This is a useful expression, since it's something that we solved in our earlier discussion of recurrences at the outset of Chapter 5. Specifically, this recurrence implies $T(n) = O(n^2)$.

So when $m = n$ and we get an even split, the running time grows like the square of n. Motivated by this, we move back to the fully general recurrence for the problem at hand and guess that $T(m, n)$ grows like the product of m and n. Specifically, we'll guess that $T(m, n) \leq kmn$ for some constant k, and see if we can prove this by induction. To start with the base cases $m \leq 2$ and $n \leq 2$, we see that these hold as long as $k \geq c/2$. Now, assuming $T(m', n') \leq km'n'$ holds for pairs (m', n') with a smaller product, we have

$$\begin{aligned} T(m, n) &\leq cmn + T(q, n/2) + T(m - q, n/2) \\ &\leq cmn + kqn/2 + k(m - q)n/2 \\ &= cmn + kqn/2 + kmn/2 - kqn/2 \\ &= (c + k/2)mn. \end{aligned}$$

Thus the inductive step will work if we choose $k = 2c$, and this completes the proof. ■

6.8 Shortest Paths in a Graph

For the final three sections, we focus on the problem of finding shortest paths in a graph, together with some closely related issues.

The Problem

Let $G = (V, E)$ be a directed graph. Assume that each edge $(i, j) \in E$ has an associated *weight* c_{ij}. The weights can be used to model a number of different things; we will picture here the interpretation in which the weight c_{ij} represents a *cost* for going directly from node i to node j in the graph.

Earlier we discussed Dijkstra's algorithm for finding shortest paths in graphs with positive edge costs. Here we consider the more complex problem in which we seek shortest paths when costs may be negative. Among the motivations for studying this problem, here are two that particularly stand out. First, negative costs turn out to be crucial for modeling a number of phenomena with shortest paths. For example, the nodes may represent agents in a financial setting, and c_{ij} represents the cost of a transaction in which we buy from agent i and then immediately sell to agent j. In this case, a path would represent a succession of transactions, and edges with negative costs would represent transactions that result in profits. Second, the algorithm that we develop for dealing with edges of negative cost turns out, in certain crucial ways, to be more flexible and *decentralized* than Dijkstra's algorithm. As a consequence, it has important applications for the design of distributed

routing algorithms that determine the most efficient path in a communication network.

In this section and the next two, we will consider the following two related problems.

- Given a graph G with weights, as described above, decide if G has a negative cycle—that is, a directed cycle C such that
$$\sum_{ij \in C} c_{ij} < 0.$$
- If the graph has no negative cycles, find a path P from an origin node s to a destination node t with minimum total cost:
$$\sum_{ij \in P} c_{ij}$$
should be as small as possible for any s-t path. This is generally called both the *minimum-cost path problem* and the *shortest-path problem*.

In terms of our financial motivation above, a negative cycle corresponds to a profitable sequence of transactions that takes us back to our starting point: we buy from i_1, sell to i_2, buy from i_2, sell to i_3, and so forth, finally arriving back at i_1 with a net profit. Thus negative cycles in such a network can be viewed as good *arbitrage opportunities*.

It makes sense to consider the minimum-cost s-t path problem under the assumption that there are no negative cycles. As illustrated by Figure 6.20, if there is a negative cycle C, a path P_s from s to the cycle, and another path P_t from the cycle to t, then we can build an s-t path of arbitrarily negative cost: we first use P_s to get to the negative cycle C, then we go around C as many times as we want, and then we use P_t to get from C to the destination t.

