Well, ya turn left by the fire station in the village and take the old post road by the reservoir and. . . no, that won't do.

Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until. . . no, that won't work either.

East Millinocket, ya say? Come to think of it, you can't get there from here.

— Robert Bryan and Marshall Dodge,
Bert and I and Other Stories from Down East (1961)

Hey farmer! Where does this road go?
Been livin' here all my life, it ain't gone nowhere yet.

Hey farmer! How do you get to Little Rock?
Listen stranger, you can't get there from here.

Hey farmer! You don't know very much do you?
No, but I ain't lost.

— Michelle Shocked, “Arkansas Traveler" (1992)

13 Shortest Paths

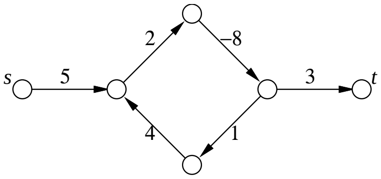
13.1 Introduction

Given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, a *source* s and a *target* t, we want to find the shortest directed path from s to t. In other words, we want to find the path p starting at s and ending at t minimizing the function

$$w(p) = \sum_{e \in p} w(e).$$

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?', we 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 since the driving times along the same road might be different in different directions.[2]

Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, since the presence of a negative cycle might mean that there is no shortest path. In general, 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. If there is a negative cycle between s and t, then se can always find a shorter path by going around the cycle one more time.



There is no shortest path from s to t.

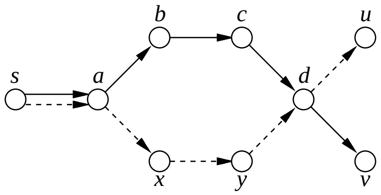
Every algorithm known for solving this problem actually solves (large portions of) the following more general *single source shortest path* or *SSSP* problem: find the shortest path from the source vertex s

[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 both live in Urbana now.

[2] In 1999 and 2000, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.

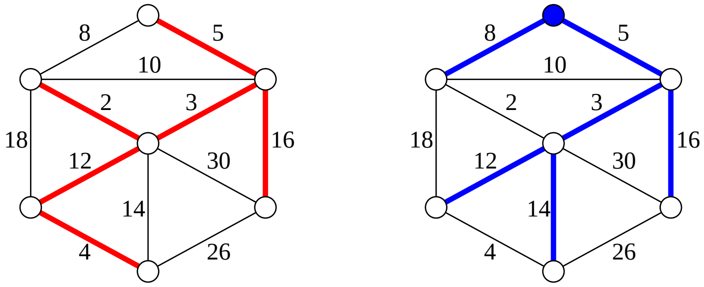
to *every* other vertex in the graph. In fact, the problem is usually solved by finding a *shortest path tree* rooted at s that contains all the desired shortest paths.

It's not hard to see that if shortest paths are unique, then they form a tree. To prove this, it's enough to observe that any subpath of a shortest path is also a shortest path. If there are multiple shortest paths to the same vertices, we can always choose one path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths so that the two paths only diverge once.



If $s{\to}a{\to}b{\to}c{\to}d{\to}v$ and $s{\to}a{\to}x{\to}y{\to}d{\to}u$ are both shortest paths,
then $s{\to}a{\to}b{\to}c{\to}d{\to}u$ is also a shortest path.

Shortest path trees and minimum spanning trees are usually very different. For one thing, there is only one minimum spanning tree, but in general, there is a different shortest path tree for every source vertex. Moreover, in general, *all* of these shortest path trees are different from the minimum spanning tree.



A minimum spanning tree and a shortest path tree (rooted at the topmost vertex) of the same graph.

All of the algorithms described in this lecture also work for undirected graphs, with some slight modifications. Most importantly, we must specifically prohibit alternating back and forth across the same undirected negative-weight edge. (Our unmodified algorithms would interpret any negative-weight edge as a negative cycle of length 2.)

To emphasize the direction, I will consistently use the nonstandard notation $u{\to}v$ to denote a directed edge from u to v.

13.2 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, there are several different SSSP algorithms, but they are all special cases of the a single generic algorithm, first proposed by Ford in 1956, and independently by Dantzig in 1957.[3] 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.

