## 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

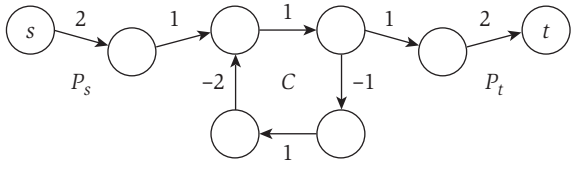
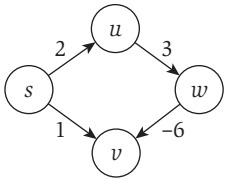
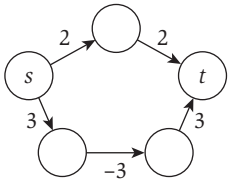



**Figure 6.20** In this graph, one can find $s$-$t$ paths of arbitrarily negative cost (by going around the cycle $C$ many times).




(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.

computes 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.

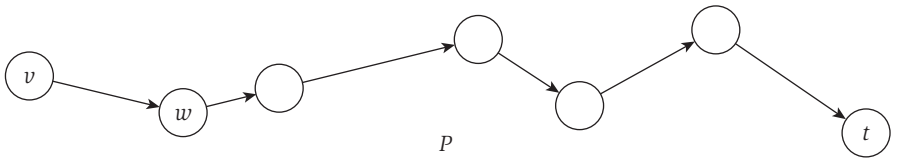



**Figure 6.22** The minimum-cost path $P$ from $v$ to $t$ using at most $i$ edges.

**(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 $P$ from $s$ to $t$ with the fewest number of edges does not repeat any vertex $v$. For if $P$ did repeat a vertex $v$, we could remove the portion of $P$ 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.

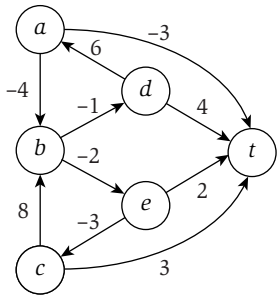
- If the path $P$ uses at most $i - 1$ edges, then $\text{OPT}(i, v) = \text{OPT}(i - 1, v)$.
- If the path $P$ uses $i$ edges, and the first edge is $(v, w)$, then $\text{OPT}(i, v) = c_{vw} + \text{OPT}(i - 1, w)$.

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 $\text{OPT}(n-1, s)$.




**(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), the 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(a), where the goal is to find a shortest path from each node to $t$. The table in Figure 6.23(b) 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

***An Improved Running-Time Analysis*** 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 $O(mn)$. ■

***Improving the Memory Requirements*** 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.

***Finding the Shortest Paths*** One issue to be concerned about is whether this space-efficient version of the algorithm saves enough information to recover the shortest paths themselves. In the case of the Sequence Alignment Problem in the previous section, we had to resort to a tricky divide-and-conquer method to recover the solution from a similar space-efficient implementation. Here, however, we will be able to recover the shortest paths much more easily.

To help with recovering the shortest paths, we will enhance the code by having each node $v$ maintain the first node (after itself) on its path to the destination $t$; we will denote this first node by $first[v]$. To maintain $first[v]$, we update its value whenever the distance $M[v]$ is updated. In other words, whenever the value of $M[v]$ is reset to the minimum $\min_{w \in V}(c_{vw} + M[w])$, we set $first[v]$ to the node $w$ that attains this minimum.

Now let $P$ denote the directed "pointer graph" whose nodes are $V$, and whose edges are $\{(v, first[v])\}$. The main observation is the following.

**(6.27)** *If the pointer graph $P$ contains a cycle $C$, then this cycle must have negative cost.*

**Proof.** Notice that if $first[v] = w$ at any time, then we must have $M[v] \geq c_{vw} + M[w]$. Indeed, the left- and right-hand sides are equal after the update that sets $first[v]$ equal to $w$; and since $M[w]$ may decrease, this equation may turn into an inequality.

Let $v_1, v_2, \ldots, v_k$ be the nodes along the cycle $C$ in the pointer graph, and assume that $(v_k, v_1)$ is the last edge to have been added. Now, consider the values right before this last update. At this time we have $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$ for all $i = 1, \ldots, k-1$, and we also have $M[v_k] > c_{v_k v_1} + M[v_1]$ since we are about to update $M[v_k]$ and change $first[v_k]$ to $v_1$. Adding all these inequalities, the $M[v_i]$ values cancel, and we get $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$: a negative cycle, as claimed. ■

Now note that if $G$ has no negative cycles, then (6.27) implies that the pointer graph $P$ will never have a cycle. For a node $v$, consider the path we get by following the edges in $P$, from $v$ to $first[v] = v_1$, to $first[v_1] = v_2$, and so forth. Since the pointer graph has no cycles, and the sink $t$ is the only node that has no outgoing edge, this path must lead to $t$. We claim that when the algorithm terminates, this is in fact a shortest path in $G$ from $v$ to $t$.

