1

Preliminaries: Shortest Path Algorithms

> The White Rabbit put on his spectacles. "Where shall I begin, please your Majesty?" he asked.
>
> "Begin at the beginning," the King said, very gravely, "and go on till you come to the end: then stop."
>
> – Lewis Carroll, *Alice in Wonderland*

Although we will assume that the reader has previously studied combinatorial algorithms, it is useful to start by presenting algorithms for computing a shortest path. Anyone who has studied combinatorial algorithms before will certainly have encountered these algorithms, but the ideas in them are so central to the topic of network flow algorithms that a short overview of the two most fundamental algorithms is in order.

In the shortest path problem we are given a directed graph $G = (V, A)$, and a distinguished vertex s, which we will call the *source*. For each arc $(i, j) \in A$ we are given a cost $c(i, j)$ of traveling from i to j. A non-empty *path* from s to i is a sequence of arcs $(s, j_1), (j_1, j_2), (j_2, j_3), \ldots, (j_k, i)$ that starts at s, ends at i, and such that the head of each arc is the tail of the next. If no vertex is repeated in the path, it is a *simple* path. A path that starts and ends at the same vertex is called a *cycle*. A *simple cycle* is a cycle in which only the start and end vertices are repeated.

For each $i \in V$ we want to compute a path from s to i of minimum total cost, if such a path exists; we will call it a *minimum-cost* path from s to i. We will let $d(i)$ denote the cost of such a path. If there is no path from s to i in G, then we will set $d(i)$ to ∞. As we will see in a moment, it is possible that the minimum-cost path is not well defined – this can occur if there exists a cycle such that the total cost of the arcs in the cycle is negative – and we will eventually discuss this issue. Since we are assigning costs, rather than lengths,

to arcs, we could refer to the cheapest path problem, rather than the shortest path problem, but the latter name is standard, and so we will use it.

Throughout the book, we will use n to denote the number of vertices in the graph (that is, $n = |V|$) and m to denote the number of arcs or edges (that is, $m = |A|$).

The shortest path problem is, in some sense, the simplest possible type of flow problem involving costs. Usually flow problems specify a capacity for each arc, capping the rate at which flow can enter the arc. Here we have an *uncapacitated* problem; there are no capacities, and we can send as much flow on an arc as we want. Then if we want to send $a(i)$ units of flow from s to i, we compute the cheapest path from s to i, and the cost of shipping on this path is $a(i)d(i)$.

In what follows, we first discuss an algorithm we can use when all the arc costs are nonnegative. Then we give an algorithm that works when arc costs are negative (subject to the issue of negative-cost cycles).

1.1 Nonnegative Costs: Dijkstra's Algorithm

When arc costs $c(i,j)$ are nonnegative for all arcs $(i,j) \in A$, we can use *Dijkstra's algorithm*, due to Dijkstra [51]. The algorithm maintains a *distance labeling* d on the vertices of the graph; the label $d(i)$ is the algorithm's current guess of the cost of the cheapest path from s to i. As discussed above, we will refer to this as the distance from s to i; henceforward, we will fearlessly interchange the notions of cost and distance. We will maintain the property that the algorithm's guess $d(i)$ is always an upper bound on the true shortest-path distance from s to i. This notion of a distance labeling is one that will recur throughout our discussion of network flow algorithms.

We will also mark vertices as we become certain that their current distance label is correct. Initially, all vertices are unmarked.

Since the algorithm maintains a distance labeling that is an upper bound on the true distance, the easiest place to start is with $d(i) = \infty$ for all $i \in V$. Actually, this is overly pessimistic, because we can set the label of s to zero. Since all arc costs are nonnegative, there cannot be a path from s to s with cost less than zero, and the path with no arcs from s to s trivially has cost zero. Thus we can set $d(s) = 0$, and mark s, since we are certain this label is correct.

What now? Well, we can update the labels for all vertices i such that there is an arc $(s,i) \in A$. Since we know the length of the shortest path from s to s is zero, we know that there exists a path of length at most $d(s) + c(s,i) = c(s,i)$ from s to i (namely, the path consisting of the single arc (s,i)). So $d(s)+c(s,i)$

is a legitimate upper bound on the length of the shortest path from s to i, and we can set $d(i) = \min(d(i), d(s) + c(s,i))$ for i such that $(s,i) \in A$. This update will maintain the property that $d(i)$ is an upper bound on the shortest s-i path.