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

- *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; they play the same role as the 'parent' pointers in our generic graph traversal algorithm. We already know that $dist(s) = 0$ and $pred(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $dist(v) = \infty$ and $pred(v) = \text{NULL}$ to indicate that we do not know of *any* path from s to v.

We call an edge $u \to v$ *tense* if $dist(u) + w(u \to v) < dist(v)$. If $u \to v$ is tense, then the tentative shortest path $s \rightsquigarrow v$ is incorrect, since the path $s \rightsquigarrow u \to v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

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

If there are no tense edges, our algorithm is finished, and we have our desired shortest path tree.

The correctness of the relaxation algorithm follows directly from three simple claims:

1. If $dist(v) \neq \infty$, then $dist(v)$ is the total weight of the predecessor chain ending at v:
$$s \to \cdots \to pred(pred(v)) \to pred(v) \to v.$$
This is easy to prove by induction on the number of relaxation steps. (Hint, hint.)

2. If the algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for *any* path $s \rightsquigarrow v$. This is easy to prove by induction on the number of edges in the path $s \rightsquigarrow v$. (Hint, hint.)

3. The algorithm halts if and only if there is no negative cycle reachable from s. The 'only if' direction is easy—if there is a reachable negative cycle, then after the first edge in the cycle is relaxed, the cycle *always* has at least one tense edge. The 'if' direction follows from the fact that every relaxation step reduces either the number of vertices with $dist(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by the difference between two edge weights.

Actually proving the first two claims above is not as straightforward as I make it sound; there are some unfortunate subtleties. It's *much* easier to prove the following weaker claims, which also imply correctness:

- For every vertex v, *dist*(v) is always greater than or equal to the shortest-path distance from s to v. (Induction on the number of relaxation steps.)
- If no edge is tense, then for every vertex v, *dist*(v) is the shortest-path distance from s to v. (Induction on the number of edges in the true shortest path; see Bellman-Ford. If the edge weights are non-negative, induction on the rank of the shortest-path distance also works; see Dijkstra!)
- If no edge is tense, then for every vertex v, the path $s \to \cdots \to pred(pred(v)) \to pred(v) \to v$ is a shortest path from s to v. (Induction on the number of edges in the path; why does *pred*(v) have its current value?)
- The algorithm halts if there are no negative cycles reachable from s; see above.

I haven't said anything about how we detect which edges can be relaxed, or in what order we relax them. In order to make this easier, we can refine the relaxation algorithm slightly, into something closely resembling the generic graph traversal algorithm. We maintain a 'bag' of vertices, initially containing just the source vertex s. Whenever we take a vertex u out of the bag, we scan all of its outgoing edges, looking for something to relax. Whenever we successfully relax an edge $u \to v$, we put v into the bag. Unlike our generic graph traversal algorithm, the same vertex might be visited many times.

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

```
GenericSSSP($s$):
  InitSSSP($s$)
  put $s$ in the bag
  while the bag is not empty
      take $u$ from the bag
      for all edges $u \to v$
          if $u \to v$ is tense
              Relax($u \to v$)
              put $v$ in the bag
```

Just as with graph traversal, using different data structures for the 'bag' gives us different algorithms. There are three obvious choices to try: a stack, a queue, and a heap. Unfortunately, if we use a stack, we have to perform $\Theta(2^V)$ relaxation steps in the worst case! (Proving this is a good homework problem.) The other two possibilities are much more efficient.

13.3 Dijkstra's Algorithm

If we implement the bag as a heap, where the key of a vertex v is $dist(v)$, we obtain an algorithm first 'published'[4] by Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, and Seitz in 1957, and then later independently rediscovered by Edsger Dijkstra in 1959. A very similar algorithm was also described by Dantzig in 1958.

