The Shimbel / Moore / Woodbury-Dantzig / Bellman-Ford / Kalaba / Minty / Brosh[12] algorithm can be summarized in one line:

> BELLMAN-FORD: Relax ***ALL*** the tense edges, then recurse.

```
BellmanFord(s)
  InitSSSP(s)
  while there is at least one tense edge
      for every edge u→v
          if u→v is tense
              Relax(u→v)
```

The following lemma is the key to proving both correctness and efficiency of Bellman-Ford. For every vertex v and non-negative integer i, let $\boldsymbol{dist_{\le i}(v)}$ denote the length of the shortest *walk* in G from s to v consisting of *at most* i edges. In particular, $dist_{\le 0}(s) = 0$ and $dist_{\le 0}(v) = \infty$ for all $v \neq s$.

Lemma 8.6. *For every vertex v and non-negative integer i, after i iterations of the main loop of BellmanFord, we have $dist(v) \le dist_{\le i}(v)$.*

Proof: The proof proceeds by induction on i. The base case $i = 0$ is trivial, so assume $i > 0$. Fix a vertex v, and let W be the shortest walk from s to v consisting of at most i edges (breaking ties arbitrarily). By definition, W has length $dist_{\le i}(v)$. There are two cases to consider.

- Suppose W has no edges. Then W must be the trivial walk from s to s, so $v = s$ and $dist_{\le i}(s) = 0$. We set $dist(s) \leftarrow 0$ in InitSSSP, and $dist(s)$ can never increase, so we always have $dist(s) \le 0$.
- Otherwise, let $u \to v$ be the last edge of W. The induction hypothesis implies that after $i-1$ iterations, $dist(u) \le dist_{\le i-1}(u)$. During the ith iteration of the outer loop, when we consider the edge $u \to v$ in the inner loop, either $dist(v) < dist(u) + w(u \to v)$ already, or we set $dist(v) \leftarrow dist(u) + w(u \to v)$. In both cases, we have $dist(v) \le dist_{\le i-1}(u) + w(u \to v) = dist_{\le i}(v)$. As usual, $dist(v)$ cannot increase (although $dist(v)$ might decrease further before the ith iteration of the outer loop ends).

In both cases, we conclude that $dist(v) \le dist_{\le i}(v)$ at the end of the ith iteration. □

If the input graph has no negative cycles, the shortest walk from s to any other vertex is a simple path with at most $V - 1$ edges; it follows that BELLMAN-FORD halts with the correct shortest-path distances after at most $V - 1$ iterations.

[12] Go read everything in *Hyperbole and a Half* again. And then adopt another cat, so you can buy it another copy of the book.

Said differently, if any edge is still tense after $V-1$ iterations, then the input graph must contain a negative cycle! Thus, we can rewrite the algorithm more concretely as follows:

```
BellmanFord(s)
  InitSSSP(s)
  repeat V − 1 times
      for every edge u→v
          if u→v is tense
              Relax(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

Each iteration of the inner loop trivially requires $O(E)$ time, so the overall algorithm runs in ***O(VE) time***. Thus, Bellman-Ford is *always* efficient, even if the graph has negative edges, and in fact even if the graph has negative *cycles*.

If all edge weights are non-negative, however, Dijkstra's algorithm is faster, at least in the worst case. (In practice, Dijkstra's algorithm is often faster than Bellman-Ford even for graphs with negative edges.)

Moore's Improvement

Neither Moore nor Bellman described the Bellman-Ford algorithm in the form I've presented here. Moore presented his version of the algorithm ("Algorithm D") in the same paper that proposed breadth-first search ("Algorithm A") for unweighted graphs; indeed, the two algorithms are nearly identical. Although Moore's algorithm has the same $O(VE)$ worst-case running time as BellmanFord, it is often significantly faster in practice, intuitively because it avoids checking edges that are "obviously" not tense.

Moore derived his weighted shortest-path algorithm by making two modifications to breadth-first search. First, replace each "+1" with "+$w(u\to v)$" in the innermost loop, to take the edge weights into account. Second, check whether a vertex is already in the FIFO queue before Inserting it, so that the queue always contains at most one copy of each vertex.[13]

Following our earlier analysis of breadth-first search, I'll introduce a "token" to break the execution of the algorithm into phases. Just like BFS, each phase begins when the token is Pushed into the queue, and ends when the token is Pulled out of the queue again. Just like BFS, the algorithm ends when the queue contains *only* the token. The resulting algorithm is shown in Figure 8.16.

Because the queue contains at most one copy of each vertex at any time, each vertex is Pulled from the queue at most once in each phase, and therefore

[13]Moore's algorithm is still *correct* without this check, but the $O(VE)$ time bound is not.