The key insight for Dijkstra's algorithm is that of all unmarked vertices, we can now correctly mark the one with the minimum distance label; if there is more than one vertex of minimum distance label, then we can choose one arbitrarily. We will prove that this is correct in a moment. Suppose vertex i has minimum distance label $d(i)$ and we mark vertex i. Then as above, for all arcs (i,j), we know that there is a path of length at most $d(i) + c(i,j)$ to vertex j (consisting of the shortest s-i path followed by the arc (i,j)). Thus we can update $d(j) = \min(d(j), d(i) + c(i,j))$ for all j such that $(i,j) \in A$. We then mark the unmarked vertex of minimum distance label, and iterate. Observe that each distance label can only decrease throughout the course of the algorithm.

In addition, by the preceding discussion, we know that the path to vertex j consists of a path from s to some vertex i, followed by the arc (i,j). Hence we can keep track of the current path to j by maintaining a pointer $p(j)$ to the vertex i preceding it on the path; we will call $p(j)$ the *parent* of j. When we update $d(j)$, if we set $d(j) = d(i) + c(i,j)$, we also set $p(j) = i$, so that we know that the current path from s to j is the arc (i,j) added to current path from s to i. To find the path from s to j, we start at j and trace the parent pointers from j back to s. For simplicity, we set the parent of s to be null.

We summarize Dijkstra's algorithm in Algorithm 1.1, and we prove its correctness below.

```
d(i) ← ∞ for all i ∈ V
p(i) ← null for all i ∈ V
Unmark all i ∈ V
d(s) ← 0
while not all vertices are marked do
    Find unmarked i ∈ V that minimizes d(i) and mark i
    for j such that (i, j) ∈ A do
        if d(j) > d(i) + c(i, j) then
            d(j) ← d(i) + c(i, j)
            p(j) ← i
```

Algorithm 1.1 Dijkstra's algorithm for the shortest path problem.

Theorem 1.1: *If all arc costs are nonnegative, then Dijkstra's algorithm (Algorithm 1.1) correctly determines the shortest distance from the source s to each vertex $i \in V$.*

Proof We argue by induction on the algorithm that when the algorithm marks a vertex j, the value $d(j)$ must be the length of the shortest path from s to j. We have argued previously that $d(s) = 0$, and clearly s is the first vertex marked by the algorithm. Now suppose some iteration of the algorithm is about to mark vertex $j \neq s$, and the algorithm has correctly computed $d(i)$ for all vertices i previously marked by the algorithm. We recall that $d(j)$ is an upper bound on the length of the shortest s-j path, so $d(j)$ is incorrect only if there is a shortest s-j path P that has length strictly less than $d(j)$. Assume that such a path P exists; we will show that we reach a contradiction. We follow path P from s to j until we reach the last vertex $i \neq j$ on the path that was marked; there will be some such vertex because s has already been marked and j is not marked. Let (i,k) be the arc out of i on path P, with k not marked; note that possibly $k = j$. (See Figure 1.1.) By our induction hypothesis, since i has been marked, $d(i)$ is the length of the shortest path from s to i. After we marked i, it must have been the case that $d(k) \leq d(i) + c(i,k)$: either this was already true or we set $d(k) = d(i) + c(i,k)$ after marking i. The length of the remainder of the path P from k to j must be nonnegative, because all the arc costs are nonnegative. Thus $d(k)$ is a lower bound on the length of the path P, which is strictly less than $d(j)$ by assumption. However, we have now reached a contradiction because k is unmarked and has a distance label strictly less than $d(j)$: if $k = j$, then we have $d(j) < d(j)$, or if $k \neq j$, another unmarked vertex has minimum distance label rather than j. □

It is easy to see that we can implement the algorithm in $O(m+n^2) = O(n^2)$ time by looking for the unmarked vertex of minimum label in each step (recall that $m = |A|$ is the number of arcs in the graph and $n = |V|$ is the number of vertices). Observe that we consider each arc (i, j) in the graph exactly once, when the tail i of the arc is first marked.