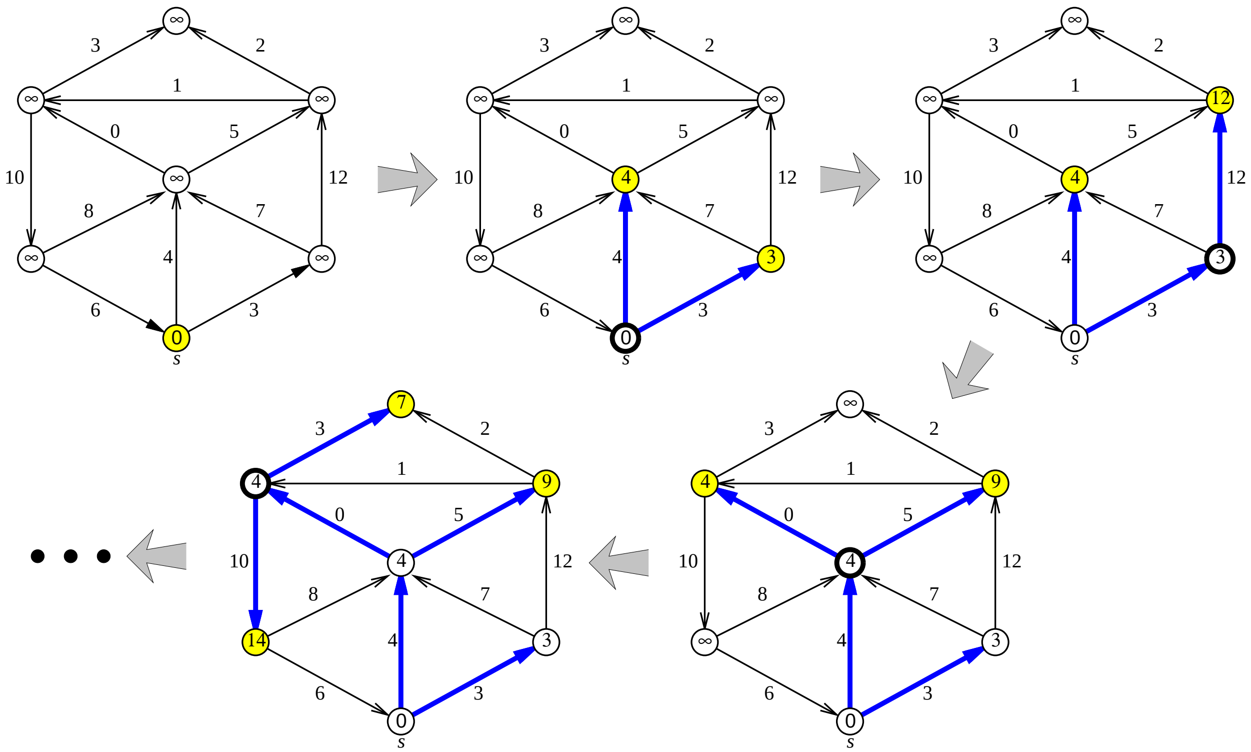
Dijkstra's algorithm, as it is universally known[5], is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s. It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once. Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DecreaseKey operation every time an edge is relaxed. Thus, the algorithm performs at most E DecreaseKeys. Similarly, there are at most V Insert and ExtractMin operations. Thus, if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $\boldsymbol{O(E + V \log V)}$; if we use a regular binary heap, the running time is $O(E \log V)$.

This analysis assumes that no edge has negative weight. Dijkstra's algorithm (in the form I'm presenting here) is still *correct* if there are negative edges[6], but the worst-case running time could be exponential. (Proving this unfortunate fact is a good homework problem.)

[4] in the first annual report on a research project performed for the Combat Development Department of the Army Electronic Proving Ground

[5] I will follow this common convention, despite the historical inaccuracy, because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz algorithm".

[6] Many textbooks present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges.



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

13.4 The A^* Heuristic

A slight generalization of Dijkstra's algorithm, commonly known as the A^* algorithm, is frequently used to find a shortest path from a single source node s to a single target node t. A^* uses a black-box function $\textsc{GuessDistance}(v,t)$ that returns an estimate of the distance from v to t. The only difference between Dijkstra and A^* is that the key of a vertex v is $\mathit{dist}(v) + \textsc{GuessDistance}(v,t)$.

The function $\textsc{GuessDistance}$ is called *admissible* if $\textsc{GuessDistance}(v,t)$ never overestimates the actual shortest path distance from v to t. If $\textsc{GuessDistance}$ is admissible and the actual edge weights are all non-negative, the A^* algorithm computes the actual shortest path from s to t at least as quickly as Dijkstra's algorithm. The closer $\textsc{GuessDistance}(v,t)$ is to the real distance from v to t, the faster the algorithm. However, in the worst case, the running time is still $O(E + V \log V)$.