```
MOORE(s):
  INITSSSP(s)
  PUSH(s)
  PUSH(✠)                    ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      u ← PULL( )
      if u = ✠
          PUSH(✠)            ⟨⟨start the next phase⟩⟩
      else
          for all edges u→v
              if u→v is tense
                  RELAX(u→v)
                  if v is not already in the queue
                      PUSH(v)
```

Figure 8.16. Moore's shortest-path algorithm. Bold red lines involving the token ✠ are only for analysis.

each edge $u \to v$ is checked for tenseness at most once in each phase. Moreover, every edge that is tense when a phase begins is relaxed during that phase. (Some edges that become tense during the phase might also be relaxed during that phase, and some relaxed edges might become tense again in the same phase.) Thus, MOORE can be viewed as a refinement of BELLMANFORD that uses a queue to maintain tense edges, rather than testing every edge by brute force. In particular, a similar inductive proof establishes the following analogue of Lemma 8.6:

Lemma 8.7. *For every vertex v and non-negative integer i, after i phases of MOORE, we have $dist(v) \le dist_{\le i}(v)$.*

Thus, if the input graph has no negative cycles, MOORE halts after at most $V - 1$ phases. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the worst-case running time of a single phase is $O(E)$. Thus, the overall running time of MOORE is $\boldsymbol{O(VE)}$. In practice, however, MOORE often computes shortest paths considerably faster than BELLMANFORD, because it only scans an edge $u \to v$ if $dist(u)$ was changed in the previous phase.

If the input graph contains a negative cycle, MOORE never halts. Fortunately, like BELLMANFORD, it is easy to modify Moore's algorithm to report negative cycles if they exist. Perhaps the easiest modification is to *actually* maintain a token, and count the number of times the token is PULLed from the queue. Then the input graph contains a negative cycle if and only if the queue is non-empty immediately after the token is PULLed for the $(V-1)$th time.

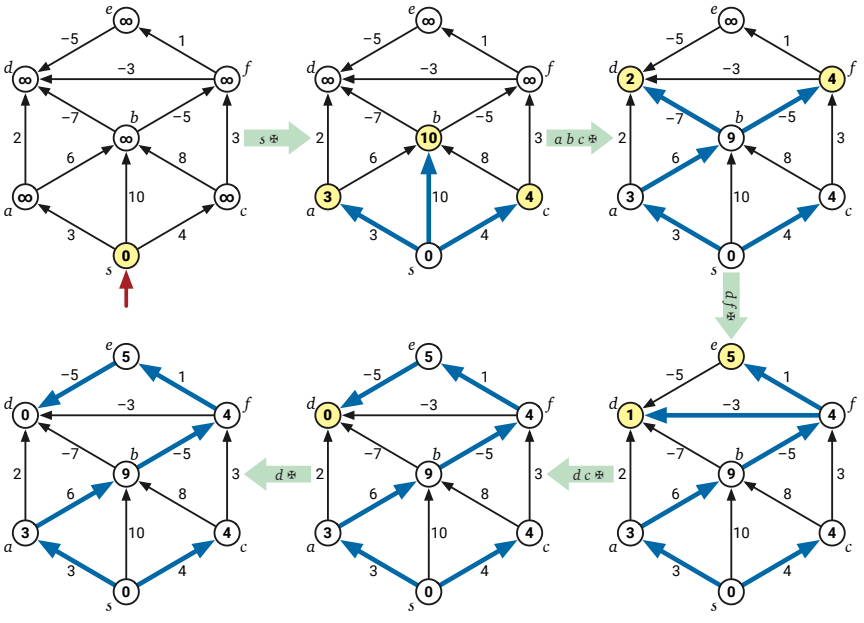


Figure 8.17. A complete run of Moore's algorithm on a directed graph with negative edges. Nodes are pulled from the queue in the order s ✠ a b c ✠ d f ✠ d c ✠ d ✠ ✠, where ✠ is the end-of-phase token. At the start of each phase, bold edges indicate predecessors, and shaded vertices are in the vertex queue. Compare with Figures 8.6 and 8.15.

Dynamic Programming Formulation

Like almost everything else with his name on it, Richard Bellman derived the "Bellman-Ford" shortest-path algorithm via dynamic programming. As usual, we need to start with a recursive definition of shortest path distances. It's tempting to use the same identity that we exploited for directed acyclic graphs:

$$dist(v) = \begin{cases} 0 & \text{if } v = s \\ \min_{u \to v} (dist(u) + w(u \to v)) & \text{otherwise} \end{cases}$$

Unfortunately, if the input graph is not a dag, this recurrence doesn't work! Suppose the input graph contains the directed cycle $u \to v \to w \to u$. To compute $dist(w)$ we first need $dist(v)$, and to compute $dist(v)$ we first need $dist(u)$, but to compute $dist(u)$ we first need $dist(w)$. If the input graph has any directed cycles, we get stuck in an infinite loop!

To support a proper recurrence, we need to add an additional structural parameter to the distance function, which decreases monotonically at each recursive call, defined so that the function is trivial to evaluate when the

parameter reaches 0. Bellman chose ***the maximum number of edges*** as this additional parameter.[14]

As in our earlier analysis, let $dist_{\le i}(v)$ denote the length of the shortest walk from s to v consisting of at most i edges. Bellman observed that this function obeys the following ~~Bellman's equation~~ recurrence:

$$dist_{\le i}(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{c} dist_{\le i-1}(v) \\ \min\limits_{u \to v} \left(dist_{\le i-1}(u) + w(u \to v) \right) \end{array} \right\} & \text{otherwise} \end{cases}$$

Let's assume that the graph has no negative cycles, so our goal is to compute $dist_{\le V-1}(v)$ for every vertex v. Here is a straightforward dynamic-programming evaluation of this recurrence, where $dist[i, v]$ stores the value of $dist_{\le i}(v)$. Correctness of the final shortest-path distances follows from the correctness of the recurrence, and the $O(VE)$ running time is obvious. This is essentially how Bellman presented his shortest-path algorithm.

