Shortest-Paths Trees*

Bang Ye Wu Kun-Mao Chao

1 Introduction

Given a road map of the United States on which the distance between each pair of adjacent intersections is marked, how can a motorist determine the shortest route from New York City to San Francisco? The brute-force way is to generate all possible routes from New York City to San Francisco, and select the shortest one among them. This approach apparently generates too many routes that are not worth considering. For example, a route from New York City to Miami to San Francisco is a poor choice. In this chapter we introduce some efficient algorithms for finding all the shortest paths from a given starting location.

Consider a connected, undirected network with one special node, called the source (or root). Associated with each edge is a distance, a nonnegative number. The objective is to find the set of edges connecting all nodes such that the sum of the edge lengths from the source to each node is minimized. We call it a *shortest-paths tree* (SPT) rooted at the source.

In order to minimize the total path lengths, the path from the root to each node must be a shortest path connecting them. Otherwise, we substitute such a path with a shortest path, and get a "lighter" spanning tree whose total path lengths from the root to all nodes are smaller.

Shortest-paths trees are not necessarily unique. Figure 2 gives two shortest-paths trees rooted at vertex a for the graph from Figure 1. Take a look at the paths from a to e. In Figure 2(a), it goes from a to g, and then g to e. In Figure 2(b), it goes from a to b, b to d, and then d to e. Both of them are of length 7, which is the length of a shortest path from a to e. Notice that the total edge weight of two shortest-paths trees may not be the same. For example, the total edge weight of the shortest-paths tree in Figure 2(a) is 18, whereas that of the shortest-paths tree in Figure 2(b) is 17.

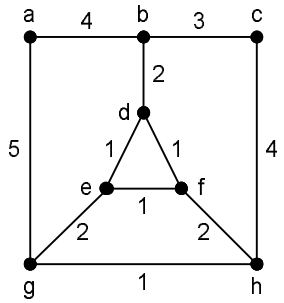

Figure 1: A weighted graph.

*An excerpt from the book "Spanning Trees and Optimization Problems," by Bang Ye Wu and Kun-Mao Chao (2004), Chapman & Hall/CRC Press, USA.

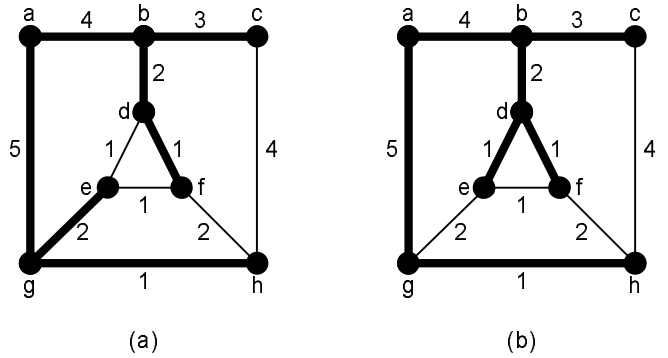

Figure 2: A shortest-paths tree rooted at vertex a for the graph from Figure 1.

In fact, the total edge weight of a minimum spanning tree (see Figure 3) is 14.

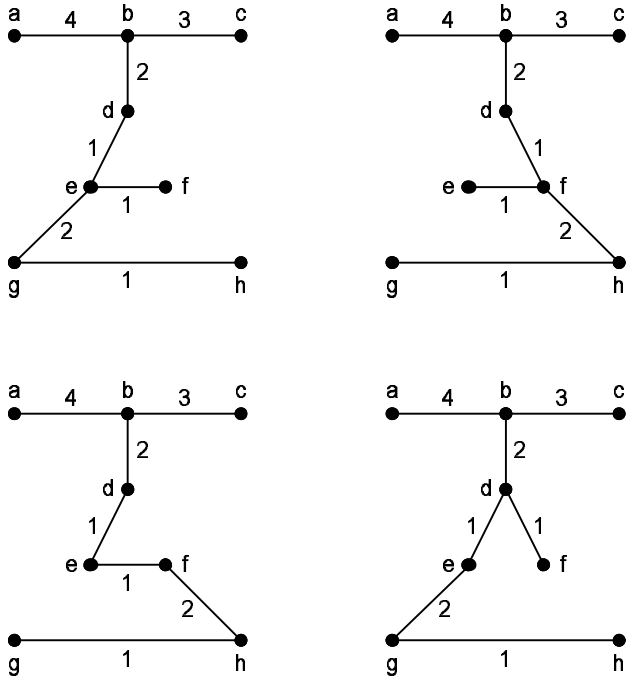

Figure 3: Some minimum spanning trees.

