I study my Bible as I gather apples. First I shake the whole tree, that the ripest might fall. Then I climb the tree and shake each limb, and then each branch and then each twig, and then I look under each leaf.

— attributed to Martin Luther (c. 1500)

Life is an unfoldment, and the further we travel the more truth we can comprehend. To understand the things that are at our door is the best preparation for understanding those that lie beyond.

— attributed to Hypatia of Alexandria (c. 400) by Elbert Hubbard in *Little Journeys to the Homes of Great Teachers* (1908)

Your mind will answer most questions if you learn to relax and wait for the answer. Like one of those thinking machines, you feed in your question, sit back, and wait . . .

— William S. Burroughs, *Naked Lunch* (1959)

The methods given in this paper require no foresight or ingenuity, and hence deserve to be called algorithms.

— Edward R. Moore, "The Shortest Path Through a Maze" (1959)

8 Shortest Paths

Suppose we are given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, and we want to find the shortest path from a *source* vertex s to a *target* vertex t. That is, we want to find the directed path P starting at s and ending at t that minimizes the function

$$w(P) := \sum_{u \to v \in P} w(u \to v).$$

For example, if I want to answer the question "What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?", I might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Champaign, and t is Columbus.[1] The graph is directed, because driving times along the same road might be different

[1]West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We live in Urbana now.

in different directions. (At one time, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.)

8.1 Shortest Path Trees

Almost every algorithm known for computing shortest paths from one vertex to another actually solves (large portions of) the following more general ***single source shortest path*** or ***SSSP*** problem: Find shortest paths from the source vertex s to *every* other vertex in the graph. This problem is usually solved by finding a ***shortest path tree*** rooted at s that contains all the desired shortest paths.

It's not hard to see that if shortest paths are unique, then they form a tree, because any subpath of a shortest path is itself a shortest path. If there are multiple shortest paths to some vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree. If there are shortest paths from s to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths without changing its length, so that the two paths only diverge once.

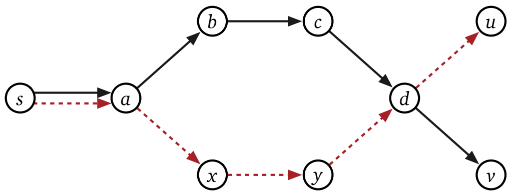


Figure 8.1. If $s{\to}a{\to}b{\to}c{\to}d{\to}v$ (solid) and $s{\to}a{\to}x{\to}y{\to}d{\to}u$ (dashed) are shortest paths, then $s{\to}a{\to}b{\to}c{\to}d{\to}u$ (along the top) is also a shortest path.

Although they are both optimal spanning trees, shortest-path trees and minimum spanning trees are very different creatures. Shortest-path trees are rooted and directed; minimum spanning trees are unrooted and undirected. Shortest-path trees are most naturally defined for directed graphs; minimum spanning trees are more naturally defined for undirected graphs. If edge weights are distinct, there is only one minimum spanning tree, but every source vertex induces a different shortest-path tree; moreover, it is possible for *every* shortest path tree to use a different set of edges from the minimum spanning tree.

♥8.2 Negative Edges

For most shortest-path problems, where the edge weights correspond to distance or length or time, it is natural to assume that all edge weights are non-negative, or even positive. However, for many applications of shortest-path algorithms, it is natural to consider edges with negative weight. For example, the weight

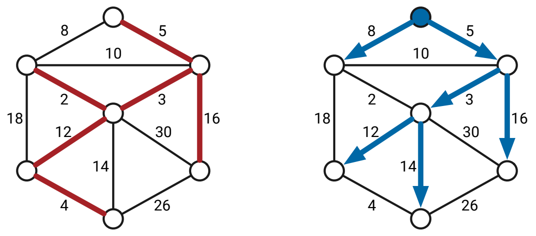


Figure 8.2. A minimum spanning tree and a shortest path tree of the same undirected graph.

of an edge might represent the *cost* of moving from one vertex to another, so negative-weight edges represent transitions with negative cost, or equivalently, transitions that earn a profit.

Negative edges are a thorn in the side of most shortest-path problems, because the presence of a negative *cycle* might imply that shortest paths may not be well-defiend. To be precise, a shortest path from s to t exists if and only if there is at least one path from s to t, but there is no path from s to t that touches a negative cycle. For *any* path from s to t that touches a negative cycle, there is a shorter path from s to t that goes around the cycle one more time.[2] Thus, if at least one path from s to t touches a negative cycle, there is no shortest path from s to t.

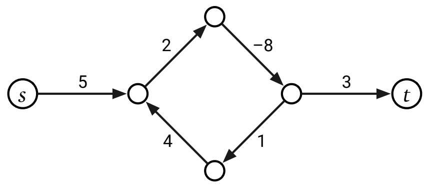


Figure 8.3. There is no shortest walk from s to t.

In part because we need to consider negative edge weights, this chapter explicitly considers *only* directed graphs. All of the algorithms described here also work for undirected graphs with essentially trivial modifications, *if and only if* negative edges are prohibited. Correctly handling negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, because this would transform any negative edge into a short negative cycle. Subpaths of an *undirected* shortest path that contains a negative edge are *not* necessarily shortest paths; consequently, the set of all undirected shortest paths from a single source vertex may not define a tree, even if shortest paths are unique.

[2]Technically, we should be discussing shortest *walks* here, rather than shortest *paths*, but the abuse of terminology is standard. If s can reach t, there must be a shortest simple path from s to t; it's just NP-hard to compute (when there are negative cycles), by an easy reduction from the Hamiltonian path problem. On the other hand, if there is a shortest *walk* from s to t, that walk must be a simple path, and therefore must be the shortest simple path from s to t. Blerg.

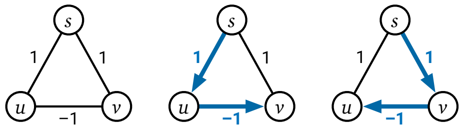


Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

A complete treatment of undirected graphs with negative edges is beyond the scope of this book. I will only mention, for people who want to follow up via Google, that a *single* shortest path in an undirected graph with negative edges can be computed in $O(VE + V^2 \log V)$ time, by a reduction to maximum weighted matching.

8.3 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, many different SSSP algorithms can be described as special cases of a single generic algorithm, first proposed by Lester Ford in 1956 and independently described by George Dantzig in 1957[3] and again by George Minty in 1958. Each vertex v in the graph stores two values, which (inductively) describe a *tentative* shortest path from s to v.

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or NULL if there is no such vertex.

The predecessor pointers automatically define a tentative shortest-path *tree* rooted at s; these pointers are exactly the same as the parent pointers in our generic graph traversal algorithm. At the beginning of the algorithm, we initialize the distances and predecessors as follows:

```
INITSSSP(s):
    $dist(s) \leftarrow 0$
    $pred(s) \leftarrow$ NULL
    for all vertices $v \neq s$
        $dist(v) \leftarrow \infty$
        $pred(v) \leftarrow$ NULL
```

During the execution of the algorithm, an edge $u \rightarrow v$ is ***tense*** if $dist(u) + w(u \rightarrow v) < dist(v)$. If $u \rightarrow v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \rightarrow v$ is shorter. We can correct (or at least improve) this obvious overestimate by ***relaxing*** the edge as follows:

[3]Specifically, Dantzig showed that the shortest path problem can be phrased as a linear programming problem, and then described an interpretation of his simplex method in terms of the original graph. His description is (morally) equivalent to Ford's relaxation strategy.

```
RELAX(u→v):
    $dist(v) \leftarrow dist(u) + w(u\to v)$
    $pred(v) \leftarrow u$
```

Now that everything is set up, Ford's generic algorithm has a simple one-line description:

Repeatedly relax tense edges, until there are no more tense edges.

```
FORDSSSP(s):
    INITSSSP(s)
    while there is at least one tense edge
        RELAX any tense edge
```

If FordSSSP eventually terminates (because there are no more tense edges), then the predecessor pointers correctly define a shortest-path tree, and each value $dist(v)$ is the actual shortest-path distance from s to v. In particular, if s cannot reach v, then $dist(v) = \infty$, and if any negative cycle is reachable from s, then the algorithm never terminates.