```
BellmanFordDP(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i − 1,v]
          for every edge u→v
              if dist[i,v] > dist[i − 1,u] + w(u→v)
                  dist[i,v] ← dist[i − 1,u] + w(u→v)
```

We can transform this dynamic programming algorithm into our original formulation of BellmanFord through a short series of minor modifications. First, each iteration of the outermost loop considers each edge $u \to v$ exactly once, but the order in which we consider those edges doesn't actually matter. Thus, we can safely remove one level of indentation from the last three lines! The modified algorithm may consider edges in a different *order*, but it still correctly computes $dist_{\le i}(v)$ for all i and v.

[14] As we'll see in the next chapter, this is not the only reasonable choice.

```
BELLMANFORDDP2(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
      for every edge u→v
          if dist[i,v] > dist[i−1,u] + w(u→v)
              dist[i,v] ← dist[i−1,u] + w(u→v)
```

Next we change the indices in the last two lines from $i-1$ to i. This change may cause the distances $dist[i,v]$ to approach the true shortest-path distances more quickly than before, but the algorithm correctly computes the true shortest path distances. Instead of $dist[i,v] = dist_{\le i}(v)$, we now have $dist[i,v] \le dist_{\le i}(v)$ for all i and v, mirroring Lemmas 8.6 and 8.7.

```
BELLMANFORDDP3(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
      for every edge u→v
          if dist[i,v] > dist[i,u] + w(u→v)      ⟨⟨not i − 1!⟩⟩
              dist[i,v] ← dist[i,u] + w(u→v)     ⟨⟨not i − 1!⟩⟩
```

But this algorithm is a little silly. In the ith iteration of the outermost loop, we first copy the $(i-1)$th row of the array $dist[*,*]$ to the ith row, and then modify the elements of the ith row. So we really don't need a two-dimensional array at all; the iteration index i is completely redundant! In our final modification, we maintain only a one-dimensional array of tentative distances.

```
BELLMANFORDFINAL(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

This final dynamic programming algorithm is almost identical to our original formulation of BELLMANFORD! The first three lines initialize the shortest path distances, and the last two lines relax the edge $u \to v$ if that edge is tense.

BellmanFordFinal is missing only two features of our earlier formulation: It does not maintain predecessor pointers or detect negative cycles. Fortunately, adding those features is straightforward.