The heuristic is especially useful in situations where the actual graph is not known. For example, A^* can be used to solve many puzzles (15-puzzle, Freecell, Shanghai, Sokoban, Atomix, Rush Hour, Rubik's Cube, ...) and other path planning problems where the starting and goal configurations are given, but the graph of all possible configurations and their connections is not given explicitly.

13.5 Shimbel's Algorithm ('Bellman-Ford')

If we replace the heap in Dijkstra's algorithm with a queue, we get an algorithm that was first published by Shimbel in 1955, then independently rediscovered by Moore in 1957, by Woodbury and Dantzig in 1957, and by Bellman in 1958. Since Bellman used the idea of relaxing edges, which was first proposed by Ford in 1956, this is usually called the 'Bellman-Ford' algorithm. Shimbel's algorithm is efficient even if there are negative edges, and it can be used to quickly detect the presence of negative cycles. If there are no negative edges, however, Dijkstra's algorithm is faster. (In fact, in practice, Dijkstra's algorithm is often faster even for graphs with negative edges.)

> Is describing this algorithm with a queue really helpful, or just confusing? After all, it's just a straightforward dynamic programming algorithm. And the structure of the algorithm is very close to the proof of correctness of the generic algorithm (induction on the number of edges in the shortest path).

The easiest way to analyze the algorithm is to break the execution into *phases*, by introducing an imaginary *token*. Before we even begin, we insert the token into the queue. The current phase ends when we take the token out of the queue; we begin the next phase by reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. A simple inductive argument (hint, hint) implies the following invariant:

> At the end of the ith phase, for every vertex v, $dist(v)$ is less than or equal to the length of the shortest path $s \leadsto v$ consisting of i or fewer edges.

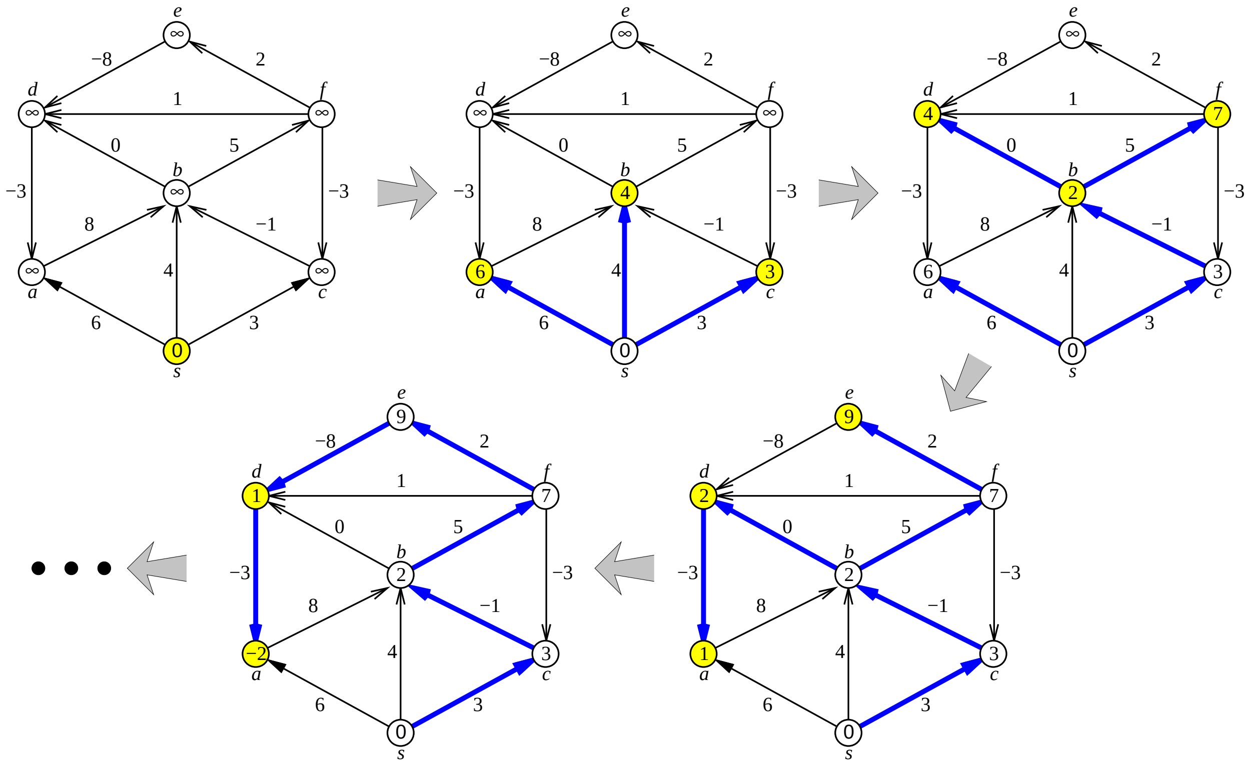
Since a shortest path can only pass through each vertex once, either the algorithm halts before the Vth phase, or the graph contains a negative cycle. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the running time of a single phase is $O(E)$. Thus, the overall running time of Shimbel's algorithm is $\boldsymbol{O(VE)}$.