The correctness of Ford's generic algorithm follows from the following series of simpler claims:

1. At any moment during the execution of the algorithm, for every vertex v, the distance $dist(v)$ is either ∞ or the length of a walk from s to v. This claim can be proved by induction on the number of relaxations.

2. If the graph has no negative cycles, then $dist(v)$ is either ∞ or the length of some *simple path* from s to v. Specifically, if $dist(v)$ is the length of a walk from s to v that contains a directed cycle, that cycle must have negative length. This claim implies that if G has no negative cycles, the relaxation algorithm eventually halts, because there are only a finite number of simple paths in G.

3. If no edge in G is tense, then for every vertex v, the distance $dist(v)$ is the length of the predecessor path $s\to\cdots pred(pred(v))\to pred(v)\to v$. Specifically, if v violates this condition but its predecessor $pred(v)$ does not, the edge $pred(v)\to v$ is tense.

4. If no edge in G is tense, then for every vertex v, the path of predecessor edges $s\to\cdots\to pred(pred(v))\to pred(v)\to v$ is in fact a shortest path from s to v. Specifically, if v violates this condition but its predecessor u *in some shortest path* does not, the edge $u\to v$ is tense. This claim also implies that if G has a negative cycle, then some edge is *always* tense, so the generic algorithm never halts.

So far I haven't said anything about how to find tense edges, or which tense edge(s) to relax if there is more than one. Just like whatever-first search, there are several different instantiations of Ford's generic relaxation algorithm. Unlike whatever-first search, however, the efficiency and correctness of each search strategy depends on the structure of the input graph.

The rest of this chapter considers the four most common instantiations of Ford's algorithm, each of which is the best choice for a different class of input graphs. I'll leave the remaining details of the generic correctness proof as exercises, and instead give (more informative, self-contained) correctness proofs for each of these four specific algorithms.

8.4 Unweighted Graphs: Breadth-First Search

In the simplest special case of the shortest path problem, all edges have weight 1, and the length of a path is just the number of edges. This special case can be solved by a species of our generic graph-traversal algorithm called ***breadth-first search***. Breadth-first search is often attributed to Edward Moore, who described it in 1957 (as "Algorithm A") as the first published method to find the shortest path through a maze.[4] Especially in the context of VLSI wiring and robot path planning, breadth-first search is sometimes attributed to Chin Yang Lee, who described several applications of Moore's "Algorithm A" (with proper credit to Moore) in 1961. However, in 1945, more than a decade before Moore considered mazes, Konrad Zuse described an implementation of breadth-first search, as a method to count and label the components of a disconnected graph.[6]

[4]Moore was motivated by a weakness in Claude Shannon's maze-solving robot "Theseus", which Shannon designed and constructed in 1950. (Theseus used a memoized version of depth-first search, implemented using electromechanical relays; this was almost certainly the first *implementation* of depth-first search in graphs.) According to Moore, "When this machine was used with a maze which had more than one solution, a visitor asked why it had not been built to always find the shortest path. Shannon and I each attempted to find economical methods of doing this by machine. He found several methods suitable for analog computation,[5] and I obtained these algorithms."

[5]Analog methods for computing shortest paths through mazes have been proposed using ball bearings, fluid/plasma flow, chemical reaction waves, chemotaxis, resistor networks, electric circuits with LEDs, memristor networks, glow discharge in microfluidic chips, growing plants, slime mold, amoebas, ants, bees, nematodes, and tourists.

[6]Konrad Zuse was one of the early pioneers of computing; he designed and built his first programmable computer (later dubbed the Z1) in the late 1930s from metal strips and rods in his parents' living room; the Z1 and its original blueprints were destroyed by a British air raid in 1944. Zuse's 1945 PhD thesis describes the very first high-level programming language, called *Plankalkül*. The first complete example of a Plankalkül program in Zuse's thesis is an implementation of breadth-first search to count components, along with a pseudocode explanation and an illustrated step-by-step trace of the algorithm's execution on a disconnected graph with eight vertices. Due to the collapse of the Nazi government, Zuse was unable to submit his PhD thesis, and Plankalkül remained unpublished until 1972. The first Plankalkül compiler was finally implemented in 1975 by Joachim Hohmann.

Breadth-first search maintains a first-in-first-out queue of vertices, which initially contains only the source vertex s. At each iteration, the algorithm PULLS a vertex u from the front of the queue and examines each of its outgoing edges $u \to v$. Whenever the algorithm discovers an outgoing tense edge $u \to v$, it relaxes that edge and PUSHES vertex v onto the queue. The algorithm ends when the queue becomes empty.

```
BFS($s$):
  INITSSSP($s$)
  PUSH($s$)
  while the queue is not empty
      $u \leftarrow$ PULL( )
      for all edges $u \to v$
          if $dist(v) > dist(u) + 1$          ⟨⟨if $u \to v$ is tense⟩⟩
              $dist(v) \leftarrow dist(u) + 1$
              $pred(v) \leftarrow u$            ⟨⟨relax $u \to v$⟩⟩
              PUSH($v$)
```

Breadth-first search is somewhat easier to analyze if we break its execution into *phases*, by introducing an imaginary *token*. Before we PULL any vertices, we PUSH the token into the queue. The current phase ends when we PULL the token out of the queue; we begin the next phase when we PUSH the token into the queue again. Thus, the first phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. The modified algorithm is shown in Figure 8.5, and Figure 8.6 shows an example of this algorithm in action. Let me emphasize that these modifications are merely a convenience for analysis; with or without the token, the algorithm PUSHES and PULLS vertices in the same order, scans edges in the same order, and outputs exactly the same distances and predecessors.

```
BFSWITHTOKEN($s$):
  INITSSSP($s$)
  PUSH($s$)
  PUSH(✠)                  ⟨⟨start the first phase⟩⟩
  while the queue contains at least one vertex
      $u \leftarrow$ PULL( )
      if $u$ = ✠
          PUSH(✠)          ⟨⟨start the next phase⟩⟩
      else
          for all edges $u \to v$
              if $dist(v) > dist(u) + 1$          ⟨⟨if $u \to v$ is tense⟩⟩
                  $dist(v) \leftarrow dist(u) + 1$
                  $pred(v) \leftarrow u$            ⟨⟨relax $u \to v$⟩⟩
                  PUSH($v$)
```

Figure 8.5. Breadth-first search with an end-of-phase token (✠); bold red lines are only for analysis.

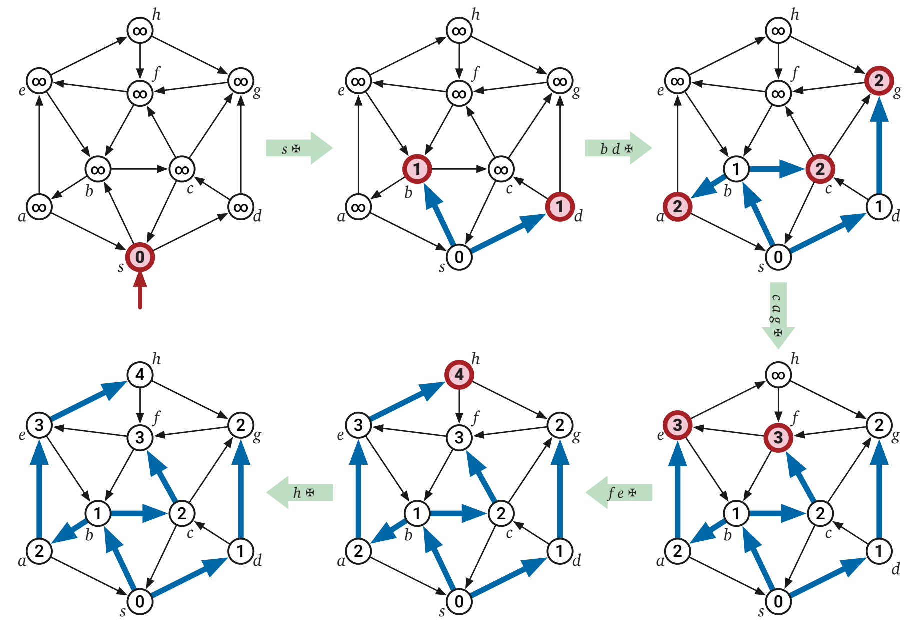


Figure 8.6. A complete run of breadth-first search in a directed graph. Vertices are pulled from the queue in the order *s* ✠ *b d* ✠ *c a g* ✠ *f e* ✠ *h* ✠ ✠, where ✠ is the end-of-phase token. Bold vertices are in the queue at the end of each phase. Bold edges describe the evolving shortest path tree.