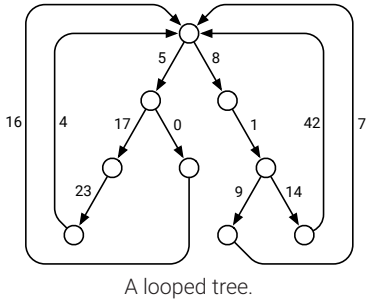
Exercises

0. Let G be a directed graph with arbitrary edge weights (which may be positive, negative, or zero), possibly with negative cycles, and let s be an arbitrary vertex of G.

   (a) Suppose every vertex v stores a number $dist(v)$ (but no predecessor pointers). Describe and analyze an algorithm to determine whether $dist(v)$ is the shortest-path distance from s to v, for every vertex v.

   (b) Suppose instead that every vertex $v \neq s$ stores a pointer $pred(v)$ to another vertex in G (but no distances). Describe and analyze an algorithm to determine whether these predecessor pointers define a single-source shortest path tree rooted at s.

1. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

2. Suppose we are given a directed graph G with weighted edges and two vertices s and t.

   (a) Describe and analyze an algorithm to find the shortest path from s to t when exactly one edge in G has negative weight. *[Hint: Modify Dijkstra's algorithm. Or don't.]*

(b) Describe and analyze an algorithm to find the shortest path from s to t when exactly k edges in G have negative weight. How does the running time of your algorithm depend on k?

3. Suppose we are given an undirected graph G in which every *vertex* has a positive weight.

   (a) Describe and analyze an algorithm to find a *spanning tree* of G with minimum total weight. (The total weight of a spanning tree is the sum of the weights of its vertices.)

   (b) Describe and analyze an algorithm to find a *path* in G from one given vertex s to another given vertex t with minimum total weight. (The total weight of a path is the sum of the weights of its vertices.)

   [Hint: One of these problems is trivial.]

4. For any edge e in any graph G, let $G \setminus e$ denote the graph obtained by deleting e from G. Suppose we are given a graph G and two vertices s and t. The *replacement paths* problem asks us to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G. The output is an array of E distances, one for each edge of G.

   (a) Suppose G is a *directed* graph, and the shortest path from vertex s to vertex t passes through *every* vertex of G. Describe an algorithm to solve this special case of the replacement paths problem in $O(E \log V)$ time.

   ♥(b) Describe an algorithm to solve the replacement paths problem for arbitrary *undirected* graphs in $O(E \log V)$ time.

   In both subproblems, you may assume that all edge weights are non-negative. *[Hint: If we delete an edge of the original shortest path, how do the old and new shortest paths overlap?]*

5. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G, and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in $O(E \log V)$ time.

6. (a) Describe and analyze a modification of Bellman-Ford that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

(b) Describe and analyze a modification of Bellman-Ford that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with $dist(v) = -\infty$; otherwise, $dist(v)$ should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

♥(c) Repeat parts (a) and (b), but for Ford's generic relaxation algorithm. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle; your modified algorithms should also run in $O(2^V)$ time.

7. Consider the following even looser variant of Ford's generic relaxation algorithm:

```
FellmanBored(s):
  InitSSSP(s)
  repeat
      e_i ← any edge in G
      if e_i is tense
          Relax(e_i)
  until you get bored or whatever
```

Prove that if FellmanBored examines the edges of any walk W starting from s, in order along W, then the last distance label in W is at most the length of W. More formally: If the edges of any walk $v_0 \to v_1 \to \cdots \to v_\ell$, where $v_0 = s$, define a *subsequence* of the edges $e_1, e_2, e_3, \ldots$ examined by FellmanBored, then we have $dist(\ell) \le \sum_{i=1}^{\ell} w(v_{i-1} \to v_i)$. *[Hint: This property is almost easier to prove than it is to state correctly.]*

8. This problem considers several ways to detect negative cycles using Ford's generic relaxation algorithm.

(a) Prove that if $pred(s)$ ever changes after InitSSSP, then the input graph contains a negative cycle through s.

(b) Show that $pred(s)$ might never change after InitSSSP, even when the input graph contains a negative cycle through s.

(c) Let P denote the current graph of predecessor edges $pred(v) \to v$, and let X denote the set of all currently *tens*e edges; both of these sets evolve as the algorithm executes. Prove that the input graph has no negative cycles if and only if $P \cup X$ is always a dag.

(d) Let R denote the set of all edges that have been relaxed so far; this set grows as the algorithm executes. Prove that the input graph has no negative cycles if and only if R is always a dag.

