specified using a predecessor structure: $pred_{kj}$ is the node that immediately precedes node j on the shortest path from k to j. For any fixed node, k, the set of those values taken over all j provides a tree of shortest paths from node k to all other reachable nodes; if node j is not reachable from node k, then the associated distance label, d_{kj}, will be set to a large value (e.g., nC), and $pred_{kj}$ will be set to a flag value, such as zero, indicating that no path has been found.

The Floyd–Warshall algorithm (Warshall 1962) will solve any all-pairs shortest-path problem with a runtime bound of $O(n^3)$ (or determine that there is a negative cost directed cycle), but its primary drawback is that its theoretical runtime is its practical runtime, taking the full $O(n^3)$ regardless of sparsity, symmetry, or regularity in the arc costs. The Floyd–Warshall algorithm returns two dense matrices of size $O(n^2)$, one each for d_{ij} and $pred_{ij}$. If the network contains a negative cycle, then at least one diagonal entry will be negative, and the results can be used to help identify the negative cycle. If all diagonal entries are zero, then the result is optimal.

The all-pairs shortest-path problem can also be solved by simply solving the shortest-path problem from each start node to all other nodes in the network for a theoretical runtime of $O(nf(n,m))$, where the runtime of the particular shortest-path algorithm used is $O(f(n,m))$. If all arc costs are nonnegative, we can use Dijkstra's algorithm (Dijkstra 1959) with a runtime of $f(n,m) = O(n^2)$, giving a theoretical runtime of $O(n^3)$ for the repeated shortest-path algorithm. Theoretical speedups to Dijkstra's algorithm can be used as well, although they are of widely varying practical impact. We have implemented and tested a two-heap version of Dijkstra's algorithm on nonnegative data and report its performance, but we do not focus our exposition on that algorithm.

In the general case, where there might be negative costs on some arcs, we can use one of the many implementations of a *label-correcting* algorithm. We have implemented and tested two specific versions of the label-correcting shortest-path method in our repeated shortest-path algorithm: a *dequeue*-based algorithm with an exponential theoretical runtime but with very good practical performance that is fairly standard in the literature (Pape 1974 initially used the term *circular list*, adopting dequeue in Pape 1980; Gallo and Pallottino 1986 used *double-ended queue* and shortened it to *deque*) and a *two-queue* approach (also in Gallo and Pallottino 1986) that has a polynomial worst case runtime but is not covered in textbooks as often as or as thoroughly as dequeue implementations. We have found in practice that the two-queue algorithm is almost exactly as fast as the dequeue version, and given its speed and polynomial complexity, we focus the remainder of our presentation on that algorithm.

3. Summary of Label-Correcting Algorithms

The generic label-correcting algorithm for finding the shortest paths from a given start node to all other nodes starts by setting the distance label of each node to nC (representing an unreachable node) and then initializes the distance label of the start node to zero. It maintains a list of all "interesting" nodes: those nodes with distance labels that have been reduced from a prior suboptimal value but with a set of outbound arcs, or *adjacency list*, that has not been examined for possible updates. This list has the start node as its sole element at the beginning of the algorithm. Although this list is nonempty, the algorithm selects and removes a node, i, from the list, examines each arc in the adjacency list of node i for a possible distance label update, adds any node j that has had its distance label updated to the list (if it is not already on the list from an earlier update), and records for node j the predecessor node i that enabled the update. When the list is empty, no additional updates are possible, and the current distance labels and predecessor values are optimal.

The standard polynomial time implementation of label correcting is based on Bellman (1958), and it is sometimes referred to as the *Bellman–Ford* algorithm. The original presentation results in an algorithm that scans the entire list of arcs n times looking for distance label updates, yielding an O(nm) runtime. Contemporary treatments maintain the list of interesting nodes as a first-in, first-out (FIFO) queue that focuses only on the arcs that could possibly generate an update. However, even with this improvement, the algorithm is slow in practice; it can make a large number of small useless updates to the same set of nodes if it has not yet found a good path to those nodes.

One way to speed up a label-correcting algorithm is to prioritize finding optimal distance labels for the "old" nodes that it has already seen in prior updates before rushing to explore "new" nodes that have been updated for the first time (and that have potentially very poor distance labels). This is usually accomplished by partitioning the updated nodes into two separate sets. This works well in practice, providing significant improvements in practical runtime at a cost of theoretical worst case runtime. Glover et al. (1985) discuss the partitioning shortest-path algorithm, which subsumes several such algorithms.

The simplest implementation of this idea is to use a dequeue, or double-ended queue, to maintain the list of interesting nodes. Nodes are always selected from the front of the dequeue; however, new nodes are always added to the back of the dequeue, whereas old nodes are pushed onto the front of the dequeue to be processed as if they are on a stack. This ensures that any old nodes on the list will be processed before any new nodes. If there are no old nodes, the new nodes will be processed in FIFO order. The primary appeal of the dequeue data structure is that it can be kept in n contiguous memory locations with no concern about wraparound at either end: there is always enough room at the front of the dequeue for all of the old nodes in the network, and there is always room at the end of the dequeue for any remaining new nodes. Unfortunately, the stack-like behavior of this algorithm, especially toward the end of its operation when most nodes are old, ruins the polynomial runtime, and the best theoretical bound is exponential (Shier and Wizgall 1981). However, its practical performance is extremely fast on all but the most pathological examples.