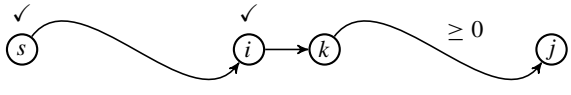

Figure 1.1 Illustration of the proof of Theorem 1.1; this is the shortest s-j path P of length strictly less than $d(j)$. The node i is the last marked node on the path, and k is the next node on the path, and must be unmarked. The proof argues that then $d(k)$ is a lower bound on the length of the path, and thus some node other than j should have been the next to be selected and marked.

```
h ←new heap();
d(i) ← ∞ for all i ∈ V
p(i) ← null for all i ∈ V
d(s) ← 0
for all i ∈ V do
    h.insert(i, d(i))
while not h.empty? do
    i ←h.extract-min()
    for j such that (i, j) ∈ A do
        if d(j) > d(i) + c(i, j) then
            d(j) ← d(i) + c(i, j)
            p(j) ← i
            h.decrease-key(j, d(j))
```

Algorithm 1.2 Dijkstra's algorithm for the shortest path problem using a heap data structure.

We can get a better asymptotic running time by using a data structure known as a *heap*. A heap contains a set of items, and each item has an associated value called its *key*. A heap data structure supports the following operations: *new heap()*, which returns an empty heap; *h.insert(i,k)*, which inserts item i into heap h with key value k; *h.decrease-key(i,k′)*, which decreases the key of i to k' (it is assumed that k' is no greater than the current key of item i); *h.extract-min()*, which returns an item i of minimum key value in heap h and removes i from the heap; and *h.empty?*, which returns true if the heap h has no items in it, and returns false otherwise. We can then rewrite Algorithm 1.1 in terms of these operations, which we do in Algorithm 1.2. The items in the heap are the vertices and their keys are the distance labels. Notice that we replace the marking of nodes with non-membership in the heap; if the node is in the heap, then it is unmarked.

Heaps are easy to implement using arrays; we do not give the details here, but point the interested reader to standard books on algorithms (see Chapter Notes for references). The most straightforward implementation of a heap data structure takes $O(1)$ time for a *new heap*, $O(\log n)$ time for an *insert* (given that we are inserting at most n items), $O(\log n)$ time for a *decrease-key*, $O(\log n)$ time for an *extract-min*, and $O(1)$ time for *empty?*. These running times for the data structure yield an overall running time of $O(m \log n)$ time for Dijkstra's algorithm, since we perform n *extract-min*s, n *insert*s, and at most m *decrease-key*s. Faster theoretical running times are known: Using a

data structure called a *Fibonacci heap*, it is possible to implement Dijkstra's algorithm in $O(m + n \log n)$ time. See Chapter Notes for more details.

1.2 Negative Costs: The Bellman–Ford Algorithm

We now turn to the case in which the cost of an arc may be negative. While it is difficult to think of instances of problems involving physical travel on networks in which there are arcs of negative length, it is often useful when modeling problems to allow for negative costs; we will encounter this situation many times in the flow algorithms to come.

Once we allow for negative-cost arcs, however, we have to contend with the possibility that there might not be an s-i path of shortest overall length: for any bound B, there might be a path of length less than B. See Figure 1.2 for an example: the s-t path (s,a), (a,t) has cost 2, the path (s,a), (a,b), (b,c), (c,a), (a,t) has cost 1, the path (s,a), (a,b), (b,c), (c,a), (a,b), (b,c), (c,a), (a,t) has cost 0, and so on. Each time we traverse the cycle a-b-c the cost drops by 1. In order to prevent this possibility, we request that our algorithm for the shortest path problem either finds a shortest path or states that it cannot do so because there is a *negative-cost cycle* reachable from s. A cycle has negative cost if the sum of the costs of the arcs in the cycle is negative, while a vertex i is *reachable* from s if there is a path from s to i, and a cycle is reachable from s if any vertex on the cycle is reachable from s. We leave it as an exercise to the reader (Exercise 1.2) to show that there are simple shortest paths from s to each $i \in V$ reachable from s if and only if there are no negative-cost cycles reachable from s (recall that in a simple path, no vertex in the path is repeated). In order to simplify our discussion somewhat, we start by assuming that there are no negative-cost cycles in the input graph, and we then show how to detect them in the next section.

As in Dijkstra's algorithm, this algorithm will maintain a set of distance labels $d(i)$ for all $i \in V$, where initially $d(s) = 0$ and $d(i) = \infty$ for all $i \in V$,

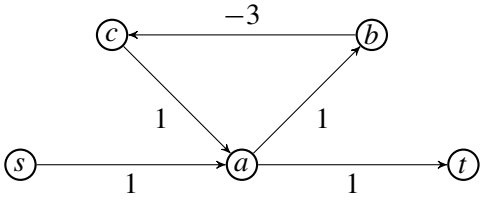

Figure 1.2 A negative-cost cycle on the path from s from t.

```
d(i) ← ∞ for all i ∈ V
p(i) ← null for all i ∈ V
d(s) ← 0
for k ← 1 to n − 1 do
    for all (i, j) ∈ A do
        if d(j) > d(i) + c(i, j) then
            d(j) ← d(i) + c(i, j)
            p(j) ← i
```

Algorithm 1.3 The Bellman–Ford algorithm for the shortest path problem.