Let me emphasize that in the following lemma, $dist(v)$ is just a variable maintained by the algorithm. While $dist(v)$ *intuitively* represents a tentative shortest-path distance, we cannot assume (yet) that $dist(v)$ is ever actually equal to the true shortest-path distance from s to v. Don't worry; we'll get there.

Lemma 8.1. *For every integer $i \geq 0$ and every vertex v, at the end of the ith phase, either $dist(v) = \infty$ or $dist(v) \leq i$, and v is in the queue if and only if $dist(v) = i$.*

Proof: The proof proceeds by induction on i. The base case $i = 0$ is straightforward: At the start of the first phase ("at the end of the zeroth phase"), the queue contains only the start vertex s and the token ✠, and InitSSSP just set $dist(s) \leftarrow 0$ and $dist(v) \leftarrow \infty$ for all $v \neq s$.

So fix an integer $i > 0$. The inductive hypothesis implies that at the *start* of the ith phase, the queue contains every vertex u with $dist(u) = i - 1$, followed by the token ✠. In other words, the queue looks like this:

$$\rightarrow \quad \maltese \quad i-1 \quad i-1 \quad \cdots \quad i-1 \quad \rightarrow$$

Thus, before we Pull the token ✠ from the queue, ending the ith phase, we Pull *every* vertex u with $dist(u) = i - 1$.

For each such vertex u, we consider every outgoing edge $u{\to}v$. If $u{\to}v$ is tense, we set $dist(v) \leftarrow dist(u) + 1$, so that $dist(v) = i$, and then immediately

Pᴜsʜ v into the queue. These are the only assignments to distance labels during the ith phase. Thus, by induction, during the entire ith phase, the queue contains some vertices with distance label $i-1$, followed by the token, followed by some vertices with distance label i:

$$\rightarrow \; i \;\; \cdots \;\; i \;\; \maltese \;\; i-1 \;\; \cdots \;\; i-1 \; \rightarrow$$

In particular, just before the ith phase ends, the queue contains the token, followed by some vertices with distance label i.

$$\rightarrow \; i \;\; i \;\; \cdots \;\; i \;\; \maltese \; \rightarrow$$

Moreover, vertex v appears in this final queue if and only if $dist(v)$ was changed during the ith phase. Thus, at the end of the ith phase, the queue contains *every* vertex v with $dist(v) = i$. □

Lemma 8.1 implies that the main body of BFS assigns distance labels in non-decreasing order; on the other hand, the distance label $dist(v)$ of each vertex v never increases. It follows that for each vertex v, the line "$dist(v) \leftarrow dist(u) + 1$" is executed *at most once*, during phase $dist(v)$. Similarly:

- Each predecessor pointer $pred(v)$ is changed at most once, during phase $dist(v)$.
- Each vertex v is Pᴜsʜed into the queue at most once, during phase $dist(v)$.
- Each vertex u is Pᴜʟʟed from the queue at most once, during phase $dist(u)+1$.
- For each edge $u\rightarrow v$, the comparison "is $dist(v) > dist(u) + 1$" is performed at most once, during phase $dist(u) + 1$.

Altogether, these observations imply that breadth-first search runs in ***O*(*V* + *E*) *time***. Intuitively, we can think of the vertices in the queue as a "wavefront" expanding monotonically outward from the source vertex s, passing over each vertex and edge of the graph at most once. This expanding wavefront analogy was already proposed by Chin Yang Lee in 1961, inspired by visualizations produced by his implementation of Moore's Algorithm A.

These observations also imply that we can replace the condition "if $dist(v) > dist(u) + 1$" by the (arguably) simpler test "if $dist(v) = \infty$". Then distances play the same role as the marks maintained by other graph-traversal algorithms, which ensure that each vertex is visited only once. Specifically, a vertex is "marked" if and only if its distance label is finite.

But we still need to prove that the final distance labels are correct!

Theorem 8.2. *When BFS ends, $dist(v)$ is the length of the shortest path in G from s to v, for every vertex v.*

Proof: Fix an arbitrary vertex v, and consider an arbitrary path $v_0 \to v_1 \to \cdots \to v_\ell$ in G, where $v_0 = s$ and $v_\ell = v$. I claim that $dist(v_j) \le j$ for each index j; in particular $dist(v) \le \ell$. We can prove this claim by induction on j as follows.

- Trivially $dist(v_0) = dist(s) = 0$.
- For any index $j > 0$, the induction hypothesis implies $dist(v_{j-1}) \le j - 1$. Immediately after we Pull vertex v_{j-1} from the queue, either $dist(v_j) \le dist(v_{j-1}) + 1$ already, or we set $dist(v_j) \leftarrow dist(v_{j-1}) + 1$. In either case, we have $dist(v_j) \le dist(v_{j-1}) + 1 \le j$.

We just proved that $dist(v)$ is at most the length of an *arbitrary* path from s to v; it follows that $dist(v)$ is at most the length of the *shortest* path from s to'v.

A similar induction proof implies that $dist(v)$ is the length of the predecessor path $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$, so this must be the shortest path. □

8.5 Directed Acyclic Graphs: Depth-First Search

Shortest paths are also easy to compute in directed acyclic graphs, even when the edges are weighted, and in particular, even when some edges have negative weight. (We don't have to worry about negative cycles, because by definition, dags don't have *any* cycles!) Indeed, this is a completely standard dynamic programming algorithm.

Let G be a directed graph with weighted edges, and let s be the fixed start vertex. For any vertex v, let $dist(v)$ denote the length of the shortest path in G from s to v. This function satisfies the following simple recurrence:

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

In fact, this identity holds for *all* directed graphs, but it is only a *recurrence* for directed acyclic graphs. If the input graph G contained a cycle, a recursive evaluation of this function would fall into an infinite loop; however, because G is a dag, each recursive call visits an earlier vertex in topological order.

The dependency graph for this recurrence is the reversal of the input graph G: subproblem $dist(v)$ depends on $dist(u)$ if and only if $u \to v$ is an edge in G. Thus, we compute the distance of every in $\boldsymbol{O(V + E)}$ ***time*** by performing a depth-first search in the reversal of G and considering vertices in postorder. Equivalently, we can consider the vertices in the original graph G in topological order, as shown in Figure 8.7.

The resulting dynamic-programming algorithm is another example of Ford's generic relaxation algorithm! To make this connection clearer, we can move the initialization $dist(v)$ outside the main loop and add computation of predecessor pointers, as shown in Figure 8.8. Figure 8.9 shows this algorithm in action.

```
DagSSSP(s):
  for all vertices v in topological order
      if v = s
          dist(v) ← 0
      else
          dist(v) ← ∞
          for all edges u→v
              if dist(v) > dist(u) + w(u→v)          ⟨⟨if u→v is tense⟩⟩
                  dist(v) ← dist(u) + w(u→v)         ⟨⟨relax u→v⟩⟩
```

Figure 8.7. Computing shortest paths in a dag using dynamic programming

```
DagSSSP(s):
  InitSSSP(s)
  for all vertices v in topological order
      for all edges u→v
          if u→v is tense
              Relax(u→v)
```

Figure 8.8. Computing shortest paths in a dag using Ford's algorithm. (These are the same algorithm.)