A slightly more complicated method is to maintain two separate FIFO queues: one for the old nodes and one for the new nodes. If the old queue is nonempty, the algorithm draws from the front of it (in FIFO order now), but if it is empty, it draws from the front of the queue of new nodes (also in FIFO order). This algorithm achieves a polynomial runtime bound of O(n^2m) (Pallottino 1979), which is theoretically worse than the single-queue FIFO method. However, it is much faster in practice, doing as well as or even better than the dequeue method on almost all problems that we test. It requires a bit more care in managing the queues, but all operations can be done using a single array of integer pointers in a linked list data structure.

Our repeated shortest-path algorithm begins by initializing the data structures for the distance labels and predecessors; then, it loops over each source node, k, and uses a shortest-path algorithm to solve for shortest paths from node k to all other nodes, and it adds pre- and postprocessing routines to achieve a significant speedup over other algorithms for all-pairs shortest-path problems. Although we have also implemented and tested our algorithm with a two-heap version of Dijkstra's algorithm and a dequeue label-correcting algorithm, we present our implementation of the two-queue algorithm (referred to as "2Q" in our displays of results) explicitly to show exactly how our proposed speedup works.

4. A Two-Queue Implementation of Label Correcting

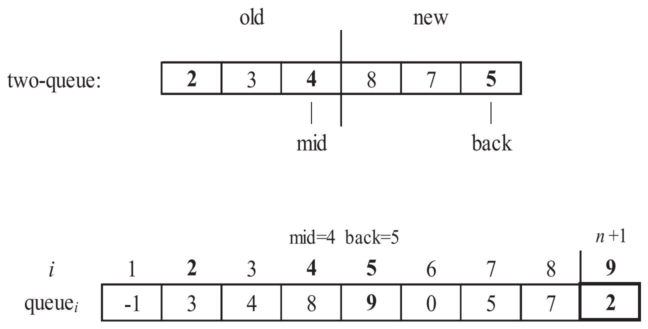
We assume that the nodes are indexed by consecutive integers from 1 to n. The arcs are stored in an adjacency list data structure: for each node i, adj_i is the list of nodes j adjacent to node i: $adj_i = \{j : (i, j) \in A\}$. We treat all arc costs c_{ij} and distance labels d_{ij} as integer values, although our algorithms will work (unmodified) with rational (floating point) data. All of the "pseudocode" in our figures is typeset in a fixed width font, whereas mathematical terms will be typeset in italics as usual. All of our subscripted values will be represented by arrays, with a separate set of brackets ("[. . .]") for each index. For example, the set of distance labels d_{ij} is represented in our code by a two-dimensional array, d, and an individual element is accessed as d[i][j]. All arguments to subroutines are assumed to be pass-by-value so that any modification made in a subroutine persists. Specifically, the d_{ij} and $pred_{ij}$ values will be updated by each call to a shortest-path subroutine. This avoids having to "return" their values explicitly.

For our two-queue implementation, we use a single array of integer pointers, $queue_i$, indexed from 1 to $n + 1$, with position $n + 1$ serving as a pointer to the first node in a single queue that represents the concatenation of the two queues that we wish to maintain with old nodes in the front and new nodes in back; if the queue is empty, then $queue_{n+1} = n + 1$. For each node i, $queue_i$ is the next node after node i in the queue in a "linked list" structure that eventually threads through all nodes currently on the queue. A node that is not in the queue at all and has not been seen before has $queue_i = 0$, and a node that is not currently in the queue but has been seen before has $queue_i = -1$. We use a pointer, *back*, to access the node in the last position of the queue. If the dequeue is empty, *back* will also have the value $n + 1$. We use a second pointer, *mid*, that refers to the last old node in the queue. The *mid* pointer is used to insert an updated old node into the correct position. If there is no old node in the queue, then *mid* will be set to $n + 1$, and therefore, an empty queue will also have $mid = n + 1$. Figure 1 illustrates the values in the *queue* array for a particular list of nodes in a two-queue data structure.

Figure 2 displays the pseudocode for our 2Q implementation. There are two lines annotated with brackets, **{Step 1: Optional Preprocessing}** and **{Step 2: Optional Postprocessing}**, which represent our proposed code for speeding up the repeated shortest-path algorithm for solving the all-pairs shortest-path problem. We will expand these sections in our discussion of node label updates.

The FIFO management of old nodes in the two-queue structure results in a breadth-first sequence for all node examinations. The two-queue algorithm starts by examining the start node, k, as a new node and any new nodes reached from k. If, at some point, there are any old nodes on the queue, the two-queue algorithm will process all of them (and any old nodes updated by this processing) before any more new nodes are encountered on the queue. It would take a pathological example to achieve the $O(n^2m)$ runtime bound for two queue; in practice, the runtimes are essentially linear in m and almost exactly the same as our (similarly implemented) dequeue algorithm. On almost every case that we study, the average number of times that a node reappears on the queue is very small: the average over all of our tests is between one and two appearances per node, with the most extreme cases in our test set having some nodes appear five or six times, even for networks with tens of thousands of nodes, with any degree of sparsity or density and for any distribution of arc costs.

Figure 1. Illustration of the Values in the *Queue* Array and the Pointers *Mid* and *Back* for a Two Queue on Eight Nodes with Six Nodes in the Queue



Notes. In this example, we have $n = 8$ nodes. Node 2 is at the front of the queue, and therefore, we have $queue_9 = 2$. Next in the queue is node 3, and therefore, $queue_2 = 3$; this continues to the last node in the queue, node 5, with $back = 5$ and $queue_5 = n + 1 = 9$. Node 4 is the last old node in the queue, and therefore, $mid = 4$. For this example, node 1 is not currently on the queue and has been seen before; therefore, $queue_1 = -1$, whereas node 6 has not been seen up to this point and has $queue_6 = 0$.