As long as all the edge weights are nonnegative, the shortest-paths tree is well defined. Unfortunately, things get somewhat complicated in the presence of negative edge weights. For an undirected graph, a long path gets "shorter" when we repeatedly add an edge with negative weight to it. In this situation, a shortest path that contains an edge with negative weight is not well defined since a lesser-weight path can always be found by going back and forth on the negative-weight edge. Consider the graph in Figure 4. In this graph, edge (d, e) is of negative weight. Since, for an undirected graph, an edge can be traversed in both directions, a path that repeatedly uses (d, e) will reduce its length. However, for a directed graph, as long as there exists no negative-weight cycle reachable from the source, the shortest-path weights are well defined. Thus when talking about the

topics related to shortest paths, we usually focus on solving problems in directed graphs. It should be noted, however, that most such algorithms can be easily adapted for undirected graphs.

In this chapter, the terms "edge" and "arc" are used interchangeably. We discuss two well-known algorithms for constructing a shortest-paths tree: Dijkstra's algorithm and the Bellman-Ford algorithm. Dijkstra's algorithm assumes that all edge weights in the graph are nonnegative, whereas the Bellman-Ford algorithm allows negative-weight edges in the graph. If there is no negative-weight cycle, the Bellman-Ford algorithm produces the shortest paths and their weights. Otherwise, the algorithm detects the negative cycles and indicates that no solution exists.

2 Dijkstra's Algorithm

Djikstra's algorithm solves the problem of finding the shortest path from a source to a destination. It turns out that one can find the shortest paths from a given source to all vertices in a graph in the same time; hence, this problem is sometimes called the single-source shortest paths problem. In fact, this algorithm can be used to deliver the set of edges connecting all vertices such that the sum of the edge lengths from the source to each node is minimized.

For each vertex $v \in V$, Dijkstra's algorithm maintains an attribute $\delta[v]$, which is an upper bound on the weight of a shortest path from the source to v. We call $\delta[v]$ a *shortest-path estimate*. Initially, the shortest-path estimates of all vertices other than the source are set to be ∞. Dijkstra's algorithm also maintains a set S of vertices whose final shortest-path weights from the source have not yet been determined. The algorithm repeatedly selects the vertex $u \in S$ with the minimum shortest-path estimate, and re-evaluates the shortest-path estimates of the vertices adjacent to u. The re-evaluation is often referred to as a *relaxation* step. Once a vertex is removed from S, its shortest-path weight from the source is determined and finalized.

Algorithm: DIJKSTRA
Input: A weighted, directed graph $G = (V, E, w)$; a source vertex s.
Output: A shortest-paths spanning tree T rooted at s.

for each vertex $v \in V$ **do**
    $\delta[v] \leftarrow \infty$
    $\pi[v] \leftarrow$NIL
$\delta[s] \leftarrow 0$
$T \leftarrow \emptyset$
$S \leftarrow V$

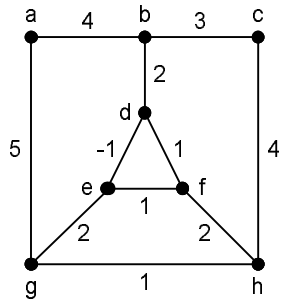

Figure 4: An undirected graph with a negative-weight edge.

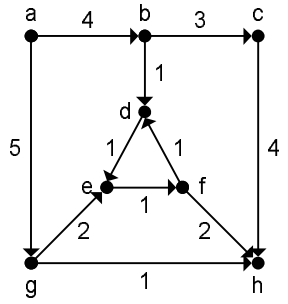

Figure 5: A weighted, directed graph.