Once we understand how the phases of Shimbel's algorithm behave, we can simplify the algorithm considerably. Instead of using a queue to perform a partial breadth-first search of the graph in each phase, we can simply scan through the adjacency list directly and try to relax every edge in the graph.

```
ShimbelSSSP($s$)
  InitSSSP($s$)
  repeat $V$ 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!'
```

This is how most textbooks present the 'Bellman-Ford' algorithm.[7] The $O(VE)$ running time of this version of the algorithm should be obvious, but it may not be clear that the algorithm is still correct. To prove correctness, we just have to show that our earlier invariant holds; as before, this can be proved by induction on i.

[7] In fact, this is closer to the description that Shimbel and Bellman used. Bob Tarjan recognized in the early 1980s that Shimbel's algorithm is equivalent to Dijkstra's algorithm with a queue instead of a heap.



Four phases of Shimbel's algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order $s \diamond a\ b\ c \diamond d\ f\ b \diamond a\ e\ d \diamond d\ a \diamond \diamond$, where $\diamond$ is the token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

13.6 Greedy Relaxation?

Here's another algorithm that fits our generic framework, but which I've never seen analyzed.

$$\boxed{\text{Repeatedly relax the tensest edge.}}$$

Specifically, let's define the 'tension' of an edge $u \to v$ as follows:

$$\mathit{tension}(u \to v) = \max\{0, \mathit{dist}(v) - \mathit{dist}(u) - w(u \to v)\}$$

(This is defined even when $\mathit{dist}(v) = \infty$ or $\mathit{dist}(u) = \infty$, as long as we treat ∞ just like some indescribably large but finite number.) If an edge has zero tension, it's not tense. If we relax an edge $u \to v$, then $\mathit{dist}(v)$ decreases $\mathit{tension}(u \to v)$ and $\mathit{tension}(u \to v)$ becomes zero.

Intuitively, we can keep the edges of the graph in some sort of heap, where the key of each edge is its tension. Then we repeatedly pull out the tensest edge $u \to v$ and relax it. Then we need to recompute the tension of other edges adjacent to v. Edges leaving v possibly become more tense, and edges coming into v possibly become less tense. So we need a heap that efficiently supports the operations Insert, ExtractMax, IncreaseKey, and DecreaseKey.

If there are no negative cycles, this algorithm eventually halts with a shortest path tree, but how quickly? Can the same edge be relaxed more than once, and if so, how many times? Is it faster if all the edge weights are positive? Hmm.... This sounds like a good extra credit problem![8]

[8] I first proposed this bizarre algorithm in 1998, the very first time I taught an algorithms class. As far as I know, nobody has even seriously attempted an analysis. Or maybe it *has* been analyzed, but it requires an exponential number of relaxation steps in the worst case, so nobody's ever bothered to publish it.

Exercises

1. This question asks you to fill in the remaining proof details for Ford's generic shortest-path algorithm: while at least one edge is tense, relax an arbitrary tense edge. You may assume that the input graph does not contain a negative cycle, and that all shortest paths in the input graph are unique.

   (a) Prove that the generic shortest-path algorithm halts.

   (b) Prove that after *every* call to RELAX, for every vertex v, either $\mathit{dist}(v) = \infty$ or $\mathit{dist}(v)$ is the total weight of some path from s to v.