Figure 2. Pseudocode for a Two-Queue Implementation of the Label-Correcting Algorithm

```
begin two_queue(n,adj[],c[][],k,d[][],pred[][]):
    for i = 1 to n:
        queue[i] = 0
    next i
    queue[n+1] = n+1
    back = n+1
    {Step 1: Optional Preprocessing}
    i = k
    queue[k] = -1
    while i<=n:
        for j in adj[i]:
            if d[k][j] > d[k][i] + c[i][j]:
                d[k][j] = d[k][i] + c[i][j]
                pred[k][j] = i
                if queue[j] == 0:
                    queue[back] = j
                    queue[j] = n+1
                    back = j
                else if queue[j] < 0:
                    if mid == n+1:
                        queue[j] = queue[n+1]
                    else:
                        queue[j] = queue[mid]
                    end if
                    queue[mid] = j
                    if back == mid:
                        back = j
                    mid = j
                end if
            end if
        next j
        i = queue[n+1]
        queue[n+1] = queue[i]
        queue[i] = -1
        if back == i:
            back = n+1
        end if
        if mid == i:
            mid = n+1
        end if
    end while
    {Step 2: Optional Post-processing}
end twoqueue_lc
```

Notes. Our *queue* array of integer pointers provides an in-place linked list structure for easy addition and deletion of nodes; the *mid* pointer allows for adding nodes at the end of the queue of old nodes (and before the queue of new nodes). The two bracketed comments {Step 1:...} and {Step 2:...} indicate the location of our proposed speedups.

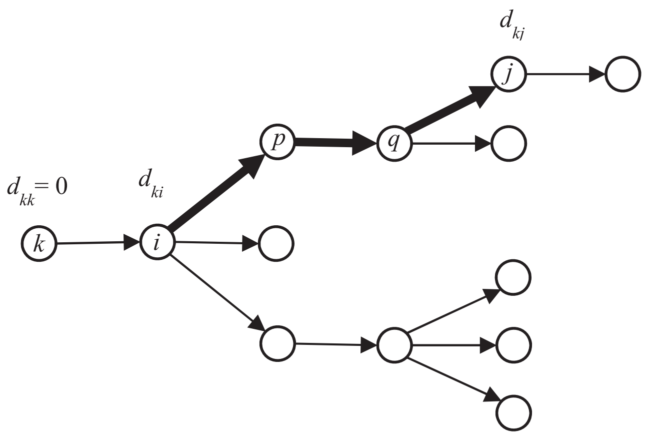
5. Practical Speedups to Repeated Shortest Paths

At each step of the repeated shortest-path algorithm, we have an opportunity to use the information obtained from running the shortest-path algorithm from any prior node. Let S_k be the set of all pairs of nodes, (i, j), that are connected by a directed path from i to j in the directed out tree of all shortest paths from node k. Note that, if $(i,j) \in S_k$, we can use the $pred_{kj}$ values to determine the path from i to j in that tree in reverse order from j back to i. Because subpaths of optimal paths are also optimal, if $(i,j) \in S_k$, then the corresponding path from i to j is a *shortest path* from i to j, and it has length $d_{kj} - d_{ki}$. Figure 3 illustrates this situation for a shortest-path tree rooted at node k and highlights a path from node i to node j, $\langle i, p, q, j \rangle$, that is a subpath in the tree.

After we have solved the shortest-path problem from, say, node 1, we have a shortest path and its length, d_{ij}, for each pair of nodes $(i,j) \in S_1$. We can use these distances and paths to reduce the number of computations required in the following shortest-path solves; this reduction depends on the size of S_k for each k. If our underlying graph is symmetric then for each pair of nodes $(i,j) \in S_1$, there is a corresponding reverse path from j back to i; we can use these reverse paths to generate initial estimates of the shortest-path distance from j to i, and we find that this improves the runtimes even if the costs are not symmetric. If the *costs* are also symmetric, then these reverse-path distances are optimal. For almost every test case that we examine, using this update in both directions in the shortest-path tree gives a significant speedup over the repeated shortest-path algorithm without the update, and in all cases except for extremely densely connected networks, it is significantly faster than Floyd–Warshall.

Our extension of the repeated shortest-path algorithm involves the two optional steps mentioned in

Figure 3. Illustration of a Shortest-Path Tree from Node k



Note. The path from i to j, indicated in bold, is a shortest path from i to j, and its length is $d_{kj} - d_{ki}$.

Figure 4. Pseudocode for {Optional Preprocessing} at Step k

```
for i = 1 to n:
    if i != k and d[k][i] < nC:
        queue[i] = queue[back]
        queue[back] = i
        back = i
    end if
next i
```

Notes. For a particular start node, k, each node i with a finite distance label from k is added to the queue of nodes to be examined as an old node. The placement of this code in the algorithm ensures that these nodes will be processed immediately after node k.

Figure 2. For a particular start node, k, the first optional step checks all other nodes to see if a prior iteration has discovered a path from k to that node (indicated by a value of d_{ki} less than nC) and if so, adds that node to the queue of updated nodes behind node k. This gives each application of the label-correcting algorithm after the first a warm start of optimal or near-optimal labels, and it has the benefit that any optimally labeled nodes will be updated once at the beginning of the subsequent label-correcting code and will never reappear on the queue. Figure 4 provides pseudocode for this preprocessing step.