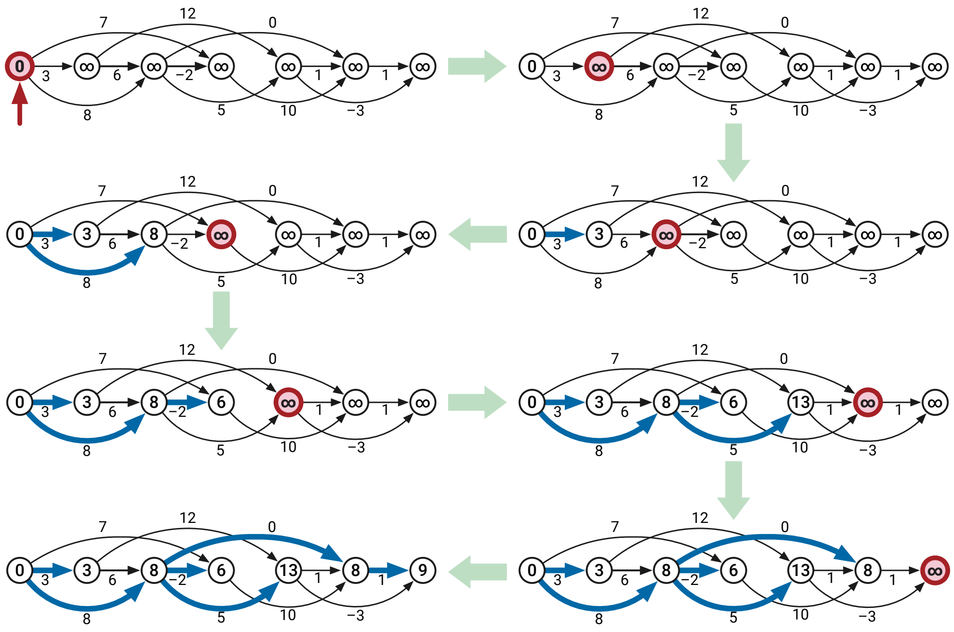


Figure 8.9. Computing shortest paths in a dag, by relaxing **incoming** edges in topological order. In each iteration, bold edges indicate predecessors, and the bold vertex is about to be scanned. Compare with Figure 8.10.

DagSSSP differs from breadth-first search and other instances of Ford's relaxation strategy in one minor respect. Whenever these other shortest-path algorithms consider a vertex, they attempt to relax each of its *outgoing* edges, intuitively *pushing* the wavefront forward from the source; whereas, DagSSSP attempts to relax each of the *incoming* edges of each vertex, intuitively *pulling* the wavefront forward.

However, if we modify DagSSSP to relax outgoing edges instead of incoming edges, we obtain another algorithm that computes shortest paths in dags in ***$O(V+E)$ time*** and that more closely resembles our other shortest-path algorithms.

```
PushDagSSSP(s):
  InitSSSP(s)
  for all vertices u in topological order
      for all outgoing edges u→v
          if u→v is tense
              Relax(u→v)
```

Figure 8.10 shows an execution of this modified algorithm on the same graph as Figure 8.9. The correctness of PushDagSSSP follows immediately from the correctness of Ford's general relaxation strategy, but it's not hard to prove correctness directly, by induction over the vertices in topological order.

8.6 Best-First: Dijkstra's Algorithm

If we replace the FIFO queue in breadth-first search with a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we obtain an algorithm first "published" in 1957 by a team of researchers at the Case Institute of Technology led by Michael Leyzorek, in an annual project report for the Combat Development Department of the US Army Electronic Proving Ground. The same algorithm was independently discovered by Edsger Dijkstra in 1956 (but not published until 1959), again by George Minty some time before 1960, and again by Peter Whiting and John Hillier in 1960. A nearly identical algorithm was also described by George Dantzig in 1958. Although several early sources called it "Minty's algorithm", this approach is now universally known as "Dijkstra's algorithm", in full accordance with Stigler's Law.[7] Pseudocode for this algorithm is shown in Figure 8.11.

An easy induction proof implies that, at all times during the execution of this algorithm, an edge $u\to v$ is tense if and only if vertex u is either in the priority

[7]I will follow this common convention, despite the historical inaccuracy, partly because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz-Dantzig-Dijkstra-Minty-Whiting-Hillier algorithm", and partly because papers that aren't *actually published* don't count.

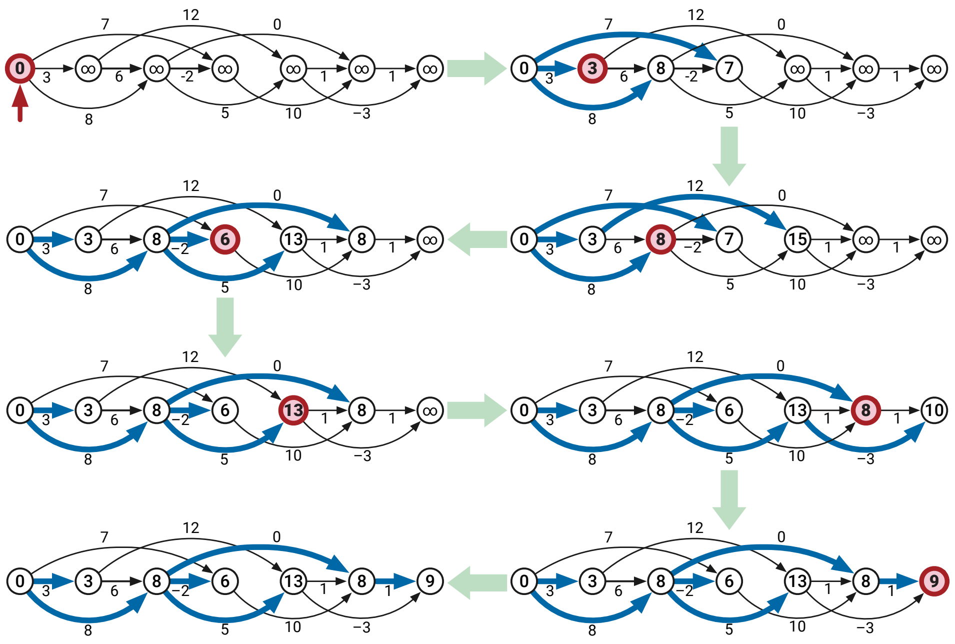


Figure 8.10. Computing shortest paths in a dag, by relaxing **outgoing** edges in topological order. In each iteration, bold edges indicate predecessors, and the bold vertex is about to be scanned. Compare with Figure 8.9.

```
DIJKSTRA($s$):
    INITSSSP($s$)
    INSERT($s, 0$)
    while the priority queue is not empty
        $u \leftarrow$ EXTRACTMIN( )
        for all edges $u \rightarrow v$
            if $u \rightarrow v$ is tense
                RELAX($u \rightarrow v$)
                if $v$ is in the priority queue
                    DECREASEKEY($v, dist(v)$)
                else
                    INSERT($v, dist(v)$)
```

Figure 8.11. Dijkstra's algorithm.

queue or is the vertex most recently Extracted from the priority queue. Thus, Dijkstra's algorithm is an instance of Ford's general strategy, which implies that it correctly computes shortest paths, provided there are no negative cycles in G.

No Negative Edges

Dijkstra's algorithm is particularly well-behaved when the input graph has no negative-weight edges. In this setting, the algorithm intuitively expands a wavefront outward from the source vertex s, passing over vertices in increasing order of their distance from s, similarly to breadth-first search. Figure 8.12 shows the algorithm in action.