Designing and Analyzing the Algorithm

A Few False Starts. Let's begin by recalling Dijkstra's algorithm for the shortest, path problem when there are no negative costs. That method computes

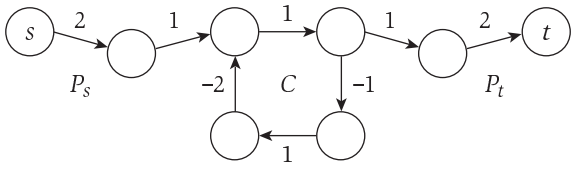
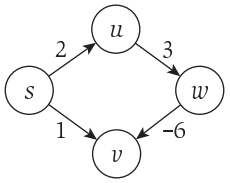
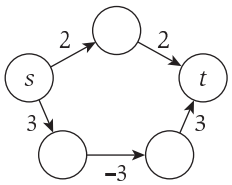


Figure 6.20 In this graph, one can find s-t paths of arbitrarily negative cost (by going around the cycle C many times).



(a)



(b)

Figure 6.21 (a) With negative edge costs, Dijkstra's algorithm can give the wrong answer for the shortest-path problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest s-t path.

a shortest path from the origin s to every other node v in the graph, essentially using a greedy algorithm. The basic idea is to maintain a set S with the property that the shortest path from s to each node in S is known. We start with $S = \{s\}$—since we know the shortest path from s to s has cost 0 when there are no negative edges—and we add elements greedily to this set S. As our first greedy step, we consider the minimum-cost edge leaving node s, that is, $\min_{i \in V} c_{si}$. Let v be a node on which this minimum is obtained. A key observation underlying Dijkstra's algorithm is that the shortest path from s to v is the single-edge path $\{s, v\}$. Thus we can immediately add the node v to the set S. The path $\{s, v\}$ is clearly the shortest to v if there are no negative edge costs: any other path from s to v would have to start on an edge out of s that is at least as expensive as edge (s, v).

The above observation is no longer true if we can have negative edge costs. As suggested by the example in Figure 6.21(a), a path that starts on an expensive edge, but then compensates with subsequent edges of negative cost, can be cheaper than a path that starts on a cheap edge. This suggests that the Dijkstra-style greedy approach will not work here.

Another natural idea is to first modify the costs c_{ij} by adding some large constant M to each; that is, we let $c'_{ij} = c_{ij} + M$ for each edge $(i, j) \in E$. If the constant M is large enough, then all modified costs are nonnegative, and we can use Dijkstra's algorithm to find the minimum-cost path subject to costs c'. However, this approach fails to find the correct minimum-cost paths with respect to the original costs c. The problem here is that changing the costs from c to c' changes the minimum-cost path. For example (as in Figure 6.21(b)), if a path P consisting of three edges is only slightly cheaper than another path P' that has two edges, then after the change in costs, P' will be cheaper, since we only add $2M$ to the cost of P' while adding $3M$ to the cost of P.

A Dynamic Programming Approach We will try to use dynamic programming to solve the problem of finding a shortest path from s to t when there are negative edge costs but no negative cycles. We could try an idea that has worked for us so far: subproblem i could be to find a shortest path using only the first i nodes. This idea does not immediately work, but it can be made to work with some effort. Here, however, we will discuss a simpler and more efficient solution, the *Bellman-Ford algorithm*. The development of dynamic programming as a general algorithmic technique is often credited to the work of Bellman in the 1950's; and the Bellman-Ford shortest-path algorithm was one of the first applications.

The dynamic programming solution we develop will be based on the following crucial observation.

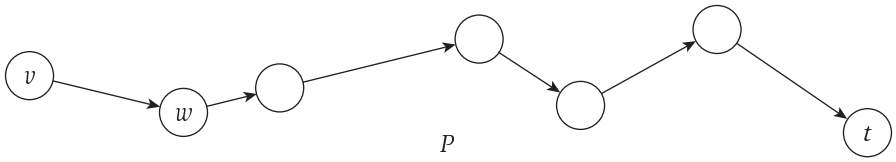


Figure 6.22 The minimum-cost path P from v to t using at most i edges.