♥9. Prove that Dijkstra's algorithm performs $\Omega(2^V)$ relaxations in the worst case when edges are allowed to have negative weight, even if the underlying graph is acyclic. Specifically, for every positive integer n, construct a n-vertex dag G_n with weighted edges, such that Dijkstra's algorithm calls RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Towers of Hanoi.]*

♥10. Prove that Ford's generic relaxation algorithm (and therefore Dijkstra's algorithm) halts after at most $O(2^V)$ relaxations, unless the input graph contains a negative cycle. *[Hint: See Problem 8(d).]*

11. Although we typically speak of "the" shortest path between two nodes, single graph could contain several minimum-length paths with the same endpoints. Even for weighted graphs, it is often convenient to choose a minimum-weight path with the fewest edges; call this a ***best path*** from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute *best* paths in G from s to every other vertex.

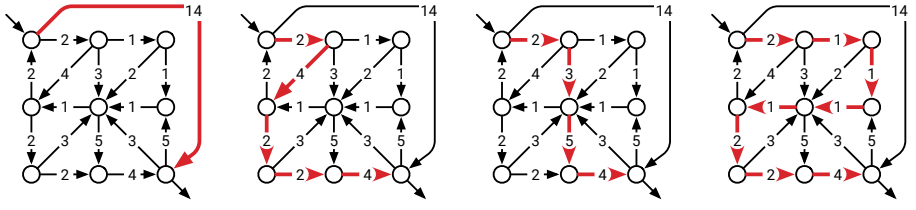


Figure 8.18. Four (of many) equal-length shortest paths. The first path is the "best" shortest path.

12. Describe and analyze an algorithm to determine the *number* of shortest paths from a source vertex s to a target vertex t in an arbitrary directed graph G with weighted edges. You may assume that all edge weights are positive and that all necessary arithmetic operations can be performed in $O(1)$ time. *[Hint: Compute shortest path distances from s to every other vertex. Throw away all edges that cannot be part of a shortest path from s to another vertex. What's left?]*

13. You just discovered your best friend from elementary school on Twitbook. You both want to meet as soon as possible, but you live in two different cites that are far apart. To minimize travel time, you agree to meet at an intermediate city, and then you simultaneously hop in your cars and start driving toward each other. But where *exactly* should you meet?

    You are given a weighted graph $G = (V, E)$, where the vertices V represent cities and the edges E represent roads that directly connect cities. Each edge e has a weight $w(e)$ equal to the time required to travel between the

two cities. You are also given a vertex p, representing your starting location, and a vertex q, representing your friend's starting location.

Describe and analyze an algorithm to find the target vertex t that allows you and your friend to meet as quickly as possible.

14. You are hired as a cyclist for the Giggle Highway View project, which will provide street-level images along the entire US national highway system. As a pilot project, you are asked to ride the Giggle Highway-View Fixed-Gear Carbon-Fiber Bicycle from "the Giggleplex" in Portland, Oregon to "Gigglesburg" in Williamsburg, Brooklyn, New York.

    You are a hopeless caffeine addict, but like most Giggle employees you are also a coffee snob; you only drink independently roasted organic shade-grown single-origin espresso. After each espresso shot, you can bike up to L miles before suffering a caffeine-withdrawal migraine.

    Giggle helpfully provides you with a map of the United States, in the form of an undirected graph G, whose vertices represent coffee shops that sell independently roasted organic shade-grown single-origin espresso, and whose edges represent highway connections between them. Each edge e is labeled with the length $\ell(e)$ of the corresponding stretch of highway. Naturally, there are espresso stands at both Giggle offices, represented by two specific vertices s and t in the graph G.

    (a) Describe and analyze an algorithm to determine whether it is possible to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine.

    (b) You discover that by wearing a more expensive fedora, you can increase the distance L that you can bike between espresso shots. Describe and analyze and algorithm to find the minimum value of L that allows you to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine.

    (c) When you report to your supervisor (whom Giggle recently hired away from their competitor Unter) that the ride is impossible, she demands to look at your map. "Oh, I see the problem; there are no *Starbucks* on this map!" As you look on in horror, she hands you an updated graph G' that includes a vertex for every Starbucks location in the United States, helpfully marked in Starbucks Green (Pantone® 3425 C).

        Describe and analyze an algorithm to find the minimum number of Starbucks locations you must visit to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine. More formally, your algorithm should find the minimum number of green vertices on any path in G' from s to t that uses only edges of length at most L.