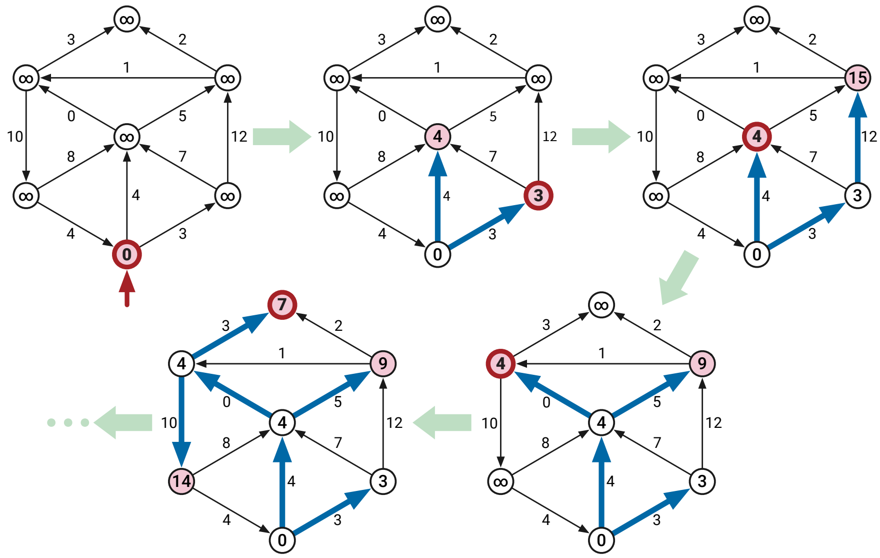


Figure 8.12. The first four iterations of Dijkstra's algorithm on a graph with no negative edges. In each iteration, bold edges indicate predecessors; shaded vertices are in the priority queue; and the bold vertex is about to be scanned. The remaining iterations do not change the distances or the shortest-path tree.

We can derive a self-contained proof of correctness for Dijkstra's algorithm in this setting by formalizing this wavefront intuition. For each integer i, let u_i denote the vertex returned by the ith call to ExtractMin, and let d_i be the value of $dist(u_i)$ just after this Extraction. In particular, we have $u_1 = s$ and $d_1 = 0$. We cannot assume at this point that the vertices u_i are distinct; in principle, the same vertex might be Extracted more than once.

Lemma 8.3. *If G has no negative-weight edges, then for all $i < j$, we have $d_i \le d_j$.*

Proof: Assume G has no negative weight edges. Fix an arbitrary index i; to prove the lemma, it suffices to prove that $d_{i+1} \ge d_i$. There are two cases to consider.

- If G contains the edge $u_i \to u_{i+1}$, and this edge is relaxed during the ith iteration of the main loop, then at the end of the ith iteration, we have $dist(u_{i+1}) = dist(u_i) + w(u_i \to u_{i+1}) \geq dist(u_i)$, because all edge weights are non-negative.
- Otherwise, at the start of the ith iteration, u_{i+1} must already be in the priority queue, and it must have priority $dist(u_{i+1}) \geq dist(u_i)$, because u_i is the vertex returned by ExtractMin. Moreover, $dist(u_{i+1})$ does not change during the ith iteration.

In both cases, we conclude that $d_{i+1} \geq d_i$. The lemma now follows immediately by induction on i. □

Lemma 8.4. *If G has no negative-weight edges, each vertex of G is Extracted from the priority queue at most once.*

Proof: Suppose v is Extracted more than once. Specifically, suppose v is Extracted in the ith iteration of the main loop, reInserted during the jth iteration, and reExtracted during the kth iteration, for some indices $i < j < k$. Then in the notation of the previous proof, we have $v = u_i = u_k$.

The distance label $dist(v)$ never increases. Moreover, $dist(v)$ strictly decreases during the jth iteration, just before v is reInserted. It follows that $d_i > d_k$. Therefore, by the previous lemma, G has at least one negative-weight edge. □

Lemma 8.4 immediately implies that each vertex is scanned at most once, and thus that each edge is relaxed at most once. However, unlike in breadth-first search, each distance label $dist(v)$ can change multiple times. The first time $dist(v)$ changes from ∞, we Insert v into the priority queue; after that, each change to $dist(v)$ is followed by a call to DecreaseKey. After v is Extracted from the priority queue, its distance label never changes.

The rest of the correctness proof is almost identical to breadth-first search.

Theorem 8.5. *If G has no negative-weight edges, then when Dijkstra ends, $dist(v)$ is the length of the shortest path in G from s to v, for every vertex v.*

Proof: Fix an arbitrary vertex v, and consider an arbitrary path $v_0 \to v_1 \to \cdots \to v_\ell$ in G, where $v_0 = s$ and $v_\ell = v$. For any index j, let L_j denote the length of the subpath $v_0 \to v_1 \to \cdots \to v_j$. We prove by induction that $dist(v_j) \leq L_j$ for all j.

- Trivially $dist(v_0) = dist(s) = 0 = L_0$.
- For any index $j > 0$, the induction hypothesis implies $dist(v_{j-1}) \leq L_{j-1}$. Immediately after we Pull vertex v_{j-1} from the queue, either $dist(v_i) \leq dist(v_{j-1}) + w(v_{j-1} \to v_j)$ already, or we set $dist(v_i) \leftarrow dist(v_{j-1}) + w(v_{j-1} \to v_j)$. In either case, we have

$$dist(v_j) \leq dist(v_{j-1}) + w(v_{j-1} \to v_j) \leq L_{j-1} + w(v_{j-1} \to v_j) = L_j.$$

We just proved that $dist(v)$ is at most the length of *every* path from s to v; it follows that $dist(v)$ is at most the length of the *shortest* path from s to v.

On the other hand, a similar induction proof implies that $dist(v)$ is the length of the predecessor path $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$. □

It remains only to bound the algorithm's running time. Altogether DIJKSTRA performs at most E DECREASEKEY operations, and at most V INSERT and EXTRACTMIN operations. Thus, if we implement the underlying priority queue using a standard binary heap, which supports each operation in $O(\log V)$ time, DIJKSTRA runs in ***$O(E \log V)$ time***.[8]

If we know in advance that our input graphs will *never* have negative edges, we can simplify Dijkstra's algorithm slightly, by INSERTing every vertex into the priority queue in the initialization phase, and then only calling DECREASEKEY in the main loop, as shown in Figure 8.13. This is the version of Dijkstra's algorithm presented by most algorithms textbooks, Wikipedia, and even Dijkstra's original paper; it's also the version of Dijkstra's algorithm that I described as "best-first search" in Chapter 5.

```
NONNEGATIVEDIJKSTRA($s$):
    INITSSSP($s$)
    for all vertices $v$
        INSERT($v, dist(v)$)
    while the priority queue is not empty
        $u \leftarrow$ EXTRACTMIN( )
        for all edges $u \to v$
            if $u \to v$ is tense
                RELAX($u \to v$)
                DECREASEKEY($v, dist(v)$)
```

Figure 8.13. Dijkstra's algorithm very slightly simplified for graphs without negative edges. Differences from DIJKSTRA are bold red.

♥Negative Edges

However, NONNEGATIVEDIJKSTRA does *not* correctly compute shortest paths in graphs with negative edges. Moreover, even when all edge weights are

[8] Shortest-path papers from the 1950s never mentioned priority queues. Dijkstra proposed a brute-force scan of all vertices on the wavefront at every iteration; his original algorithm runs in ***$O(V^2)$ time***, which is actually faster than the binary-heap implementation when $E = \Omega(V^2)$! Minty proposed a brute-force scan of all *edges* $u \to v$ such that $dist(u)$ is finite but $dist(v)$ is not; thus, his original algorithm runs in $O(VE)$ time. The use of a priority queue, implemented as a binary heap, to obtain near-linear running time was proposed by Donald Johnson in 1977. The running time can be improved to $\boldsymbol{O(E + V \log V)}$ using a more complex priority queue data structure called a *Fibonacci heaps*. There are even faster algorithms, using even more sophisticated priority queues, for the special case of integer edge weights.

positive, NonnegativeDijkstra is no faster than Dijkstra (either in theory or in practice). For both of these reasons, I think Dijkstra is more deserving of the name "Dijkstra's algorithm" than NonnegativeDijkstra. Even Edsger Dijkstra would have agreed that a correct algorithm that is sometimes (and in practice, rarely) slow is better than a fast algorithm that doesn't always work!