(6.22) *If G has no negative cycles, then there is a shortest path from s to t that is simple (i.e. does not repeat nodes), and hence has at most $n-1$ edges.*

Proof. Since every cycle has nonnegative cost, the shortest path $\mathcal{P}$ from s to t with the fewest number of edges does not repeat any vertex v. For if $\mathcal{P}$ did repeat a vertex v, we could remove the portion of $\mathcal{P}$ between consecutive visits to v, resulting in a path of no greater cost and fewer edges. ■

Let's use $\text{OPT}(i, v)$ to denote the minimum cost of a v-t path using at most i edges. By (6.22), our original problem is to compute $\text{OPT}(n-1, s)$. (We could instead design an algorithm whose subproblems correspond to the minimum cost of an s-v path using at most i edges. This would form a more natural parallel with Dijkstra's algorithm, but it would not be as natural in the context of the routing protocols we discuss later.)

We now need a simple way to express $\text{OPT}(i, v)$ using smaller subproblems. We will see that the most natural approach involves the consideration of many different options; this is another example of the principle of "multi-way choices" that we saw in the algorithm for the Segmented Least Squares Problem.

Let's fix an optimal path P representing $\text{OPT}(i, v)$ as depicted in Figure 6.22.

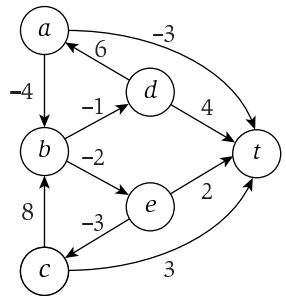
- If the path P uses at most $i-1$ edges, then we have $\text{OPT}(i, v) = \text{OPT}(i-1, v)$.
- If the path P uses i edges, and the first edge is (v, w), then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i-1, w)$.

This leads to the following recursive formula.

(6.23) *If $i > 0$ then*

$$\text{OPT}(i, v) = \min(\text{OPT}(i-1, v), \min_{w \in V}(\text{OPT}(i-1, w) + c_{vw})).$$

Using this recurrence, we get the following dynamic programming algorithm to compute the value $\text{OPT}(n-1, t)$.



(a)

	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	−3	−3	−4	−6	−6
b	∞	∞	0	−2	−2	−2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

(b)

Figure 6.23 For the directed graph in (a), shortest-path algorithm constructs the dynamic programming table in (b).

```
Shortest-Path(G, s, t)
  n = number of nodes in G
  Array M[0 ... n − 1, V]
  Define M[0, t] = 0 and M[0, v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For v ∈ V in any order
      Compute M[i, v] using the recurrence (6.23)
    Endfor
  Endfor
  Return M[n − 1, s]
```

The correctness of the method follows directly by induction from (6.23). We can bound the running time as follows. The table M has n^2 entries; and each entry can take $O(n)$ time to compute, as there are at most n nodes $w \in V$ we have to consider.

(6.24) *The* `Shortest-Path` *method correctly computes the minimum cost of an s-t path in any graph that has no negative cycles, and runs in* $O(n^3)$ *time.*

Given the table M containing the optimal values of the subproblems, the shortest path using at most i edges can be obtained in $O(in)$ time, by tracing back through smaller subproblems.

As an example, consider the graph in Figure 6.23, where the goal is to find a shortest path from each node to t. The table on the right shows the array M, with entries corresponding to the values $M[i, v]$ from the algorithm. Thus a single row in the table corresponds to the shortest path from a particular node to t, as we allow the path to use an increasing number of edges. For example, the shortest path from node d to t is updated four times, as it changes from d-t, to d-a-t, to d-a-b-e-t, and finally to d-a-b-e-c-t.

Extensions: Some Basic Improvements to the Algorithm

We can actually provide a better running-time analysis for the case in which the graph G does not have too many edges. A directed graph with n nodes can have close to n^2 edges, since there could potentially be an edge between each pair of nodes, but many graphs are much sparser than this. When we work with a graph for which the number of edges m is significantly less than n^2, we've already seen in a number of cases earlier in the book that it can be useful to write the running-time in terms of both m and n; this way, we can quantify our speed-up on graphs with relatively fewer edges.

If we are a little more careful in the analysis of the method above, we can improve the running time bound to $O(mn)$ without significantly changing the algorithm itself.

(6.25) *The* `Shortest-Path` *method can be implemented in* $O(mn)$ *time.*

Proof.