The second step is our distance label update based on the shortest-path tree rooted at node k. We perform a complete exploration of the paths in that tree by using the $pred_{kj}$ values to climb the shortest-path tree from each node t toward the root, k. At every intermediate node, i, we calculate the length of the path from i to t as $d_{kt} - d_{ki}$, which is guaranteed to be optimal by the principle of optimality for shortest-path trees, and the length of the reverse path from t to i, dr, as a successive sum of reverse arc lengths (which is only guaranteed to be optimal in the case of symmetric arc costs but is a valid upper bound on, and may be close to, the length of a shortest path from t to i). If the new path length is an improvement over the current value, we update the appropriate distance label, d_{it} or d_{ti}, respectively. Figure 5 displays the pseudocode for our postprocessing "update" of the distance labels.

After we solve for the shortest paths from each source node, k, we process the shortest-path tree to revise shortest-path distances and predecessor values for each pair of nodes $(i,j) \in S_k$. Because these represent optimal distance labels or perhaps, near-optimal ones in the nearly symmetric case, when we solve for the shortest paths from node i, we treat them as having been seen before and put them at the back of the list of old nodes before processing begins for node i.

The update code in Figures 4 and 5 applies to any repeated shortest-path method with the same data structure for labels and shortest-path tree predecessors. This includes, for instance, the two-heap Dijkstra label-setting algorithm (see Johnson 1977

Figure 5. Pseudocode for {Optional Postprocessing}

```
for t = 1 to n:  # update forward- and reverse-path labels
    if t != k:
        predt = pred[k][t]
        dr = 0
        j = t
        i = pred[k][j]
        while i > 0:
            if i > k:  # i not used as a root yet
                if d[i][t] > d[k][t] - d[k][i]:
                    d[i][t] = d[k][t] - d[k][i]
                    pred[i][t] = predt
                end if
            end if
            {begin optional code for reverse paths}
            dr = dr + c[j][i]
            if t > k and dr < d[t][i]: # t not a root yet
                d[t][i] = dr
                pred[t][i] = j
            end if
            {end optional code}
            j = i
            i = pred[s][j]
        end while
    end if
next t
```

Notes. The first, "forward-path," loop defines optimal distance labels embedded in forward paths in the current shortest-path tree for trees not yet evaluated rooted at node *i* (namely, if node *i* hasn't been used as the "root," or start node, of any shortest-path tree). The latter "reverse-path" loop refines distance labels for trees not yet evaluated rooted at *t*. If the network has symmetric arc costs, these latter updates are optimal. Otherwise, these updates may be suboptimal, but they are valid distance labels and may speed up subsequent forward search tree computations. Our "simplified update" version of the code skips this second loop; this version can be faster if arc costs are highly nonsymmetric.

for a discussion of using heaps with Dijkstra's algorithm), although the preprocessing shown in Figure 4 must be modified to create the initial two heap of nodes with finite distance labels and seed it with source node k. Gallo and Pallottino (1986) survey a wide collection of such procedures, many of which (including the two-queue algorithm) fall under the general category of partitioning shortest-path procedures (Glover et al. 1985).

6. Computational Results

We randomly generate symmetric graphs with a given number of nodes, n, and a desired out-degree for the nodes, deg, using Bernoulli trials on each possible (undirected) edge in the graph and adding both (directed) arcs associated with each edge. Although this does not guarantee that our graphs are connected, for $deg = 3$, we observe that at least 90% of the nodes in our graphs are in a single connected component; this quickly approaches 100% as deg increases to four, five, and higher. Our algorithms do not require connectivity, of course, but we feel that connected graphs better represent real-world problems.

We create our network by adding a uniformly distributed integer arc cost, c_{ij}, drawn from the closed interval [100, 10,000] to one of each pair of antisymmetric arcs, and we use an additional "skew" factor, $0 \le \sigma \le 2$, as a parameter for generating reverse arc costs to represent the amount of asymmetry in those arc costs. If $\sigma = 0$, we set the reverse arc cost as $c_{ji} = c_{ij}$, yielding a network with symmetric costs. Otherwise, we draw c_{ji} uniformly from the interval $[(1 - \sigma/2)c_{ij}, (1 + \sigma/2)c_{ij}]$, preserving the nonnegativity of the arc costs while allowing for narrower or wider variations in the costs of asymmetric arcs as σ ranges up from zero. As an example, a skew factor of 0.05 yields reverse arcs with uniformly distributed 2.5% relative variation from the forward arc costs. We also have an option (signaled in our code by setting $\sigma = -1$) to generate reverse arcs completely independently and uniformly from the interval [100, 10,000].

We generate networks with n, the number of nodes, taking values in {1,000, 2,500, 5,000, 7,500, and 10,000}. For each value of n, we generate graphs with expected node out-degrees deg in the set {1, 2, 3, 4, 5, 10, 15, 20, 25, 50, 75, 100, 150, 200, 250, and 500}.

We also generate test problems on complete graphs, but Floyd–Warshall dominates all other algorithms on complete networks, and we do not report those results here. For each of these combinations of n and deg, we create networks with arc costs using skew σ in the set {0, 0.005, 0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 1.5, and 2.0} as well as completely uncorrelated cases (i.e., with skew $\sigma = -1$).