Unfortunately, when the input graph has negative edges, the familiar "expanding wavefront" intuition is no longer accurate. The same vertex can be Extracted multiple times; the same edge can be relaxed multiple times; and distances might not be discovered in increasing order. Figure 8.15 shows an example execution where the top left vertex is Extracted six times, and the top three edges are each relaxed twice.

For graphs without negative cycles, but no other restrictions on edge weights, the worst-case running time of Dijkstra is actually exponential. Figure 8.14 shows particularly simple family of graphs (due to Douglas Shier and Christoph Witzgall) that forces Dijkstra to perform $\Theta(2^{V/2})$ relaxations.[9] A more complex family of graphs (which I'll leave as an exercise) forces $\Theta(2^V)$ relaxations, which is the worst possible. *In practice*, however, Dijkstra's algorithm is usually fast even for graphs with negative edges.

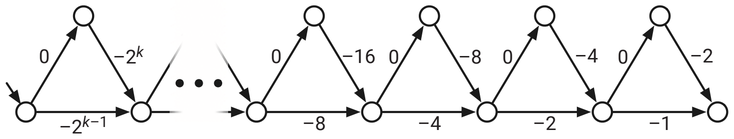


Figure 8.14. A directed graph with negative edges that forces Dijkstra to run in exponential time.

8.7 Relax ALL the Edges: Bellman-Ford

The simplest implementation of Ford's generic shortest-path algorithm was first sketched by Alfonso Shimbel in 1954, described in more detail by Edward Moore in 1957, and independently rediscovered by Max Woodbury and George Dantzig in 1957, by Richard Bellman in 1958, and by George Minty in 1958. (Neither Woodbury and Dantzig nor Minty published their algorithms.) In full compliance with Stigler's Law, the algorithm is almost universally known as ***Bellman-Ford***,[10] because Bellman explicitly used Ford's 1956 formulation of

[9]Amusingly, Shier and Witzgall's example is a dag with only $O(V)$ edges, which implies that shortest paths can be computed in only $O(V)$ time, even if we *didn't* already notice that the zig-zag path along the top *is* the shortest path tree.

[10]I will follow this common convention, despite the historical inaccuracy, partly because I don't think anyone really wants read about the "Shimbel / Moore / Woodbury-Dantzig / Bellman-Ford / Kalaba / Minty algorithm", and partly because I'm tired of people looking at me funny when I talk about "Shimbel's algorithm".

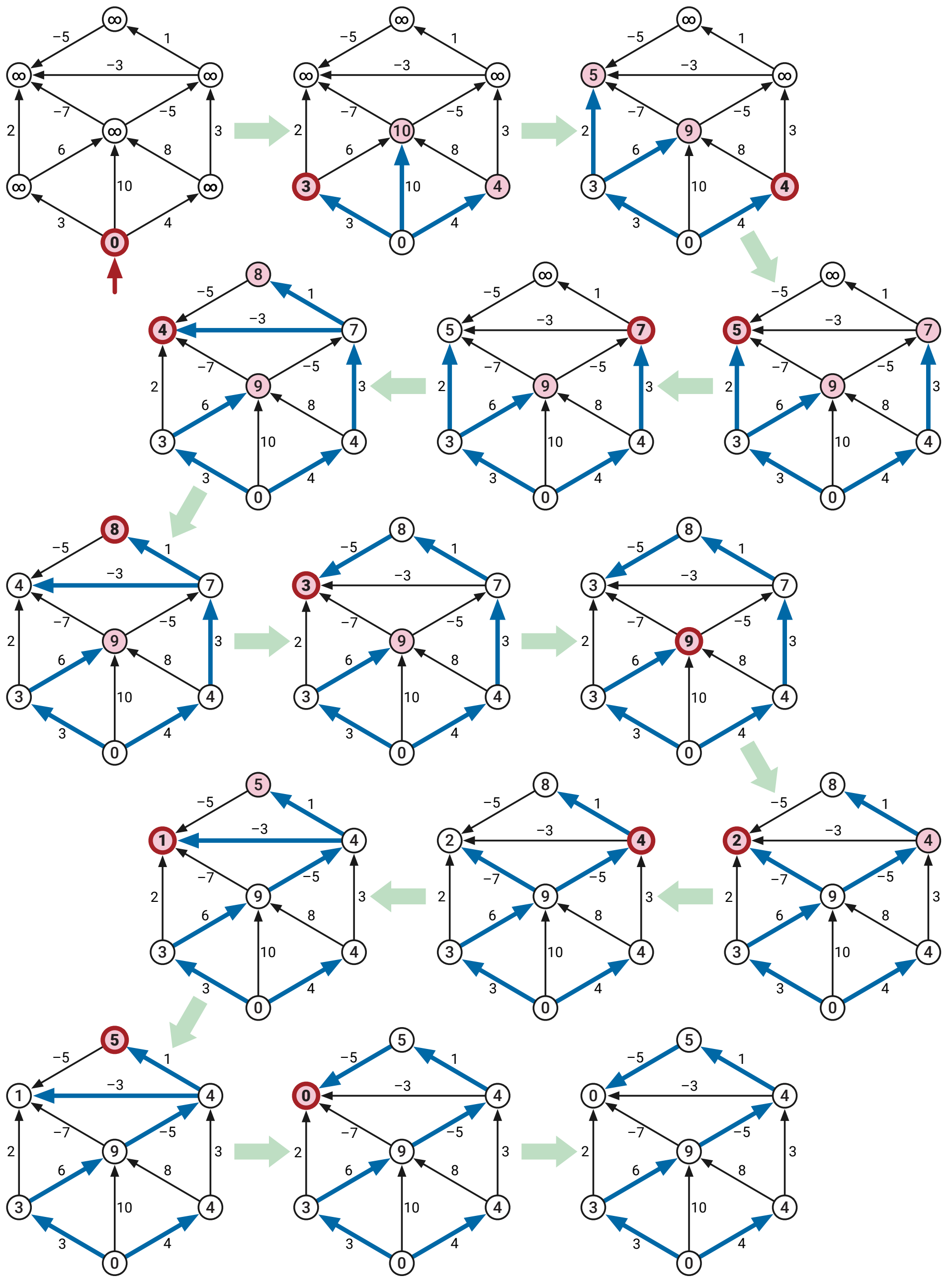


Figure 8.15. A complete run of Dijkstra's algorithm on a graph with negative edges. At each iteration, bold edges indicate predecessors; shaded vertices are in the priority queue; and the bold vertex is the next to be scanned. Compare with Figure 8.17.

relaxing edges, although some authors refer to "Bellman-Kalaba"[11] and a few early sources refer to "Bellman-Shimbel".

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 $u \to v$
          if $u \to v$ is tense
              RELAX($u \to v$)
```

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 $v \ne s$.

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).

[11]This name is most likely a reference to Richard Bellman and Robert Kalaba's 1965 monograph on dynamic programming and control theory, which describes Bellman's algorithm. Bellman and Kalaba also published an extension of Bellman's algorithm in 1960 that computes kth shortest paths, for any constant k.

[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.

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. 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 $u \to v$
          if $u \to v$ is tense
              RELAX($u \to v$)
  for every edge $u \to v$
      if $u \to v$ 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]

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

Following our earlier analysis of breadth-first search, I'll introduce a "token" ✠ to break the execution of the algorithm into phases. Just like breadth-first search, 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.

```
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.