Consider the computation of the array entry $M[i, v]$ according to the recurrence (6.23); we have

$$M[i, v] = \min(M[i-1, v], \min_{w \in V}(M[i-1, w] + c_{vw})).$$

We assumed it could take up to $O(n)$ time to compute this minimum, since there are n possible nodes w. But, of course, we need only compute this minimum over all nodes w for which v has an edge to w; let us use n_v to denote this number. Then it takes time $O(n_v)$ to compute the array entry $M[i, v]$. We have to compute an entry for every node v and every index $0 \leq i \leq n-1$, so this gives a running-time bound of $O\left(n \sum_{v \in V} n_v\right)$.

In Chapter 3, we performed exactly this kind of analysis for other graph algorithms, and used (3.9) from that chapter to bound the expression $\sum_{v \in V} n_v$ for undirected graphs. Here we are dealing with directed graphs, and n_v denotes the number of edges *leaving* v. In a sense, it is even easier to work out the value of $\sum_{v \in V} n_v$ for the directed case: each edge leaves exactly one of the nodes in V, and so each edge is counted exactly once by this expression. Thus we have $\sum_{v \in V} n_v = m$.

Plugging this into our expression $O\left(n \sum_{v \in V} n_v\right)$ for the running-time, we get a running time bound of $O(mn)$. ■

We can also significantly improve the memory requirements with only a small change to the implementation. A common problem with many dynamic programming algorithms is the large space usage, arising from the M array that needs to be stored. In the Bellman-Ford algorithm as written, this array has size n^2; however, we now show how to reduce this to $O(n)$. Rather than recording $M[i, v]$ for each value i, we will use and update a single value $M[v]$ for each node v, the length of the shortest path from v to t that we have found so far. We still run the algorithm for iterations $i = 1, 2, \ldots, n-1$, but the role of i will now simply be as a counter; in each iteration, and for each node v, we perform the update

$$M[v] = \min(M[v], \min_{w \in V}(c_{vw} + M[w])).$$

We now observe the following fact.

(6.26) *Throughout the algorithm $M[v]$ is the length of some path from v to t, and after i rounds of updates the value $M[v]$ is no larger than the length of the shortest path from v to t using at most i edges.*

Given (6.26), we can then use (6.22) as before to show that we are done after $n - 1$ iterations. Since we are only storing an M array that indexes over the nodes, this requires only $O(n)$ working memory. In fact, with a little more work, the shortest paths themselves can be recovered with $O(n)$ memory as well.

Note that in the more space-efficient version of Bellman-Ford, the path whose length is $M[v]$ after i iterations can have substantially more edges than i. For example, if the graph is a single path from s to t, and we perform updates in the order the edges appear on the path, then we get the final shortest-path values in just one iteration. This does not always happen, so we cannot claim a worst-case running-time improvement, but it would be nice to be able to use this fact opportunistically to speed up the algorithm on instances where it does happen. In order to do this, we need a stopping signal in the algorithm—something that tells us it's safe to terminate before iteration $n - 1$ is reached.

Such a stopping signal is a simple consequence of the following observation: If we ever execute a complete iteration i in which *no* $M[v]$ value changes, then no $M[v]$ value will ever change again, since future iterations will begin with exactly the same set of array entries. Thus it is safe to stop the algorithm. Note that it is not enough for a *particular* $M[v]$ value to remain the same; in order to safely terminate, we need for all these values to remain the same for a single iteration.

6.9 Shortest Paths and Distance Vector Protocols

One important application of the shortest-path problem is for routers in a communication network to determine the most efficient path to a destination. We represent the network using a graph in which the nodes correspond to routers, and there is an edge between v and w if the two routers are connected by a direct communication link. We define a cost c_{vw} representing the delay on the link (v, w); the shortest-path problem with these costs is to determine the path with minimum delay from a source node s to a destination t. Delays are naturally nonnegative, so one could use Dijkstra's algorithm to compute the shortest path. However, Dijkstra's shortest-path computation requires global knowledge of the network: it needs to maintain a set S of nodes for which shortest paths have been determined, and make a global decision about which node to add next to S. While routers can be made to run a protocol in the background that gathers enough global information to implement such an

algorithm, it is often cleaner and more flexible to use algorithms that require only local knowledge of neighboring nodes.