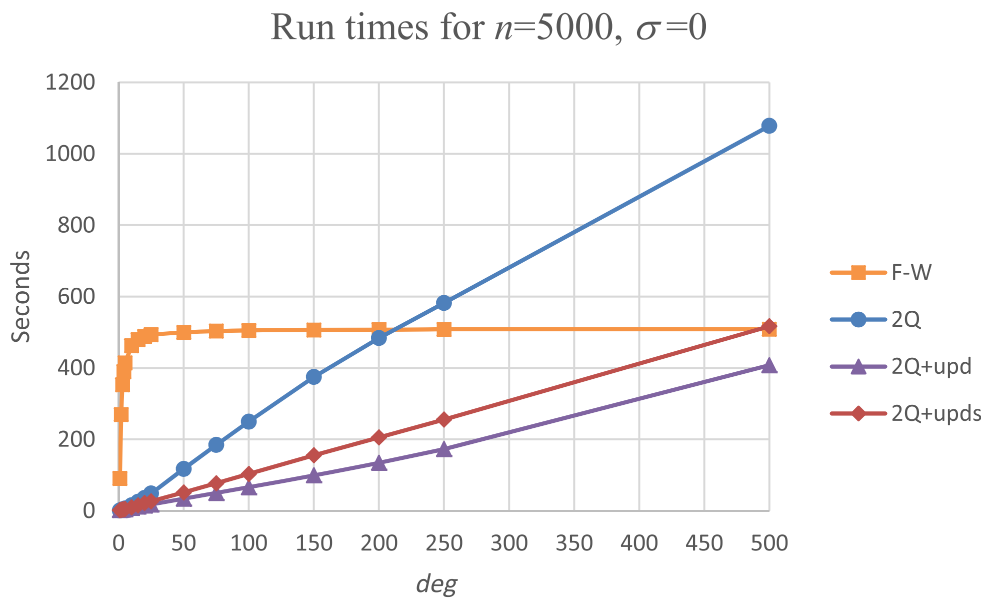
We have implemented and tested our algorithms in both optimized Fortran90 and (interpreted) Python 3.5.1 (e.g., www.python.org). All benchmarks have been run on a Lenovo P50 portable workstation with an Intel Xeon E3-1505M V5 chip rated at 2.80 GHz, single thread. For timings, we use INTEL Visual Fortran optimized for this 64-bit processer. The network data for computations reported in this paper can be downloaded from http://faculty.nps.edu/gbrown/downloads.htm.

For the smaller cases, we run a modified Floyd–Warshall algorithm ("F-W") that includes a test to skip unnecessary updates (i.e., those involving distance labels of nC on candidate paths). On all cases, we run a dequeue ("dQ") algorithm (not displayed) and two-queue ("2Q") implementations of the label-correcting algorithm from Figure 2, respectively. We also run the modified dequeue ("dQ + upd") and two-queue ("2Q + upd") algorithms with our full distance label update code as described in Figures 4 and 5, and, finally, the two-queue with a simplified update ("2Q + upds") omitting the "update reverse-path labels" loop from Figure 5. The online appendix displays our implementation of 2Q + upd in FORTRAN90. We also run a standard implementation of a two-heap version of Dijkstra's algorithm ("Dijk") and a full update version of two-heap Dijkstra using our update where the preprocessing step has been modified to run with a two heap ("Dijk + upd"). We measure runtimes to an approximate hardware accuracy of 0.02 seconds, and we report and plot the results in seconds.

Figure 6 displays the runtimes of our implementations, with and without the new update, as well as the runtime of Floyd–Warshall for 5,000-node networks with symmetric costs and deg ranging from 1 to 500. Except for the densest cases, the repeated shortest-path algorithms are significantly faster than Floyd–Warshall, and our new update code improves the runtime enough that, even at 10% density, (i.e., deg = 500 of 5,000), our new update versions are still faster than Floyd–Warshall.

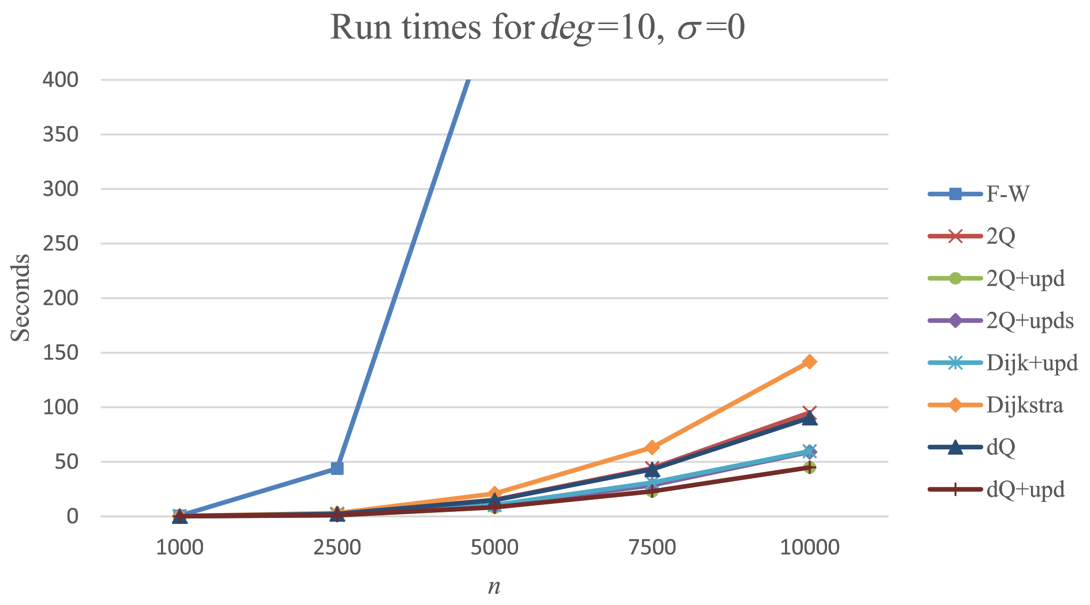
Figure 7 displays the runtimes of the repeated shortest-path algorithms on test cases of various sizes with n ranging from 1,000 up to 10,000, with deg = 10 and symmetric costs. The update version of each algorithm is noticeably faster than the non-update version, and the updated two-queue algorithm is consistently the fastest. Figure 8 displays the average number of times