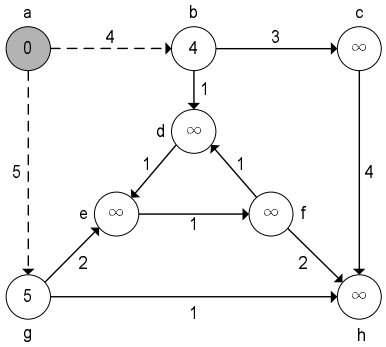

Figure 6: Vertex a is chosen, and edges (a, b) and (a, g) are relaxed.

while $S \neq \emptyset$ **do**
    Choose $u \in S$ with minimum $\delta[u]$
    $S \leftarrow S - \{u\}$
    if $u \neq s$ **then** $T \leftarrow T \cup \{(\pi[u], u)\}$
    for each vertex v adjacent to u **do**
        if $\delta[v] > \delta[u] + w(u, v)$ **then**
            $\delta[v] \leftarrow \delta[u] + w(u, v)$
            $\pi[v] \leftarrow u$

Consider the graph in Figure 5. The following figures illustrate how Dijkstra's algorithm works in constructing a shortest-paths tree rooted at vertex a.

Initially, all $\delta[\cdot]$ values are ∞, except $\delta[a] = 0$. Set S contains all vertices in the graph. Vertex a, shown as a shaded vertex in Figure 6, has the minimum δ value and is chosen as vertex u in the **while** loop. We remove a from S. Then edges (a, b) and (a, g), shown as dotted lines, are relaxed.

Now shaded vertex b in Figure 7 has the minimum δ value among all vertices in S and is chosen as vertex u in the **while** loop. We remove b from S. Dark edge (a, b) is added to the shortest-paths tree under construction, and dotted edges (b, c) and (b, d) are relaxed.

Both vertices d and g in Figure 8 have the minimum δ values among all vertices in S. Let us choose g as vertex u in the **while** loop. We remove g from S. Dark edge (a, g) is added to the shortest-paths tree, and dotted edges (g, e) and (g, h) are relaxed.

Since vertex d in Figure 9 has the minimum δ value among all vertices in S, it is chosen as

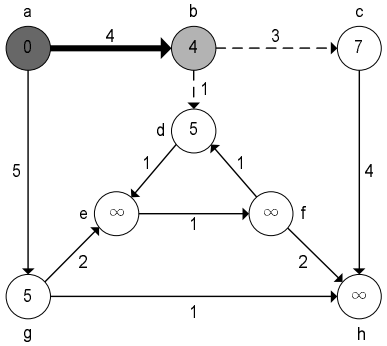

Figure 7: Vertex b is chosen, and edges (b, c) and (b, d) are relaxed. Edge (a, b) is added to the shortest-paths tree.

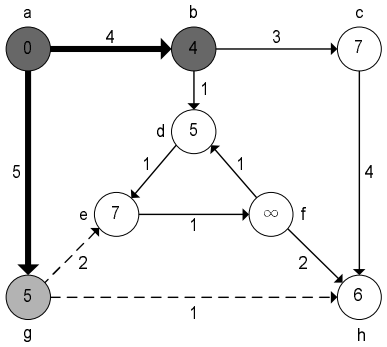

Figure 8: Vertex g is chosen, and edges (g, e) and (g, h) are relaxed. Edge (a, g) is added to the shortest-paths tree.

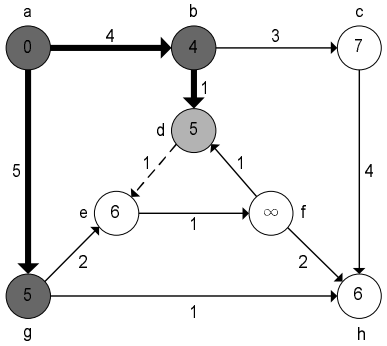

Figure 9: Vertex d is chosen, and edge (d, e) is relaxed. Edge (b, d) is added to the shortest-paths tree.