If we think about it, the Bellman-Ford algorithm discussed in the previous section has just such a "local" property. Suppose we let each node v maintain its value $M[v]$; then to update this value, v needs only obtain the value $M[w]$ from each neighbor w, and compute

$$\min_{w \in V}(c_{vw} + M[w])$$

based on the information obtained.

We now discuss an improvement to the Bellman-Ford algorithm that makes it better suited for routers and, at the same time, a faster algorithm in practice. Our current implementation of the Bellman-Ford algorithm can be thought of as a *pull-based* algorithm. In each iteration i, each node v has to contact each neighbor w, and "pull" the new value $M[w]$ from it. If a node w has not changed its value, then there is no need for v to get the value again; however, v has no way of knowing this fact, and so it must execute the pull anyway.

This wastefulness suggests a symmetric *push-based* implementation, where values are only transmitted when they change. Specifically, each node w whose distance value $M[w]$ changes in an iteration informs all its neighbors of the new value in the next iteration; this allows them to update their values accordingly. If $M[w]$ has not changed, then the neighbors of w already have the current value, and there is no need to "push" it to them again. This leads to savings in the running time, as not all values need to be pushed in each iteration. We also may terminate the algorithm early, if no value changes during an iteration. Here is a concrete description of the push-based implementation.

```
Push-Based-Shortest-Path(G, s, t):
  n = number of nodes in G
  Array M[V]
  Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V
  For i = 1, ..., n − 1
    For w ∈ V in any order
      If M[w] has been updated in the previous iteration then
        For all edges (v, w) in any order
          M[v] = min(M[v], c_vw + M[w])
        Endfor
    Endfor
    If no value changed in this iteration, then end the algorithm
  Endfor
  Return M[s]
```

In this algorithm, nodes are sent updates of their neighbors' distance values in rounds, and each node sends out an update in each iteration in which it has changed. However, if the nodes correspond to routers in a network, then we do not expect everything to run in lock-step like this; some routers may report updates much more quickly than others, and a router with an update to report may sometimes experience a delay before contacting its neighbors. Thus the routers will end up executing an *asynchronous* version of the algorithm: each time a node w experiences an update to its $M[w]$ value, it becomes "active" and eventually notifies its neighbors of the new value. If we were to watch the behavior of all routers interleaved, it would look as follows.

```
Asynchronous-Shortest-Path(G, s, t):
  n = number of nodes in G
  Array M[V]
  Initialize M[t] = 0 and M[v] = ∞ for all other v ∈ V
  Declare t to be active and all other nodes inactive
  While there exists an active node
    Choose an active node w
      For all edges (v, w) in any order
        M[v] = min(M[v], c_vw + M[w])
        If this changes the value of M[v], then
          v becomes active
      Endfor
      w becomes inactive
  EndWhile
```

One can show that even this version of the algorithm, with essentially no coordination in the ordering of updates, will converge to the correct values of the shortest-path distances to t, assuming only that each time a node becomes active, it eventually contacts its neighbors.

The algorithm we have developed here uses a single destination t, and all nodes $v \in V$ compute their shortest path to t. More generally, we are presumably interested in finding distances and shortest paths between all pairs of nodes in a graph. To obtain such distances, we effectively use n separate computations, one for each destination. Such an algorithm is referred to as a *distance vector protocol*, since each node maintains a vector of distances to every other node in the network.