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 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

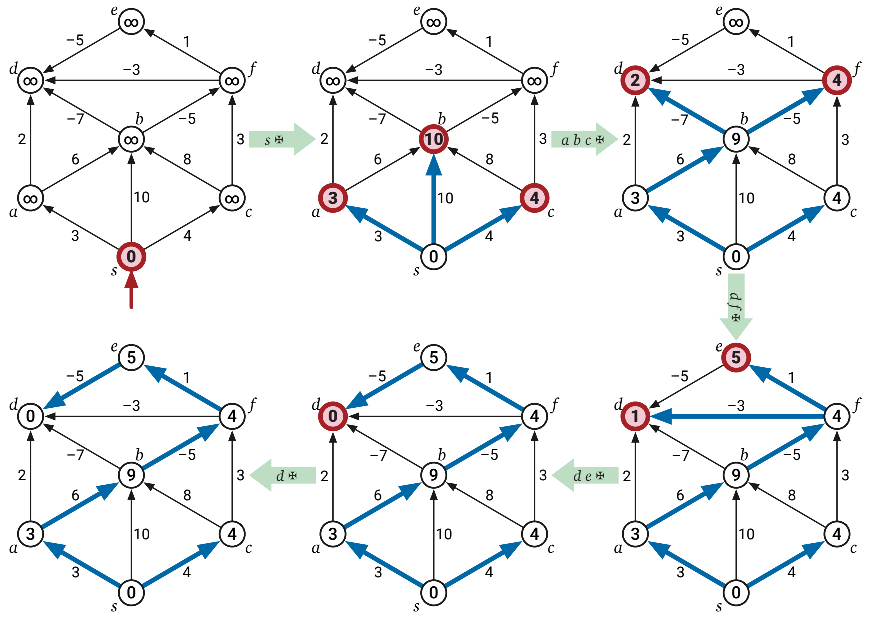


Figure 8.17. A complete run of Moore's algorithm on a directed graph with negative edges. Nodes are pulled from the queue in the order s ✠ $a\ b\ c$ ✠ $d\ f$ ✠ $d\ e$ ✠ d ✠ ✠, where ✠ is the end-of-phase token. At the start of each phase, bold edges indicate predecessors, and shaded vertices are in the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

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.

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:

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

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:

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

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($s$)
  $dist[0,s] \leftarrow 0$
  for every vertex $v \ne s$
      $dist[0,v] \leftarrow \infty$
  for $i \leftarrow 1$ to $V-1$
      for every vertex $v$
          $dist[i,v] \leftarrow dist[i-1,v]$
          for every edge $u \to v$
              if $dist[i,v] > dist[i-1,u] + w(u \to v)$
                  $dist[i,v] \leftarrow dist[i-1,u] + w(u \to v)$
```

We can transform this dynamic programming algorithm into our original formulation of BELLMANFORD through a short series of minor optimizations. 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)
  $dist[0,s] \leftarrow 0$
  for every vertex $v \neq s$
      $dist[0,v] \leftarrow \infty$
  for $i \leftarrow 1$ to $V-1$
      for every vertex $v$
          $dist[i,v] \leftarrow dist[i-1,v]$
      for every edge $u \to v$
          if $dist[i,v] > dist[i-1,u] + w(u \to v)$
              $dist[i,v] \leftarrow dist[i-1,u] + w(u \to v)$
```

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)
  $dist[0,s] \leftarrow 0$
  for every vertex $v \neq s$
      $dist[0,v] \leftarrow \infty$
  for $i \leftarrow 1$ to $V-1$
      for every vertex $v$
          $dist[i,v] \leftarrow dist[i-1,v]$
      for every edge $u \to v$
          if $dist[i,v] > dist[i,u] + w(u \to v)$        ⟨⟨not $i-1$!⟩⟩
              $dist[i,v] \leftarrow dist[i,u] + w(u \to v)$  ⟨⟨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 $dist[\cdot,\cdot]$ 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)
  $dist[s] \leftarrow 0$
  for every vertex $v \neq s$
      $dist[v] \leftarrow \infty$
  for $i \leftarrow 1$ to $V-1$
      for every edge $u \to v$
          if $dist[v] > dist[u] + w(u \to v)$
              $dist[v] \leftarrow dist[u] + w(u \to v)$
```

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.

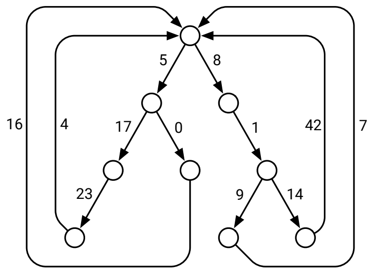
Exercises

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($s$)
  for $i \leftarrow 1$ to whatever, man, I don't care
      $e_i \leftarrow$ any edge in $G$
      if $e_i$ is tense
          RELAX($e_i$)
```

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 $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_\ell$, where $v_0 = s$, define a *subsequence* of the edges $e_1, e_2, e_3, \ldots$ examined by FELLMANBORED, then we have $dist(v_\ell) \leq \sum_{i=1}^{\ell} w(v_{i-1} \rightarrow v_i)$. *[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 $pred(v) \rightarrow v$, and let X denote the set of all currently *tense* 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: Binary counter.]*

♥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. Suppose you are given a directed graph G in which ***every edge has negative weight***, and a source vertex s. Describe and analyze an efficient algorithm that computes the shortest-path distances from s to every other vertex in G. Specifically, for every vertex t:

- If t is not reachable from s, your algorithm should report $dist(t) = \infty$.
- If G has a cycle that is reachable from s, and t is reachable from that cycle, then the shortest-path distance from s to t is not well-defined, because there are paths (formally, walks) from s to t of arbitrarily large negative length. In this case, your algorithm should report $dist(t) = -\infty$.
- If neither of the two previous conditions applies, your algorithm should report the correct shortest-path distance from s to t.

12. 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 desirable 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.

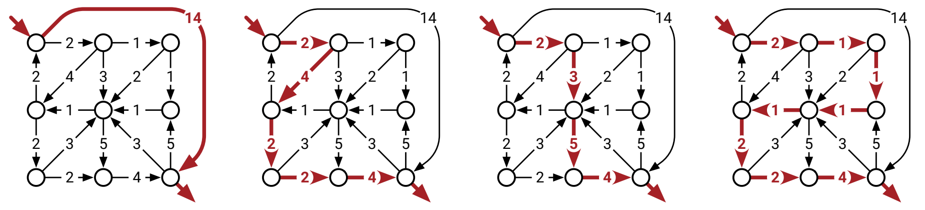


Figure 8.18. Four (of many) equal-length shortest paths. The first path is the "best" shortest path.

13. 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?]*

14. 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.

15. 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, hand-pulled, direct-trade, organic, shade-grown, single-origin espresso, unadulterated by milk or sugar, thank you *very* much. 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 hand-pulled direct-trade 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 acceptable 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.

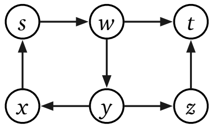
    (b) You discover that by wearing a more expensive fedora, you can increase the distance L that you can bike between espresso shots. Describe and analyze and algorithm to find the minimum value of L that allows you to bike from the Giggleplex to Gigglesburg without suffering a caffeine-withdrawal migraine.

(c) When you report to your supervisor (whom Giggle recently hired away from their competitor Ünter) 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.

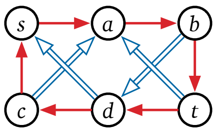
16. Suppose you are given a directed graph $G = (V, E)$ with non-negatively weighted edges and two vertices s and t. Describe and analyze an algorithm to find the shortest walk in G from s to t (possibly repeating vertices and/or edges) whose number of edges is divisible by 3.

For example, given the graph shown below, with the indicated vertices s and t, and with all edges having weight 1, your algorithm should return 6, which is the length of the walk $s \to w \to y \to x \to s \to w \to t$ has length 6.



17. Suppose you are given a directed graph G with non-negatively weighted edges, where some edges are red and the remaining edges are blue. Describe an algorithm to find the shortest walk in G from one vertex s to another vertex t in which no three consecutive edges have the same color. That is, if the walk contains two red edges in a row, the next edge must be blue, and if the walk contains two blue edges in a row, the next edge must be red.

For example, given the following graph as input, where every red edge has weight 1 and every blue edge has weight 2, your algorithm should return the integer 9, because the shortest legal walk from s to t is $s \to a \to b \Rightarrow d \to c \Rightarrow a \to b \to c$.



18. Consider a directed graph G, where each edge has a non-negative weight, and each edge is colored either red, white, or blue. A walk in G is called a *French flag walk* if its sequence of edge colors is red, white, blue, red, white, blue, and so on. More formally, a walk $v_0 \to v_1 \to \cdots \to v_k$ is a French flag walk if, for every integer i, the edge $v_i \to v_{i+1}$ is red if $i \bmod 3 = 0$, white if $i \bmod 3 = 1$, and blue if $i \bmod 3 = 2$.