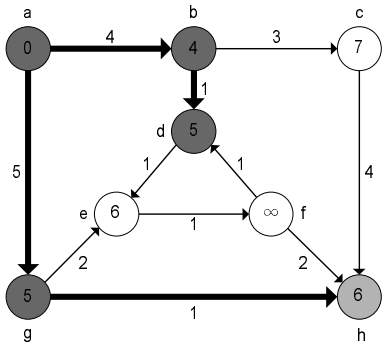

Figure 10: Vertex h is chosen. Edge (g, h) is added to the shortest-paths tree.

vertex u in the **while** loop. We remove d from S. Dark edge (b, d) is added to the shortest-paths tree, and dotted edge (d, e) is relaxed.

Both vertices e and h in Figure 10 have the minimum δ values among all vertices in S. Let us choose h as vertex u in the **while** loop. We remove h from S. Dark edge (g, h) is added to the shortest-paths tree. No edge is relaxed.

Now vertex e in Figure 11 has the minimum δ value among all vertices in S and is chosen as vertex u in the **while** loop. We remove e from S. Dark edge (d, e) is added to the shortest-paths tree under construction, and dotted edge (e, f) is relaxed.

Both vertices c and f in Figure 12 have the minimum δ values among all vertices in S. Let us choose c as vertex u in the **while** loop. We remove c from S. Dark edge (b, c) is added to the shortest-paths tree, and dotted edge (c, h) is relaxed.

Now vertex f is the only vertex in S (see Figure 13), and is chosen as vertex u in the **while** loop. We remove f from S. Dark edge (e, f) is added to the shortest-paths tree, and dotted edges (f, d) and (f, h) are relaxed.

Figure 14 gives the shortest-paths tree constructed by Dijkstra's algorithm.

You may have found in the previous example that once the vertex is removed from S, its δ value remains unchanged. This holds as long as all edge weights are nonnegative. The rationale behind this is very straightforward because we always choose the vertex in S with the *minimum* δ value. However, this may not hold for a directed graph with negative-weight edges.

Dijkstra's algorithm selects the vertex $u \in S$ with the minimum shortest-path estimate, and

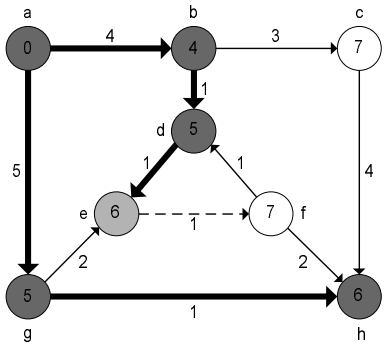

Figure 11: Vertex e is chosen, and edge (e, f) is relaxed. Edge (d, e) is added to the shortest-paths tree.

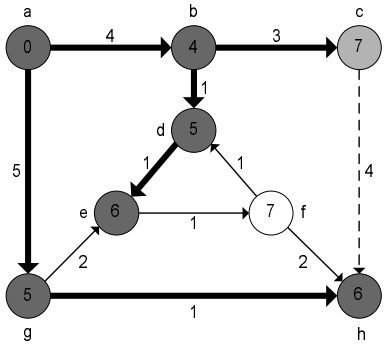

Figure 12: Vertex c is chosen, and edge (c, h) is relaxed. Edge (b, c) is added to the shortest-paths tree.

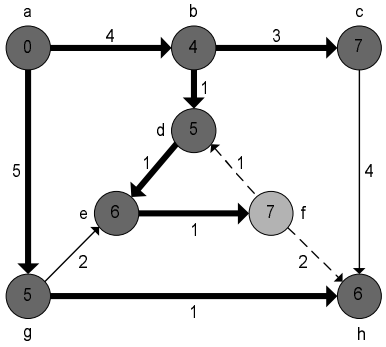

Figure 13: Vertex f is chosen, and edges (f, d) and (f, h) are relaxed. Edge (e, f) is added to the shortest-paths tree.

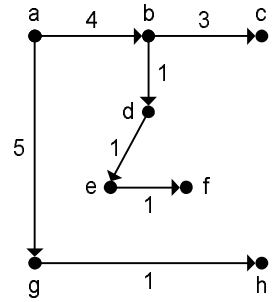

Figure 14: A shortest-paths tree constructed by Dijkstra's algorithm for the graph in Figure 5.