Problems with the Distance Vector Protocol

One of the major problems with the distributed implementation of Bellman-Ford on routers, (the protocol we have been discussing above) is that it's derived from an initial dynamic programming algorithm that assumes edge

costs will remain constant during the execution of the algorithm. Thus far we've been designing algorithms with the tacit understanding that a program executing the algorithm will be running on a single computer (or a centrally managed set of computers), processing some specified input. In this context, it's a rather benign assumption to require that the input not change while the program is actually running. Once we start thinking about routers in a network, however, this assumption becomes troublesome. Edge costs may change for all sorts of reasons: links can become congested and experience slow-downs; or a link (v, w) may even fail, in which case the cost c_{vw} effectively increases to ∞.

Here's an indication of what can go wrong with our shortest-path algorithm when this happens. If an edge (v, w) is deleted (say the link goes down), it is natural for node v to react as follows: it should check whether its shortest path to some node t used the edge (v, w), and, if so, it should increase the distance using other neighbors. Notice that this increase in distance from v can now trigger increases at v's neighbors, if they were relying on a path through v, and these changes can cascade through the network. Consider the extremely simple example in Figure 6.24, in which the original graph has three edges (s, v), (v, s) and (v, t), each of cost 1.

Now suppose the edge (v, t) in Figure 6.24 is deleted. How does node v react? Unfortunately, it does not have a global map of the network; it only knows the shortest-path distances of each of its neighbors to t. Thus it does not know that the deletion of (v, t) has eliminated all paths from s to t. Instead, it sees that $M[s] = 2$, and so it updates $M[v] = c_{vs} + M[s] = 3$—assuming that it will use its cost-1 edge to s, followed by the supposed cost-2 path from s to t. Seeing this change, node s will update $M[s] = c_{sv} + M[v] = 4$, based on its cost-1 edge to v, followed by the supposed cost-3 path from v to t. Nodes s and v will continue updating their distance to t until one of them finds an alternate route; in the case, as here, that the network is truly disconnected, these updates will continue indefinitely—a behavior known as the problem of *counting to infinity*.

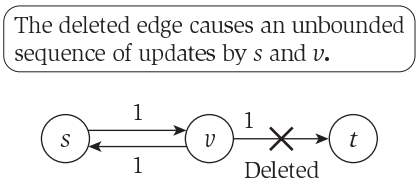


Figure 6.24 When the edge (v, t) is deleted, the distributed Bellman-Ford algorithm will begin "counting to infinity."

To avoid this problem and related difficulties arising from the limited amount of information available to nodes in the Bellman-Ford algorithm, the designers of network routing schemes have tended to move from distance vector protocols to more expressive *path vector protocols*, in which each node stores not just the distance and first hop of their path to a destination, but some representation of the entire path. Given knowledge of the paths, nodes can avoid updating their paths to use edges they know to be deleted; at the same time, they require significantly more storage to keep track of the full paths. In the history of the Internet, there has been a shift from distance vector protocols to path vector protocols; currently, the path vector approach is used in the *Border Gateway Protocol* (BGP) in the Internet core.

* 6.10 Negative Cycles in a Graph

So far in our consideration of the Bellman-Ford algorithm, we have assumed that the underlying graph has negative edge costs but no negative cycles. We now consider the more general case of a graph that may contain negative cycles.

The Problem

There are two natural questions we will consider.

- How do we decide if a graph contains a negative cycle?
- How do we actually construct a negative cycle in a graph that contains one?

The algorithm developed for finding negative cycles will also lead to an improved practical implementation of the Bellman-Ford algorithm from the previous sections.

It turns out that the ideas we've seen so far will allow us to find negative cycles that have a path reaching a sink t. Before we develop the details of this, let's compare the problem of finding a negative cycle that can reach a given t with the seemingly more natural problem of finding a negative cycle *anywhere* in the graph, regardless of its position related to a sink. It turns out that if we develop a solution to the first problem, we'll be able to obtain a solution to the second problem as well, in the following way. Suppose we start with a graph G, add a new node t to it, and connect each other node v in the graph to node t via an edge of cost 0, as shown on Figure 6.25. Let us call the new "augmented graph" G'.

(6.27) *The augmented graph G' has a negative cycle C such that there is a path from C to the sink t if and only if the original graph has a negative cycle.*