**(6.28)** *Suppose $G$ has no negative cycles, and consider the pointer graph $P$ at the termination of the algorithm. For each node $v$, the path in $P$ from $v$ to $t$ is a shortest $v$-$t$ path in $G$.*

**Proof.** Consider a node $v$ and let $w = first[v]$. Since the algorithm terminated, we must have $M[v] = c_{vw} + M[w]$. The value $M[t] = 0$, and hence the length of the path traced out by the pointer graph is exactly $M[v]$, which we know is the shortest-path distance. ■

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 reverse of 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])
        If this changes the value of M[v], then first[v] = 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 lockstep 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
          first[v] = w
          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.

## * 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 find 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

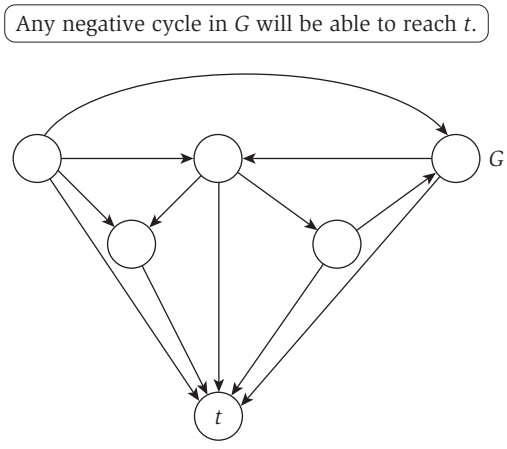



**Figure 6.25** The augmented graph.

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 in Figure 6.25. Let us call the new "augmented graph" $G'$.

**(6.29)** *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.*

**Proof.** Assume $G$ has a negative cycle. Then this cycle $C$ clearly has an edge to $t$ in $G'$, since all nodes have an edge to $t$.

Now suppose $G'$ has a negative cycle with a path to $t$. Since no edge leaves $t$ in $G'$, this cycle cannot contain $t$. Since $G'$ is the same as $G$ aside from the node $t$, it follows that this cycle is also a negative cycle of $G$. ■

So it is really enough to solve the problem of deciding whether $G$ has a negative cycle that has a path to a given sink node $t$, and we do this now.

## Designing and Analyzing the Algorithm

To get started thinking about the algorithm, we begin by adopting the original version of the Bellman-Ford Algorithm, which was less efficient in its use of space. We first extend the definitions of $\text{OPT}(i, v)$ from the Bellman-Ford Algorithm, defining them for values $i \geq n$. With the presence of a negative cycle in the graph, (6.22) no longer applies, and indeed the shortest path may

get shorter and shorter as we go around a negative cycle. In fact, for any node $v$ on a negative cycle that has a path to $t$, we have the following.

**(6.30)** *If node $v$ can reach node $t$ and is contained in a negative cycle, then*

$$\lim_{i \to \infty} \text{OPT}(i, v) = -\infty.$$

If the graph has no negative cycles, then (6.22) implies following statement.

**(6.31)** *If there are no negative cycles in $G$, then $\text{OPT}(i, v) = \text{OPT}(n-1, v)$ for all nodes $v$ and all $i \geq n$.*

But for how large an $i$ do we have to compute the values $\text{OPT}(i, v)$ before concluding that the graph has no negative cycles? For example, a node $v$ may satisfy the equation $\text{OPT}(n, v) = \text{OPT}(n-1, v)$, and yet still lie on a negative cycle. (Do you see why?) However, it turns out that we will be in good shape if this equation holds for all nodes.

**(6.32)** *There is no negative cycle with a path to $t$ if and only if $\text{OPT}(n, v) = \text{OPT}(n-1, v)$ for all nodes $v$.*

**Proof.** Statement (6.31) has already proved the forward direction. For the other direction, we use an argument employed earlier for reasoning about when it's safe to stop the Bellman-Ford Algorithm early. Specifically, suppose $\text{OPT}(n, v) = \text{OPT}(n-1, v)$ for all nodes $v$. The values of $\text{OPT}(n+1, v)$ can be computed from $\text{OPT}(n, v)$; but all these values are the same as the corresponding $\text{OPT}(n-1, v)$. It follows that we will have $\text{OPT}(n+1, v) = \text{OPT}(n-1, v)$. Extending this reasoning to future iterations, we see that none of the values will ever change again, that is, $\text{OPT}(i, v) = \text{OPT}(n-1, v)$ for all nodes $v$ and all $i \geq n$. Thus there cannot be a negative cycle $C$ that has a path to $t$; for any node $w$ on this cycle $C$, (6.30) implies that the values $\text{OPT}(i, w)$ would have to become arbitrarily negative as $i$ increased. ■

Statement (6.32) gives an $O(mn)$ method to decide if $G$ has a negative cycle that can reach $t$. We compute values of $\text{OPT}(i, v)$ for nodes of $G$ and for values of $i$ up to $n$. By (6.32), there is no negative cycle if and only if there is some value of $i \leq n$ at which $\text{OPT}(i, v) = \text{OPT}(i-1, v)$ for all nodes $v$.