Figure 6. (Color online) Runtimes of Floyd–Warshall vs. Two-Queue Label Correcting



Notes. For networks with 5,000 nodes and symmetric arc costs, the run times for the 2Q (in Figure 4, this algorithm is named "two_queue"), two-queue with update (2Q + upd), and two-queue with simplified update (2Q + upds) algorithms are significantly faster than Floyd–Warshall (F-W) for sparse graphs. As the expected node out-degree, deg, increases beyond 1% of the number of nodes (i.e., deg = 50), the runtime of Floyd–Warshall remains essentially constant, as expected, whereas the dependence of the label-correcting algorithms on the number of arcs leads to an increase in runtime that eventually surpasses that of Floyd–Warshall. As the density of connections in the network increases, the advantage of the distance label update becomes more pronounced over standard repeated shortest paths. Our full update version remains the fastest algorithm (even faster than Floyd–Warshall) beyond 10% density (deg = 500). The online appendix displays our implementation of 2Q + upd in FORTRAN90.

Figure 7. (Color online) Runtimes of Floyd–Warshall and the Seven Repeated Shortest-Path Algorithms as n Increases

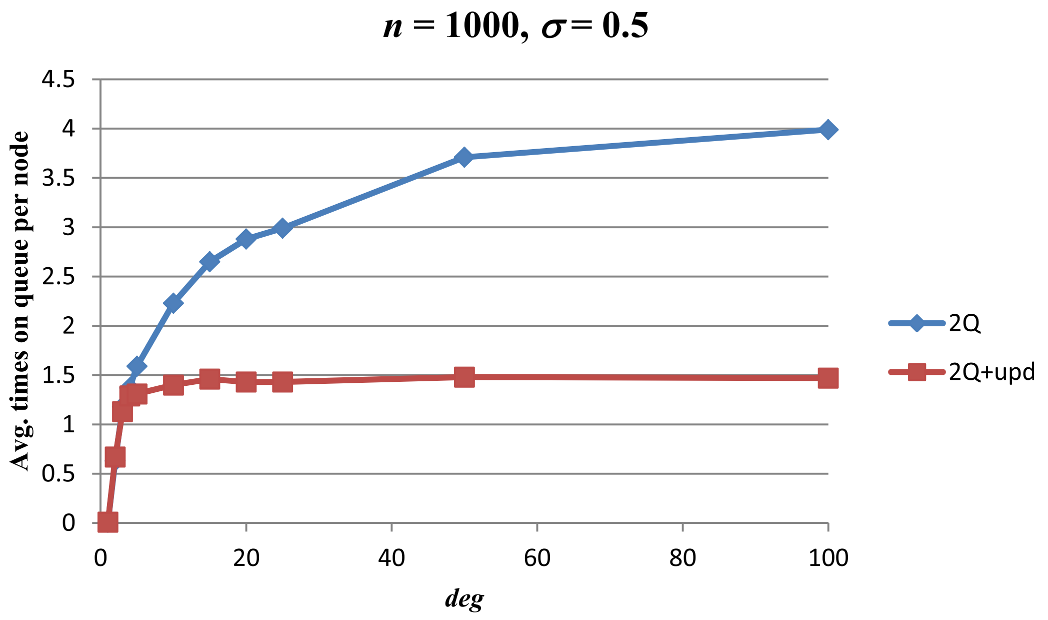


Notes. For symmetric networks with expected node out-degree $deg = 10$, we see that the update versions are consistently faster than the non-update versions, with the two-queue dominating the dequeue in both non-update and update versions. Floyd–Warshall is orders of magnitude slower for $n > 2{,}500$; at $n = 5{,}000$, its runtime is 513 seconds. For reference, the 10,000-node problem solves for 100,000,000 shortest paths.

that each node appears for processing at the front of the two-queue algorithm in both the regular version and the update version per shortest-path solve. The update version sees far fewer updates per node; over all of our test cases, the average is never higher than about 1.8, meaning that each node is far more likely to already have its optimal distance label by the time that it reaches the front of the queue in any particular solve.