    Describe an algorithm to find the *shortest* French flag walks from one starting vertex s to every other vertex in G.

19. There are n galaxies connected by m intergalactic teleport-ways. Each teleport-way joins two galaxies and can be traversed in both directions. Also, each teleport-way e has an associated cost of $c(e)$ dollars, where $c(e)$ is a positive integer. A teleport-way can be used multiple times, but the toll must be paid every time it is used.

    Judy wants to travel from galaxy s to galaxy t as cheaply as possible. However, she wants the total cost to be a multiple of five dollars, because carrying small change is not pleasant either.

    (a) Describe and analyze an algorithm to compute the minimum total cost of traveling from galaxy s to galaxy t, subject to the restriction that the total cost is a multiple of five dollars.

    (b) Solve part (a), but now assume that Judy has a coupon that allows her to use exactly one teleport-way for free.

20. After moving to a new city, you decide to choose a walking route from your home to your new office. To get a good daily workout, your route must consist of an uphill path (for exercise) followed by a downhill path (to cool down), or just an uphill path, or just a downhill path. (You'll walk the same path home, so you'll get exercise one way or the other.) But you also want the *shortest* path that satisfies these conditions, so that you actually get to work on time.

    Your input consists of an undirected graph G, whose vertices represent intersections and whose edges represent road segments, along with a start vertex s and a target vertex t. Every vertex v has an associated value $h(v)$, which is the height of that intersection above sea level, and each edge uv has an associated value $\ell(uv)$, which is the length of that road segment.

    (a) Describe and analyze an algorithm to find the shortest uphill–downhill walk from s to t. Assume all vertex heights are distinct.

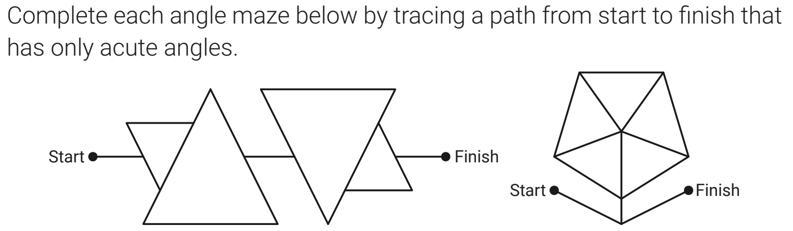
    (b) Now suppose we allow some or all vertex heights to be equal. Describe and analyze an algorithm to find the shortest "uphill then downhill" walk from s to t; you may use flat edges in both the "uphill" and "downhill" portions of your walk.

(c) Finally, suppose you discover that there is no path from s to t with the structure you want. Describe an algorithm to find a path from s to t that alternates between "uphill" and "downhill" subpaths as few times as possible, and has minimum length among all such paths.

21. After graduating from Sham-Poobanana University you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

    Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights.

22. In Exercise 20 from Chapter 5, you designed an algorithm to decide whether a given *acute-angle maze* is solvable. In this problem, you will design algorithms to find the *shortest* walk through a given acute-angle maze, for two different definitions of "length".


Complete each angle maze below by tracing a path from start to finish that has only acute angles.

Your input is a connected undirected graph G whose vertices are points in the plane and whose edges are line segments. Edges do not intersect, except at their endpoints. For example, a drawing of the letter X would have five vertices and four edges; the first maze above has 14 vertices and 21 edges. You are also given two vertices Start and Finish.

A walk from Start to Finish in G is *valid* if it contains only acute angles, or more formally, for any two consecutive edges $u \rightarrow v \rightarrow w$, either $\angle uvw = \pi$ or $0 < \angle uvw < \pi/2$. Assume you can determine in $O(1)$ time whether the angle between two given segments is straight, obtuse, right, or acute.

(a) Describe an algorithm to compute a valid walk from Start to Finish that traverses as few segments as possible. (If your walk traverses the same segment twice, count it twice.)

(b) Describe an algorithm to compute a valid walk from Start to Finish that makes as few turns as possible. *[Hint: This is **not** the same as part (a).]*

(c) Describe an algorithm to compute a valid walk from Start to Finish whose total Euclidean length is as small as possible. (Assume you can also compute the length of any segment in $O(1)$ time.)

23. After a grueling midterm at the See-Bull Center for Fake News Detection, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Sham-Poobanana. Unfortunately, no single bus visits both the See-Bull Center and your home; you must change buses at least once. There are exactly b different buses. Each bus starts at 12:00:01AM, makes exactly n stops, and finally stops running at 11:59:59PM. Buses always run exactly on schedule, and you have an accurate watch. Finally, you are far too tired to walk between bus stops.

(a) Describe and analyze an algorithm to determine the sequence of bus rides that gets you home as early as possible. Your goal is to minimize your *arrival time*, not the time you spend traveling.

(b) Oh, no! The midterm was held on Halloween, and the streets are infested with zombies! The Sham-Poobanana Mass Transit District doesn't have the funding to add additional buses or install zombie-proof bus stops, especially for only one night a year. Describe and analyze an algorithm to determine a sequence of bus rides that minimizes *the total time you spend waiting at bus stops*; you don't care how late you get home or how much time you spend on buses. (Assume you can wait inside the See-Bull Center until your first bus is just about to leave.)

24. The first morning after returning from a glorious spring break, Alice wakes to discover that her car won't start, so she has to get to her classes at Sham-Poobanana University by public transit. She has a complete transit schedule for Poobanana County. The bus routes are represented in the schedule by a directed graph G, whose vertices represent bus stops and whose edges represent bus routes between those stops. For each edge $u{\to}v$, the schedule records three positive real numbers:

- $\ell(u{\to}v)$ is the length of the bus ride from stop u to stop v (in minutes)
- $f(u{\to}v)$ is the first time (in minutes past 12am) that a bus leaves stop u for stop v.
- $\Delta(u{\to}v)$ is the time between successive departures from stop u to stop v (in minutes).

Thus, the first bus for this route leaves u at time $f(u{\to}v)$ and arrives at v at time $f(u{\to}v)+\ell(u{\to}v)$, the second bus leaves u at time $f(u{\to}v)+\Delta(u{\to}v)$ and arrives at v at time $f(u{\to}v)+\Delta(u{\to}v)+\ell(u{\to}v)$, the third bus leaves u at time

$f(u\to v) + 2 \cdot \Delta(u\to v)$ and arrives at v at time $f(u\to v) + 2 \cdot \Delta(u\to v) + \ell(u\to v)$, and so on.

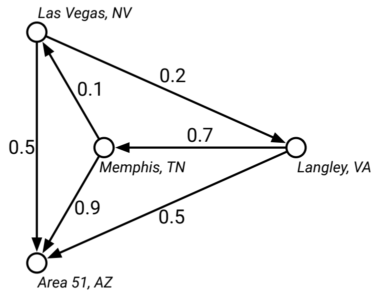
Alice wants to leaves from stop s (her home) at a certain time and arrive at stop t (The See-Bull Center) as quickly as possible. If Alice arrives at a stop on one bus at the exact time that another bus is scheduled to leave, she can catch the second bus. Because she's a student at SPU, Alice can ride the bus for free, so she doesn't care how many times she has to change buses.

Describe and analyze an algorithm to find the earliest time Alice can reach her destination. Your input consists of the directed graph $G = (V, E)$, the vertices s and t, the values $\ell(e), f(e), \Delta(e)$ for each edge $e \in E$, and Alice's starting time (in minutes past 12am).

[Hint: In this rare instance, it may be easier to modify the algorithm, instead of modifying the input graph.]

25. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t. You may assume that all necessary arithmetic operations can be performed in $O(1)$ time.



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.)

♣26. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out. . . while. . . you. . . ", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to *immediately* leave the park as fast as possible.

Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. Be sure to account for *both* the vertex probabilities *and* the edge probabilities. *[Hint: Even without the vertex probabilities, this is not the same as the previous problem!]*