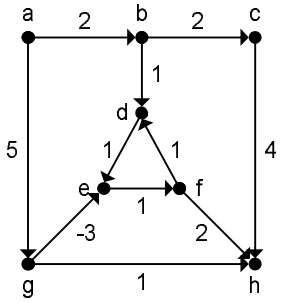

Figure 15: A weighted, directed graph with a negative-weight edge.

re-evaluates the shortest-path estimates of the vertices adjacent to u. The worst-case complexity of Dijkstra's algorithm depends on the way of finding the node with the minimum shortest-path estimate. A naive implementation that examines all nodes in S to find the minimum runs in $O(n^2)$ time. A modified Dijkstra's algorithm, where the minimum can be found by using binary heaps in $O(\log n)$ time, runs in $O(m \log n)$ time. By using Fibonacci heaps, we can achieve a running time of $O(m + n \log n)$.

Figure 15 gives an example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces an incorrect answer.

As Dijkstra's algorithm proceeds on the graph in Figure 15, it will reach the status shown in Figure 16. Since edge (g, e) is of negative weight -3, the weight of the path (a, g, e) is $5 - 3 = 2$. It is better than the weight of the finalized shortest-path (a, b, d, e), which is of total weight 4. Furthermore, edge (e, f) has been relaxed. There is no way for Dijkstra's algorithm to re-relax all edges affected by this domino effect.

The graph in Figure 17 is a wrong shortest-path tree produced by Dijkstra's algorithm. Notice that (g, e) is of negative weight, and is missing in the shortest-paths tree constructed by Djikstra's algorithm.

A correct shortest-paths tree is given in Figure 18.

3 The Bellman-Ford Algorithm

Shortest-paths algorithms typically exploit the property that a shortest path between two vertices contains other shortest paths within it. This optimal substructure property admits both dynamic programming and the greedy method. We have shown that Dijkstra's algorithm might fail in

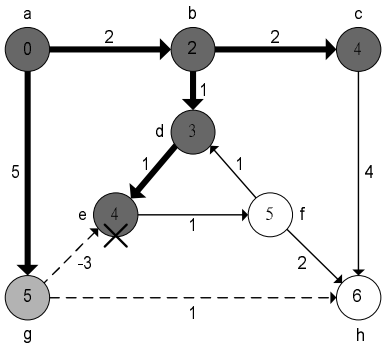

Figure 16: A failure of Dijkstra's algorithm.

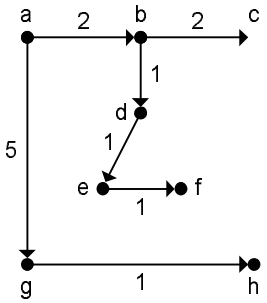

Figure 17: A wrong shortest-paths tree.

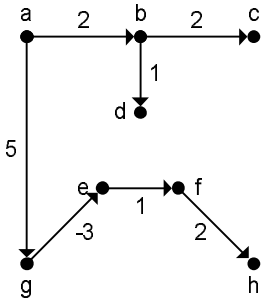

Figure 18: A correct shortest-paths tree.

delivering a shortest-paths tree when the graph has some negative-weight edges. The Bellman-Ford algorithm works in a more-general case than Dijkstra's algorithm. It solves the shortest-paths tree problem even when the graph has negative-weight edges. If there is a negative-weight cycle, the algorithm indicates that no solution exists.

Recall that Dijkstra's algorithm only relaxes those edges incident to the chosen vertex with the minimum shortest-path estimate. The failure of Dijkstra's algorithm for graphs with negative-weight edges is due to the fact that it does not calculate the domino effect caused by negative edges. Take a look at Figure 16 once again. In that figure, if we push the negative-edge effect forward, a correct shortest-paths tree can then be built. This inspires the design of the Bellman-Ford algorithm which relaxes all edges in each iteration.