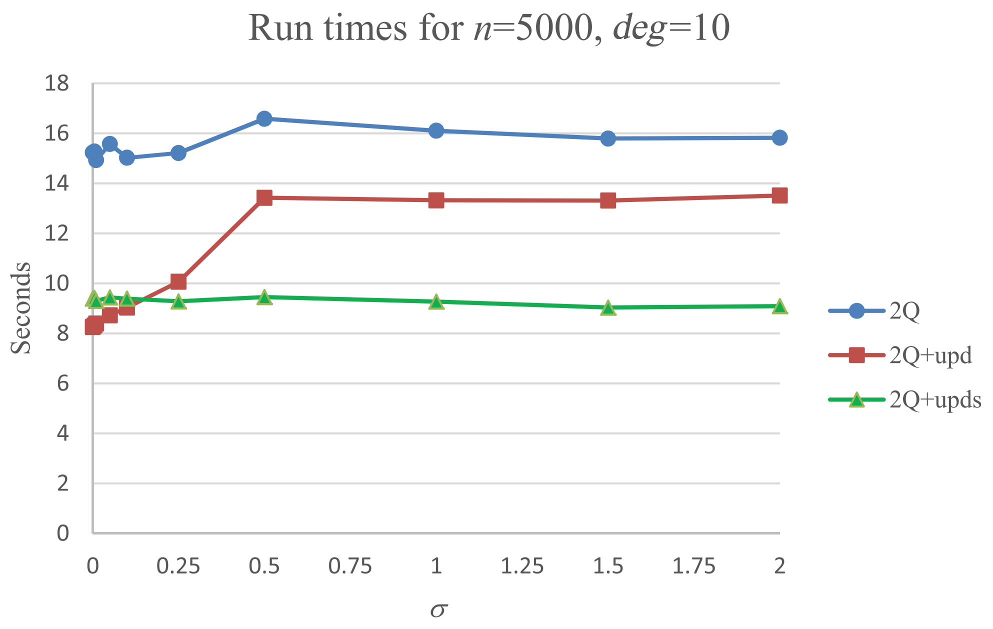
Figure 9 displays the runtime of the two-queue algorithm with and without our proposed update for networks with $n = 5{,}000$ nodes and expected out-degree $deg = 10$ and for the skew σ ranging from 0.0 to 2.0. Our update algorithm is especially helpful

Figure 8. (Color online) Average Number of Times That Each Node Appears on the Queue as Node Degree Increases



Notes. We see that the non-update version of the two-queue algorithm applied to networks with ±25% symmetry processes each node more times as the expected node degree increases but that our update version starts to flatten out at less than 1.5 appearances on the queue per node. The majority of nodes in the network only appear on the queue once, which means that, on average, most nodes already have their optimal distance label by the time that they reach the front of the queue.

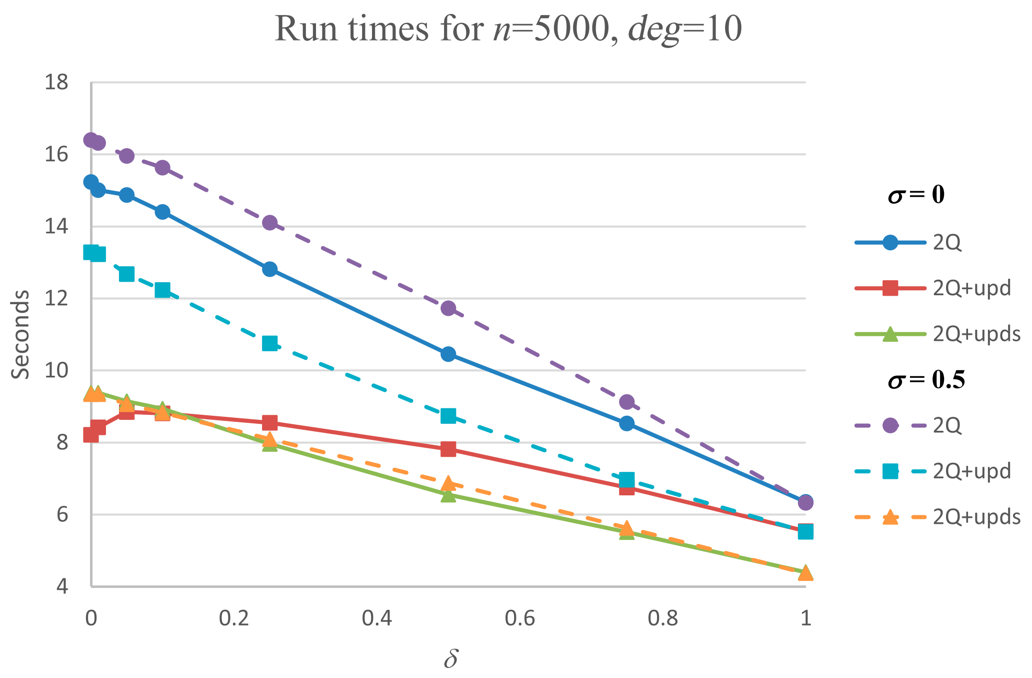
Figure 9. (Color online) Runtime of the Three Versions of the Two-Queue Algorithm as Skew Varies



Notes. The non-update version of the two-queue algorithm is essentially completely insensitive to skew, but our full update version enjoys a benefit that decreases as the reverse arc costs are less associated with the corresponding forward arc costs. The simplified update that skips the reverse arc updates is not as effective as the full update on nearly symmetric cost data, but as the skew increases past 0.25, the reduced overhead of the modified update allows it to solve more quickly. However, at all skew values, both update versions execute faster than the non-update version.

when the cost data are completely symmetric and the reverse paths of optimal paths are also optimal. As the costs become less and less symmetric, we gain less benefit from our update until the overhead from performing the update is more expensive than the benefit that we gain from it. In all cases that we examine, this

Figure 10. (Color online) Runtime of Two-Queue Algorithm with Standard and Modified Updates as the Number of Missing Reverse Arcs Varies for Two Values of Skew



Notes. As more reverse-arcs are removed from the network, the runtime of the standard update increases until about 50% of the reverse arcs are missing, at which point all algorithms start speeding up due to the reduced problem size and the reduced chance that any pair of nodes is connected by a directed path. However, our standard update is always faster than the non-update version, and it is only slightly slower than the modified update when a significant number of reverse arcs have been removed.