$i \neq s$. The algorithm will have the property that whenever $d(i)$ is finite, it is always the length of *some* path from s to i. Here the central insight is that given an arc (i, j), whenever $d(j) > d(i) + c(i, j)$, we can set $d(j) = d(i) + c(i, j)$; by our invariant, there is some path to i of length $d(i)$, and so we can find a shorter path to j that first visits i and then uses arc (i, j) to get to j. As was the case for Dijkstra's algorithm, we also maintain a set of parent pointers $p(j)$ that point to the previous vertex on the current path to j. Thus when we set $d(j) = d(i) + c(i, j)$, we know that the path to j came from vertex i, and we set $p(j) = i$. Again we observe that distance labels only decrease during the course of the algorithm.

The main idea of the analysis is to show that, after checking all arcs k times, we have correctly found all shortest paths that use at most k arcs. Thus, assuming there are no negative-cost cycles, the algorithm can terminate after $n - 1$ iterations through all the arcs, since any shortest s-i path is simple and will use at most $n - 1$ arcs. This algorithm is traditionally attributed jointly to Bellman and Ford [18, 62], although it was also discovered by others at around the same time; see Chapter Notes for a discussion. We summarize the Bellman–Ford algorithm in Algorithm 1.3. The algorithm will not work correctly if the graph contains negative-cost cycles, but we will first analyze it, and then see how to modify it in order to detect negative-cost cycles.

Lemma 1.2: *Any finite distance label $d(i)$ is the length of some s-i path in the network.*

Proof The lemma follows easily by induction on the algorithm. At the start of the algorithm, the only finite distance label is $d(s) = 0$, which is the length of the s-s path of zero arcs. Then whenever we update a distance label $d(j)$, we set $d(j) = d(i) + c(i, j)$, so that $d(j)$ is the length of the s-i path of length $d(i)$ (which exists by induction) plus the arc (i, j). □

Lemma 1.3: *After k iterations of the Bellman–Ford algorithm (Algorithm 1.3), each distance label $d(i)$ is at most the length of the shortest s-i path that uses at most k arcs.*

Proof The base case is simple: at the start of the algorithm (at the end of the 0th iteration), there is a path from s to s of length 0; this is the shortest path from s to s with at most 0 arcs. Now for the inductive case. Consider a shortest s-j path P that uses at most k arcs (if one exists), in which the last arc is (i, j). Then the subpath of P from s to i of at most $k-1$ arcs must be a shortest s-i path that uses at most $k-1$ arcs, since if there is a shorter s-i path using at most $k-1$ arcs, then it could be prepended to the arc (i, j) to obtain a shorter s-j path than P using at most k arcs. By the induction hypothesis, after $k-1$ iterations, $d(i)$ is at most the length of this shortest s-i path of at most $k-1$ arcs. After the kth iteration, we have updated $d(j)$ to be at most $d(i) + c(i, j)$, so that $d(j)$ is at most the length of path P, as desired. □

Theorem 1.4: *If there is no negative-cost cycle reachable from s, then the Bellman–Ford algorithm (Algorithm 1.3) correctly determines the length $d(i)$ of the shortest path from the source s to each $i \in V$ if one exists.*

Proof If there are no negative-cost cycles reachable from s, then by Exercise 1.2 for each i the shortest s-i path must be simple and have at most $n-1$ arcs. Thus by Lemmas 1.2 and 1.3, at the termination of the algorithm, $d(i)$ is the length of a shortest s-i path. □

Note that we have not yet proven that the parent pointers p give the shortest path in the network. Although they do give the shortest paths, it will be easier to prove this statement once we have considered the issue of negative-cost cycles in the following section; see Corollary 1.13.