Initially, the shortest-path estimates of all vertices other than the source are set to be ∞. Then the algorithm makes $n-1$ passes over all the edges of the graph. In each pass, if $\delta[v] > \delta[u]+w(u,v)$, then we set the value of $\delta[v]$ to be $\delta[u]+w(u,v)$, and modify the predecessor of vertex v. A notable feature of the Bellman-Ford algorithm is that in the k^{th} iteration, the shortest-path estimate for vertex v, i.e., $\delta[v]$, equals the length of the shortest path from the source to v with at most k edges. If, after $n-1$ passes, there exists an edge (u,v) such that $\delta[v] > \delta[u]+w(u,v)$, then a negative cycle has been detected. Otherwise, for all vertices other than the source, we build the shortest-paths tree by adding the edges from their predecessors to them.

Algorithm: BELLMAN-FORD
Input: A weighted, directed graph $G = (V, E, w)$; a source vertex s.
Output: A shortest-paths spanning tree T rooted at s.

```
for each vertex v ∈ V do
    δ[v] ← ∞
    π[v] ←NIL
δ[s] ← 0
for i ← 1 to n − 1 do
    for each (u, v) ∈ E do
        if δ[v] > δ[u] + w(u, v) then
            δ[v] ← δ[u] + w(u, v)
            π[v] ← u
for each (u, v) ∈ E do
    if δ[v] > δ[u] + w(u, v) then
        Output "A negative cycle exists."
        Exit
T ← ∅
for v ∈ V − s do
    T ← T ∪ {(π[v], v)}
```

The Bellman-Ford algorithm has the running time of $O(mn)$ since there are $O(n)$ iterations, and each iteration takes $O(m)$ time. The correctness follows from the fact that in the k^{th} iteration, the shortest-path estimate for each vertex equals the length of the shortest path from the source to that vertex with at most k edges. Since a simple path in G contains at most $n-1$ edges, the shortest-path estimates stabilize after $n-1$ iterations unless there exists a negative cycle in the graph.

The following figures show how the execution of the BELLMAN-FORD works on the graph in Figure 15. The dark dotted edges record those that do cause some effect in the relaxation. Initially,

$\delta[a] = 0$. In Figure 19, the relaxation of edge (a, b) changes $\delta[b]$ from ∞ to 2; and the relaxation of edge (a, g) changes $\delta[g]$ from ∞ to 5. Shortest-path estimates $\delta[a]$, $\delta[b]$, and $\delta[g]$ are finalized. Shortest-path estimates $\delta[c]$, $\delta[d]$, $\delta[e]$, $\delta[f]$, and $\delta[h]$ are still ∞ since their corresponding vertices cannot be reached from vertex a by a path with only one edge.

In Figure 20, the relaxation of edge (b, c) changes $\delta[c]$ from ∞ to 4; the relaxation of edge (b, d) changes $\delta[d]$ from ∞ to 3; the relaxation of edge (g, e) changes $\delta[e]$ from ∞ to 2; and the relaxation of edge (g, h) changes $\delta[h]$ from ∞ to 6. Up to this stage, shortest-path estimates $\delta[a]$, $\delta[b]$, $\delta[c]$, $\delta[d]$, $\delta[e]$, and $\delta[g]$ are finalized. Shortest-path estimate $\delta[f]$ is still ∞ since vertex f cannot be reached from vertex a by a path with at most two edges, and shortest-path estimate $\delta[h]$ will be modified later.

In Figure 21, the relaxation of edge (e, f) changes $\delta[f]$ from ∞ to 3.

In Figure 22, the relaxation of edge (f, h) changes $\delta[h]$ from 6 to 5.

Figure 23 gives the final shortest-paths tree constructed by the Bellman-Ford algorithm.

4 Applications

The shortest-paths tree problem comes up in practice and arises as a subproblem in many network optimization algorithms. The shortest path tree is widely used in IP multicast and in some of the application-level multicast routing algorithms.

4.1 Multicast

In the age of multimedia and high-speed networks, multicast is one of the mechanisms by which the power of the Internet can be further utilized in an efficient manner. When more than one receiver is interested in receiving a transmission from a single or a set of senders, multicast is the most efficient and viable mechanism.