So far we have determined whether or not the graph has a negative cycle with a path from the cycle to $t$, but we have not actually found the cycle. To find a negative cycle, we consider a node $v$ such that $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$: for this node, a path $P$ from $v$ to $t$ of cost $\text{OPT}(n, v)$ must use *exactly* $n$ edges. We find this minimum-cost path $P$ from $v$ to $t$ by tracing back through the subproblems. As in our proof of (6.22), a simple path can only have $n-1$

edges, so $P$ must contain a cycle $C$. We claim that this cycle $C$ has negative cost.

**(6.33)** *If $G$ has $n$ nodes and $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$, then a path $P$ from $v$ to $t$ of cost $\text{OPT}(n, v)$ contains a cycle $C$, and $C$ has negative cost.*

**Proof.** First observe that the path $P$ must have $n$ edges, as $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$, and so every path using $n-1$ edges has cost greater than that of the path $P$. In a graph with $n$ nodes, a path consisting of $n$ edges must repeat a node somewhere; let $w$ be a node that occurs on $P$ more than once. Let $C$ be the cycle on $P$ between two consecutive occurrences of node $w$. If $C$ were not a negative cycle, then deleting $C$ from $P$ would give us a $v$-$t$ path with fewer than $n$ edges and no greater cost. This contradicts our assumption that $\text{OPT}(n, v) \neq \text{OPT}(n-1, v)$, and hence $C$ must be a negative cycle. ■

**(6.34)** *The algorithm above finds a negative cycle in $G$, if such a cycle exists, and runs in $O(mn)$ time.*

## Extensions: Improved Shortest Paths and Negative Cycle Detection Algorithms

At the end of Section 6.8 we discussed a space-efficient implementation of the Bellman-Ford algorithm for graphs with no negative cycles. Here we implement the detection of negative cycles in a comparably space-efficient way. In addition to the savings in space, this will also lead to a considerable speedup in practice even for graphs with no negative cycles. The implementation will be based on the same pointer graph $P$ derived from the "first edges" $(v, \mathit{first}[v])$ that we used for the space-efficient implementation in Section 6.8. By (6.27), we know that if the pointer graph ever has a cycle, then the cycle has negative cost, and we are done. But if $G$ has a negative cycle, does this guarantee that the pointer graph will ever have a cycle? Furthermore, how much extra computation time do we need for periodically checking whether $P$ has a cycle?

Ideally, we would like to determine whether a cycle is created in the pointer graph $P$ every time we add a new edge $(v, w)$ with $\mathit{first}[v] = w$. An additional advantage of such "instant" cycle detection will be that we will not have to wait for $n$ iterations to see that the graph has a negative cycle: We can terminate as soon as a negative cycle is found. Earlier we saw that if a graph $G$ has no negative cycles, the algorithm can be stopped early if in some iteration the shortest path values $M[v]$ remain the same *for all nodes* $v$. Instant negative cycle detection will be an analogous early termination rule for graphs that have negative cycles.

Consider a new edge $(v, w)$, with $first[v] = w$, that is added to the pointer graph $P$. Before we add $(v, w)$ the pointer graph has no cycles, so it consists of paths from each node $v$ to the sink $t$. The most natural way to check whether adding edge $(v, w)$ creates a cycle in $P$ is to follow the current path from $w$ to the terminal $t$ in time proportional to the length of this path. If we encounter $v$ along this path, then a cycle has been formed, and hence, by (6.27), the graph has a negative cycle. Consider Figure 6.26, for example, where in both (a) and (b) the pointer $first[v]$ is being updated from $u$ to $w$; in (a), this does not result in a (negative) cycle, but in (b) it does. However, if we trace out the sequence of pointers from $v$ like this, then we could spend as much as $O(n)$ time following the path to $t$ and still not find a cycle. We now discuss a method that does not require an $O(n)$ blow-up in the running time.