"breakeven" point occurs when $\sigma > 0.5$ (or positive parallel arc costs are within about $\pm 25\%$).

We are aware that randomly generated networks are notoriously harder to solve than real ones, but we prefer this in our tests. We have also had experience (Bradley et al. 1977) randomly generating networks attempting to produce special structure, such as grids, supply chains, etc. (see, e.g., Klingman et al. 1974) We do not test networks with negative arc lengths, although all of the label-correcting methods here can accommodate these: this avoids the need to find and eliminate negative cycles in our random test problems.

If the underlying graph is not symmetric (namely, if some arcs do not have corresponding antisymmetric arcs), then there might not be reverse paths corresponding to the forward paths between pairs of nodes, and our standard update might suffer as a result. We add a new parameter, $\delta \in [0, 1]$, to our network generation to represent the probability that an arc has an associated reverse arc. Specifically, for any particular value of $\delta \in [0, 1]$, whenever we select a pair of nodes i and j to create an arc, we choose the forward direction randomly (and uniformly), and after we generate the forward arc, we generate the reverse arc with probability $(1 - \delta)$. We tested our algorithms for values of $\delta \in \{0, 0.01, 0.05, 0.10, 0.25, 0.50, 0.75, 1.00\}$, and Figure 10 displays the effect of removing greater numbers of reverse arcs during network generation for purely symmetric costs and costs generated with a skew of 0.5. In particular, we see that having reverse arcs missing does not significantly degrade the performance of our update algorithm. As expected, for high-skew problems, our simplified update performs slightly better than the full update, but the difference diminishes as larger numbers of reverse arcs are missing. Both versions of the update are always better than the non-update version.

7. Conclusions and Insights

In sparsely connected networks, a repeated shortest-path algorithm will usually run significantly faster than the Floyd–Warshall algorithm for determining all-pairs shortest paths. With a distance label update based on the principle of optimality for shortest-path trees and especially if costs on antisymmetric pairs of arcs are close to each other, these algorithms can be modified to be even more efficient, in most cases performing less than two distance label updates per node on average. Our update procedures can be adapted to almost all of the common shortest-path routines, including those based on Dijkstra's algorithm (see Gallo 1980 for a discussion of how they can be used even in the presence of negative costs).

With new insight gained, we have discovered networks in legacy applications with symmetry and near symmetry that we had overlooked in the past. We are in the process of retrofitting the improvement reported here. Our original motivating problem was to improve route navigation software for a planning tool developed by one of the authors (Washburn and Brown 2016) in Microsoft Excel using Visual Basic for Applications and used as a component of the Oceanic Routing Service of the U.S. Navy. The original Floyd–Warshall implementation has been replaced by a repeated shortest-path algorithm with our proposed update, and on networks with several hundreds of nodes and arcs, the route planning tool runs faster by a factor of about seven.

References

Bellman R (1958) On a routing problem. *Quart. Appl. Math.* 16(1): 87–90.

Bradley G, Brown GG, Graves GW (1977) Design and implementation of large scale primal transshipment algorithms. *Management Sci.* 24(1):1–34.

Dijkstra EW (1959) A note on two problems in connexion with graphs. *Numerische Mathematik* 1(1):269–271.

Floyd RW (1962) Algorithm 97: Shortest path. *Comm. ACM* 5(6):345.

Gallo G (1980) Reoptimization procedures in shortest path problems. *Rivista matematica scienze economiche sociali* 3(1):3–13.

Gallo G, Pallottino S (1986) Shortest path methods: A unifying approach. *Math. Programming Stud.* 26:38–64.

Glover F, Klingman D, Phillips N (1985) A new polynomially bounded shortest path algorithm. *Oper. Res.* 33(1):65–73.

Ingerman PZ (1962) Algorithm 141: Path matrix. *Comm. ACM* 5(11):556.

Johnson DB (1977) Efficient algorithms for shortest paths in sparse networks. *J. ACM* 24(1):1–13.

Klingman D, Napier A, Stutz J (1974) NETGEN—A program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems. *Management Sci.* 20(5):814–822.

Pallottino S (1979) *Adaptation de l'algorithme de D'Esopo-Pape pour la determination de tous le chemins les plus courts: Ameliorations et simplifications. Publication 136, Centre de Recherchers sur les Transports* (University of Montreal, Montreal, Canada).

Pape U (1974) Implementation and efficiency of Moore-algorithms for the shortest route problem. *Math. Programming* 7(1): 212–222.

Pape U (1980) Algorithm 562: Shortest path lengths. *ACM Trans. Math. Software* 6(3):450–455.

Shier D, Wizgall C (1981) Properties of labeling methods for determining shortest path trees. *J. Res. National Bureau Standards* 86(3):317–330.

Warshall SW (1962) A theorem on Boolean matrices. *J. ACM* 9(1): 11–12.

Washburn AR, Brown GG (2016) An exact method for finding shortest routes on a sphere. *Naval Res. Logistics* 63(5):374–385.