Multicast routing refers to the construction of a tree rooted at the source and spanning all destinations. Generally, there are two types of such a tree, Steiner trees and shortest-paths trees. Steiner trees or group-shared trees tend to minimize the total cost of the resulting trees. Shortest-paths trees or source-based trees tend to minimize the cost of each path from source to any destination.

In the following, we briefly describe two well-known routing protocols: the Routing Information Protocol (RIP) and Open Shortest Path First (OSPF).

RIP is a distance-vector protocol that allows routers to exchange information about destinations for computing routes throughout the network. Destinations may be networks or a special

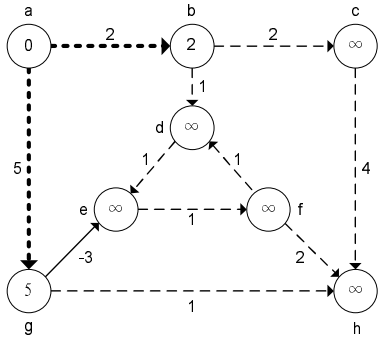

Figure 19: Shortest-path estimates $\delta[b]$ and $\delta[g]$ are modified.

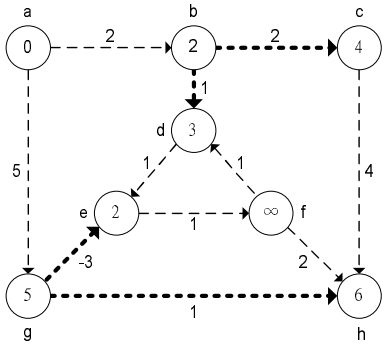

Figure 20: Shortest-path estimates $\delta[c]$, $\delta[d]$, $\delta[e]$, and $\delta[h]$ are modified.

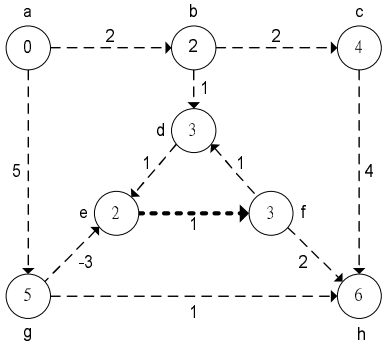

Figure 21: Shortest-path estimate $\delta[f]$ is modified.

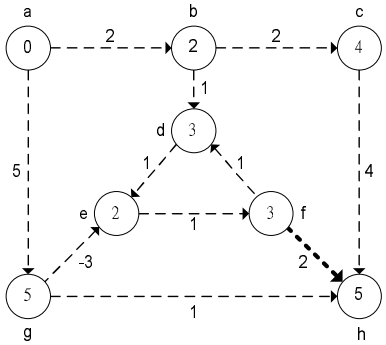

Figure 22: Shortest-path estimate $\delta[h]$ is modified.

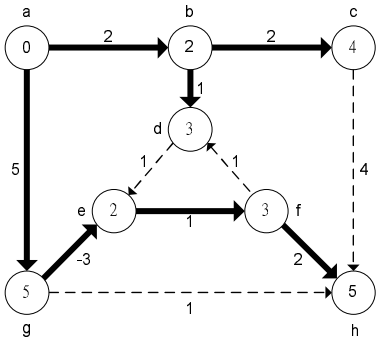

Figure 23: The shortest-paths tree constructed by the Bellman-Ford algorithm.

destination used to convey a default route. In RIP, the Bellman-Ford algorithms make each router periodically broadcast its routing tables to all its neighbors. Then a router knowing its neighbors' tables can decide to which destination neighbor to forward a packet.

OSPF is a routing protocol developed for Internet Protocol (IP) networks by the Interior Gateway Protocol (IGP) Working Group of the Internet Engineering Task Force (IETF). OSPF was created because in the mid-1980s, RIP was increasingly incapable of serving large, heterogeneous internetworks. Like most link-state algorithms, OSPF uses a graph-theoretic model of network topology to compute shortest paths. Each router periodically broadcasts information about the status of its connections. OSPF floods information about adjacencies to all routers in the network where each router locally computes the shortest paths by running Dijkstra's algorithm.