We know that before the new edge $(v, w)$ was added, the pointer graph was a directed tree. Another way to test whether the addition of $(v, w)$ creates a cycle is to consider all nodes in the subtree directed toward $v$. If $w$ is in this subtree, then $(v, w)$ forms a cycle; otherwise it does not. (Again, consider the two sample cases in Figure 6.26.) To be able to find all nodes in the subtree directed toward $v$, we need to have each node $v$ maintain a list of all other nodes whose selected edges point to $v$. Given these pointers, we can find the subtree in time proportional to the size of the subtree pointing to $v$, at most $O(n)$ as before. However, here we will be able to make additional use of the work done. Notice that the current distance value $M[x]$ for all nodes $x$ in the subtree was derived from node $v$'s old value. We have just updated $v$'s distance, and hence we know that the distance values of all these nodes will be updated again. We'll mark each of these nodes $x$ as "dormant," delete the

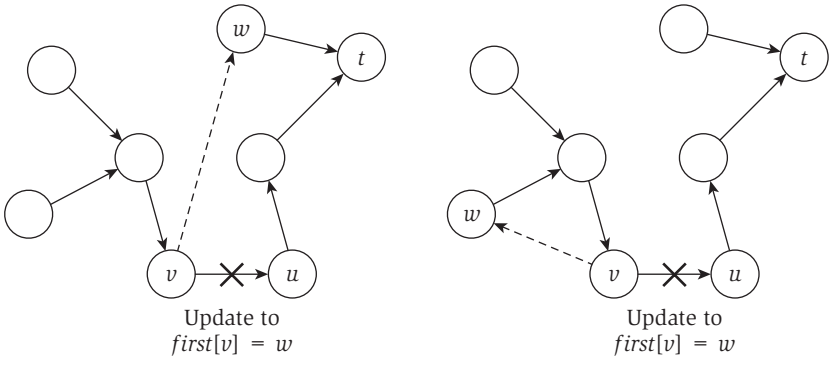



**Figure 6.26** Changing the pointer graph $P$ when $first[v]$ is updated from $u$ to $w$. In (b), this creates a (negative) cycle, whereas in (a) it does not.

edge $(x, first[x])$ from the pointer graph, and not use $x$ for future updates until its distance value changes.

This can save a lot of future work in updates, but what is the effect on the worst-case running time? We can spend as much as $O(n)$ extra time marking nodes dormant after every update in distances. However, a node can be marked dormant only if a pointer had been defined for it at some point in the past, so the time spent on marking nodes dormant is at most as much as the time the algorithm spends updating distances.

Now consider the time the algorithm spends on operations other than marking nodes dormant. Recall that the algorithm is divided into iterations, where iteration $i+1$ processes nodes whose distance has been updated in iteration $i$. For the original version of the algorithm, we showed in (6.26) that after $i$ iterations, the value $M[v]$ is no larger than the value of the shortest path from $v$ to $t$ using at most $i$ edges. However, with many nodes dormant in each iteration, this may not be true anymore. For example, if the shortest path from $v$ to $t$ using at most $i$ edges starts on edge $e = (v, w)$, and $w$ is dormant in this iteration, then we may not update the distance value $M[v]$, and so it stays at a value higher than the length of the path through the edge $(v, w)$. This seems like a problem—however, in this case, the path through edge $(v, w)$ is not actually the shortest path, so $M[v]$ will have a chance to get updated later to an even smaller value.

So instead of the simpler property that held for $M[v]$ in the original versions of the algorithm, we now have the the following claim.

**(6.35)** *Throughout the algorithm $M[v]$ is the length of some simple path from $v$ to $t$; the path has at least $i$ edges if the distance value $M[v]$ is updated in iteration $i$; and after $i$ iterations, the value $M[v]$ is the length of the shortest path for all nodes $v$ where there is a shortest $v$-$t$ path using at most $i$ edges.*

**Proof.** The *first* pointers maintain a tree of paths to $t$, which implies that all paths used to update the distance values are simple. The fact that updates in iteration $i$ are caused by paths with at least $i$ edges is easy to show by induction on $i$. Similarly, we use induction to show that after iteration $i$ the value $M[v]$ is the distance on all nodes $v$ where the shortest path from $v$ to $t$ uses at most $i$ edges. Note that nodes $v$ where $M[v]$ is the actual shortest-path distance cannot be dormant, as the value $M[v]$ will be updated in the next iteration for all dormant nodes. ■

Using this claim, we can see that the worst-case running time of the algorithm is still bounded by $O(mn)$: Ignoring the time spent on marking nodes dormant, each iteration is implemented in $O(m)$ time, and there can be at most $n-1$ iterations that update values in the array $M$ without finding

a negative cycle, as simple paths can have at most $n - 1$ edges. Finally, the time spent marking nodes dormant is bounded by the time spent on updates. We summarize the discussion with the following claim about the worst-case performance of the algorithm. In fact, as mentioned above, this new version is in practice the fastest implementation of the algorithm even for graphs that do not have negative cycles, or even negative-cost edges.

**(6.36)** *The improved algorithm outlined above finds a negative cycle in $G$ if such a cycle exists. It terminates immediately if the pointer graph $P$ of $first[v]$ pointers contains a cycle $C$, or if there is an iteration in which no update occurs to any distance value $M[v]$. The algorithm uses $O(n)$ space, has at most $n$ iterations, and runs in $O(mn)$ time in the worst case.*