Algorithm 1.3 clearly runs in $O(mn)$ time; in fact, we examine each arc exactly $n-1$ times, and thus the algorithm takes $\Theta(mn)$ time. We can ensure that it is possible for the algorithm to take fewer than $m(n-1)$ operations by observing that we often don't need to check whether $d(j) > d(i) + c(i, j)$; if $d(i)$ was not decreased in the previous iteration of the algorithm, then none of the arcs (i, j) out of i will lead to a decrease of $d(j)$ in the current iteration. As a step toward making this clearer, it will help to introduce an additional abstraction to the Bellman–Ford algorithm; the abstraction is that of a *scan*. A scan of a vertex i checks all the outgoing arcs (i, j) of i to see whether $d(j) > d(i) + c(i, j)$, and if so, performs the appropriate update; see the Procedure Scan. We rewrite Algorithm 1.3 in terms of scans in Algorithm 1.4. As an exercise, the reader can check that Dijkstra's algorithm can also be rewritten in terms of scans so that we scan a vertex precisely when we mark it.

```
d(i) ← ∞ for all i ∈ V
p(i) ← null for all i ∈ V
d(s) ← 0
for k ← 1 to n − 1 do
    for all i ∈ V do
        Scan(i)
```

Algorithm 1.4 The Bellman–Ford algorithm using scans.

```
for j such that (i, j) ∈ A do
    if d(j) > d(i) + c(i, j) then
        d(j) ← d(i) + c(i, j)
        p(j) ← i
```

Procedure Scan(i)

Now we note that we only need to scan a vertex i in an iteration if its distance label $d(i)$ was decreased in the previous iteration; if $d(i)$ was unchanged in the previous iteration, then for all arcs (i, j) coming out of i, $d(j)$ will remain at most $d(i) + c(i, j)$. To implement this idea, we use a *queue* data structure. A queue is an ordered list of items and implements the following operations: *new queue()*, which returns an empty queue; *q.add(i)*, which adds an item i to the end of the queue q; *q.remove()*, which removes the item from the front of the queue q and returns it (if there is such an item); *q.empty?*, which checks if the queue q contains any items; and *q.contains?(i)*, which checks if the queue q already contains item i. We assume that *q.contains?* is implemented in $O(1)$ time rather than the $O(n)$ time it would take to scan the queue to check for membership; we can implement the operation this way with an array, for example, when we know in advance, as we do in this case, what elements the queue might contain.

We can then rewrite the scan procedure and the Bellman–Ford algorithm as shown in Procedure QScan and Algorithm 1.5. We place a vertex j in the queue when its distance label has changed during a scan; if vertex j is not in the queue, its label has not changed, and we do not need to perform a scan on it.

We can prove that the algorithm works correctly by an inductive argument similar to that in the proof of Theorem 1.4 in which we replace induction on iterations with induction on passes over the queue.

```
d(i) ← ∞ for all i ∈ V
p(i) ← null for all i ∈ V
d(s) ← 0
q ←new queue()
q.add(s)
while not q.empty? do
    QScan(q.remove(), q);
```

Algorithm 1.5 The Bellman–Ford algorithm using queues.

```
for j such that (i, j) ∈ A do
    if d(j) > d(i) + c(i, j) then
        d(j) ← d(i) + c(i, j)
        p(j) ← i
        if not q.contains?(j) then
            q.add(j)
```

Procedure QScan(i, q)

Theorem 1.5: *If there are no negative-cost cycles reachable from s, then Algorithm 1.5 correctly determines the length $d(i)$ of the shortest path from the source s to each $i \in V$ if one exists.*

Proof As suggested above, we apply induction on passes over the queue. Pass 0 ends after s is initially added to the queue, pass 1 ends after the initial scan of s, and in general pass k ends after the scans of all vertices added to the queue in pass $k-1$. The induction hypothesis is that at the end of the kth pass, $d(i)$ is at most the length of a shortest s-i path of at most k arcs, and the proof proceeds as in the proof of Theorem 1.4.

If there are no negative-cost cycles reachable from s, then the shortest s-i path can have at most $n-1$ arcs in it, and thus by the end of the $(n-1)$st pass, the value of $d(i)$ will be the length of this path. Also, since the $d(i)$ are the lengths of the shortest paths from s to i, $d(j) \leq d(i) + c(i, j)$ for all $(i, j) \in A$ with i reachable from s, since otherwise there would be a shorter path from s to j. Thus no further vertices will be added to the queue, and the algorithm will terminate. Since there are at most $n-1$ passes, and each pass considers each vertex at most once, at most all m arcs are considered in each pass. Thus the running time of the algorithm is $O(mn)$. □