4.2 SPT-based approximations

Besides their applications to network routing problems, the shortest-paths tree algorithms could also serve as good approximations for some NP-hard problems. For example, we will later show that a shortest-paths tree rooted at some vertex is a 2-approximation of the minimum routing cost spanning tree (MRCT) problem, which is known to be NP-hard. In fact, several SPT-based approximations will be studied in the next chapters.

5 Summary

We have introduced two most basic algorithms for constructing a shortest-paths tree for a given directed or undirected weighted graph. Both of them use the technique of relaxation, progressively decreasing a shortest-path estimate $\delta[v]$ for each vertex v. The relaxation causes the shortest-path estimates to descend monotonically toward the actual shortest-path weights. Dijkstra's algorithm relaxes each edge exactly once (twice in the case of undirected graphs) if all the edge weights are positive. On the other hand, the Bellman-Ford algorithm relaxes each edge $n-1$ times, so that the effect of a negative edge can be propagated properly. If the shortest-path estimates do not stabilize after $n-1$ passes, then there must exist a negative cycle in the graph, and the algorithm indicates that no solution exists.

Bibliographic Notes and Further Reading

The shortest-paths tree problem is one of the most classical network flow optimization problems. An equivalent problem is to find a shortest path from a given source vertex $s \in V$ to every vertex $v \in V$. Algorithms for this problem have been studied for a long time. In fact, since the end of the 1950s, thousands of scientific works have been published. A good description of the classical algorithms and their implementations can be found in [2, 8].

Dijkstra's original algorithm, by Edsger W. Dijkstra [5], did not mention the usage of a priority queue. A discussion of using different priority queue techniques can be found in [2, 3].

For the shortest path problem with nonnegative arc lengths, the Fibonacci heap data structure [7] yields an $O(m + n \log n)$ implementation of Dijkstra's algorithm in the pointer model of computation. Let U denote the biggest arc length, and C be the ratio between U and the smallest nonzero arc length. In a RAM model with word operations, the fastest known algorithms achieve the following bounds: $O(m + n(\sqrt{\log n})$ [12], $O(m + n(\log C \log\log C)^{1/3})$ [9, 13], $O(m \log\log U)$ [10], and $O(m \log\log n)$ [15]. Ulrich Meyer [11] shows the problem can be solved in linear average time if input arc lengths are independent and uniformly distributed. Andrew V. Goldberg [9] shows that a simple modification of the algorithm of [4] yields an algorithm with linear average running time on the uniform arc length distribution. For undirected graphs, Mikkel Thorup [14] gave a linear-time algorithm in a word RAM model.

The Bellman-Ford algorithm is based on separate algorithms by Richard Bellman [1], and Lestor Ford, Jr. and D.R. Fulkerson [6]. Though the Bellman-Ford algorithm is simple and has a high running time, to date there is no algorithm which significantly improves its asymptotic complexity.

References

[1] R. Bellman. On a routing problem. *Quar. Appl. Math.*, 16:87–90, 1958.

[2] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 516–525, 1994.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, 1994.

[4] E.V. Denardo and B.L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Oper. Res.*, 27:161–186, 1979.

[5] E.W. Dijkstra. A note on two problems in connection with graphs. *Numer. Math.*, 1:269–271, 1959.

[6] L.R. Ford, Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[8] G. Gallo and S. Pallottino. Shortest paths algorithms. *Ann. Oper. Res.*, 13:3–79.

[9] A.V. Goldberg. A simple shortest path algorithm with linear average time. In *Proceedings of the 9th Annual European Symposium Algorithms*, pages 230–241, 2001.

[10] T. Hagerup. Improved shortest paths in the word RAM. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 61–72, 2000.

[11] U. Meyer. Single-source shortest paths on arbitrary directed graphs inlinear average time. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 797–806, 2001.

[12] R. Raman. Priority queues: small, monotone and trans-dichotomous. In *Proceedings of the 4th Annual European Symposium Algorithms*, pages 121–137, 1996.

[13] R. Raman. Recent results on single-source shortest paths problem. *SIGACT News*, 28:81–87, 1997.

[14] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46:362–394, 1999.

[15] M. Thorup. On RAM priority queues. *SIAM J. Comput.*, 30:86–109, 2000.