   (c) Prove that when the generic shortest-path algorithm halts, then for every vertex v, either $\mathit{dist}(v) = \infty$, or $\mathit{dist}(v)$ is the total weight of the predecessor chain ending at v:

   $$s \to \cdots \to \mathit{pred}(\mathit{pred}(v)) \to \mathit{pred}(v) \to v.$$

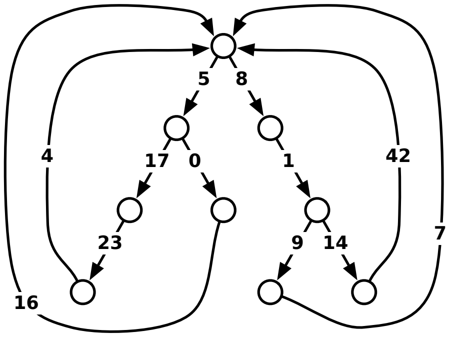
   (d) Prove that when the generic shortest-path algorithm halts, then $\mathit{dist}(v) \leq w(s \rightsquigarrow v)$ for *every* path $s \rightsquigarrow v$.

2. Prove the following statement for every integer i and every vertex v: At the end of the ith phase of Shimbel's algorithm, $\mathit{dist}(v)$ is less than or equal to the length of the shortest path $s \rightsquigarrow v$ consisting of i or fewer edges.

*3. Prove that Ford's generic shortest-path algorithm (while the graph contains a tense edge, relax it) can take exponential time in the worst case when implemented with a stack instead of a priority queue (like Dijkstra) or a queue (like Shimbel's algorithm). SpeciÞcally, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Towers of Hanoi.]*

*4. Prove that Dijkstra's shortest-path algorithm can require exponential time in the worst case when edges are allowed to have negative weight. Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that Dijkstra's algorithm calls RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: This should be easy if you've already solved the previous problem.]*

5. 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 a 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.

6. (a) Describe a modification of Ford's generic shortest-path algorithm 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. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle.

   (b) Describe a modification of Shimbel's shortest-path algorithm 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.

7. For any edge e in any graph G, let $G \setminus e$ denote the graph obtained by deleting e from G.

   (a) Suppose we are given a directed graph G in which the shortest path σ from vertex s to vertex t passes through *every* vertex of G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Your algorithm should output a set of E shortest-path distances, one for each edge of the input graph. 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?]*

   *(b) Let s and t be *arbitrary* vertices in an *arbitrary* directed graph G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Again, you may assume that all edge weights are non-negative.

8. 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.

9. Negative edges cause problems in shortest-path algorithms because of the possibility of negative cycles. But what if the input graph has no cycles?

   (a) Describe an efficient algorithm to compute the shortest path between two nodes s and t in a given *directed acyclic graph* with weighted edges. The edge weights could be positive, negative, or zero.

   (b) Describe an efficient algorithm to compute the *longest* path between two nodes s and t in a given directed acyclic graph with weighted edges.

10. After a grueling algorithms midterm, 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 Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.

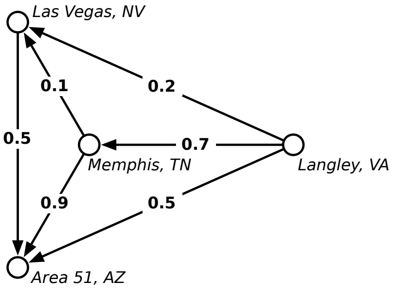
    Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend traveling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

11. After graduating 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. *[Hint: Modify the input data and apply Dijkstra's algorithm.]*

12. 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.



    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.)

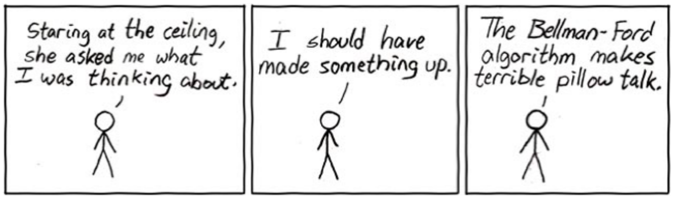
13. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he 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 leave the park immediately.

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!*


Staring at the ceiling, she asked me what I was thinking about.
I should have made something up.
The Bellman-Ford algorithm makes terrible pillow talk.

— Randall Munroe, *xkcd* (http://xkcd.com/c69.html)