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)

19 Shortest Paths

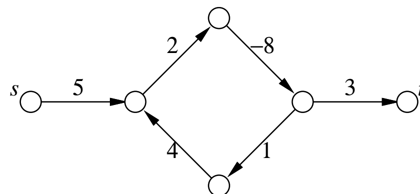
19.1 Introduction

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 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, because the presence of a negative *cycle* might imply 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 we can always find a shorter path by going around the cycle one more time.



There is no shortest path from s to t.

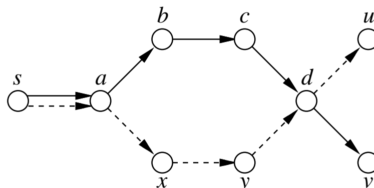
[1] West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We both live in Urbana now.

[2] At one time, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.

© Copyright 2012 Jeff Erickson. Released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License (http://creativecommons.org/licenses/by-nc-sa/3.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden. See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms for the most recent revision.

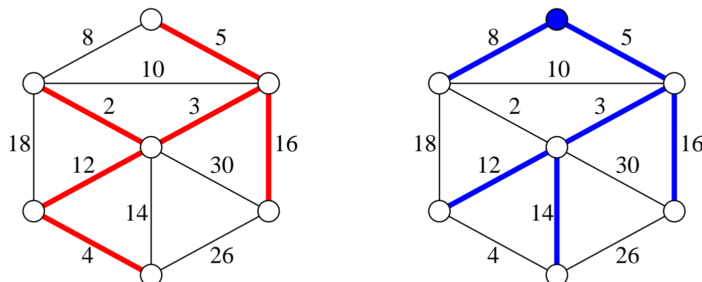
Almost every algorithm known for solving this problem actually solves (large portions of) the following more general ***single source shortest path*** or ***SSSP*** problem: find the shortest path from the source vertex s to *every* other vertex in the graph. In fact, the problem is usually solved by 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 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 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 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 very different. For one thing, there is only one minimum spanning tree (if edge weights are distinct), but each source vertex induces a different shortest path tree. Moreover, in general, *all* of these shortest path trees are different from the minimum spanning tree.

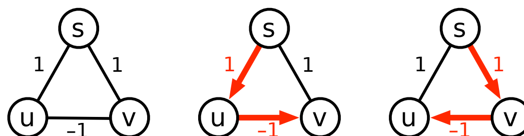


A minimum spanning tree and a shortest path tree (rooted at the topmost vertex) of the same graph.

19.2 Warning!

Throughout this lecture, we will explicitly consider *only* directed graphs. All of the algorithms described in this lecture also work for undirected graphs with some minor modifications, *but only if negative edges are prohibited*. Dealing with 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.

A complete treatment of undirected graphs with negative edges is beyond the scope of this lecture (if not the entire course). I will only mention 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.



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

19.3 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 Lester Ford in 1956, and independently by George 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.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or NULL if there is no such vertex.

In fact, 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.

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

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

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

The correctness of the generic relaxation algorithm follows from the following series of claims:

1. For every vertex v, the distance $dist(v)$ is either ∞ or the length of some 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 weight. This claim implies that if G has no negative cycles, the relaxation algorithm eventually halts, because there are only finitely many 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 $s \rightarrow \cdots pred(pred(v)) \rightarrow pred(v) \rightarrow v$. Specifically, if v violates this condition but its predecessor $pred(v)$ does not, the edge $pred(v) \rightarrow v$ is tense.

4. If no edge in G is tense, then for every vertex v, the path $s \rightarrow \cdots pred(pred(v)) \rightarrow pred(v) \rightarrow v$ is 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 $u \rightarrow v$ is tense. This claim also implies that if the G has a negative cycle, then some edge is *always* tense, so the algorithm never halts.

[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 equivalent to Ford's relaxation strategy.

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) ← 0
  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← 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→v
          if u→v is tense
              RELAX(u→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 FIFO queue, and a priority queue. 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 prove to be much more efficient.

19.4 Dijkstra's Algorithm

If we implement the bag using a priority queue, 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 George 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.) On the other hand, in practice, Dijkstra's algorithm is usually quite fast even for graphs with negative edges.

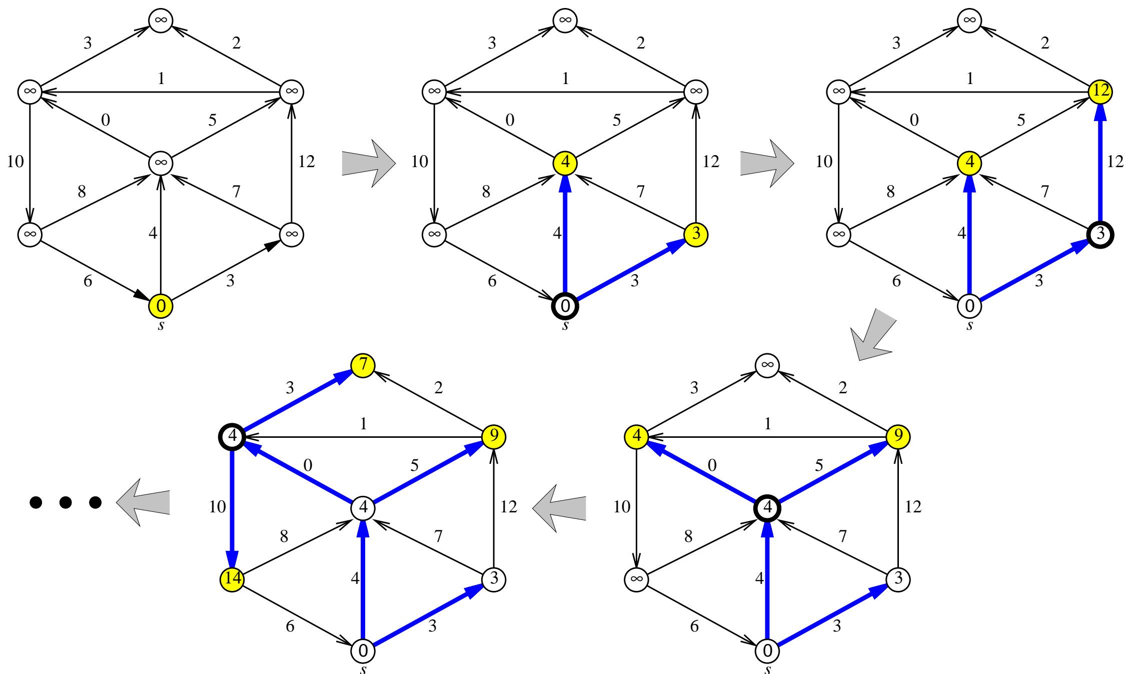
19.5 The A^* Heuristic

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

[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]Most textbooks, and indeed Dijkstra's original paper, present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges, because it *never* visits the same vertex more than once.



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.

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 $dist(v) + \text{GUESSDISTANCE}(v, t)$.

The function GUESSDISTANCE is called *admissible* if GUESSDISTANCE(v, t) never overestimates the actual shortest path distance from v to t. If 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 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, Racetrack, ...) 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.

19.6 Shimbel's Algorithm

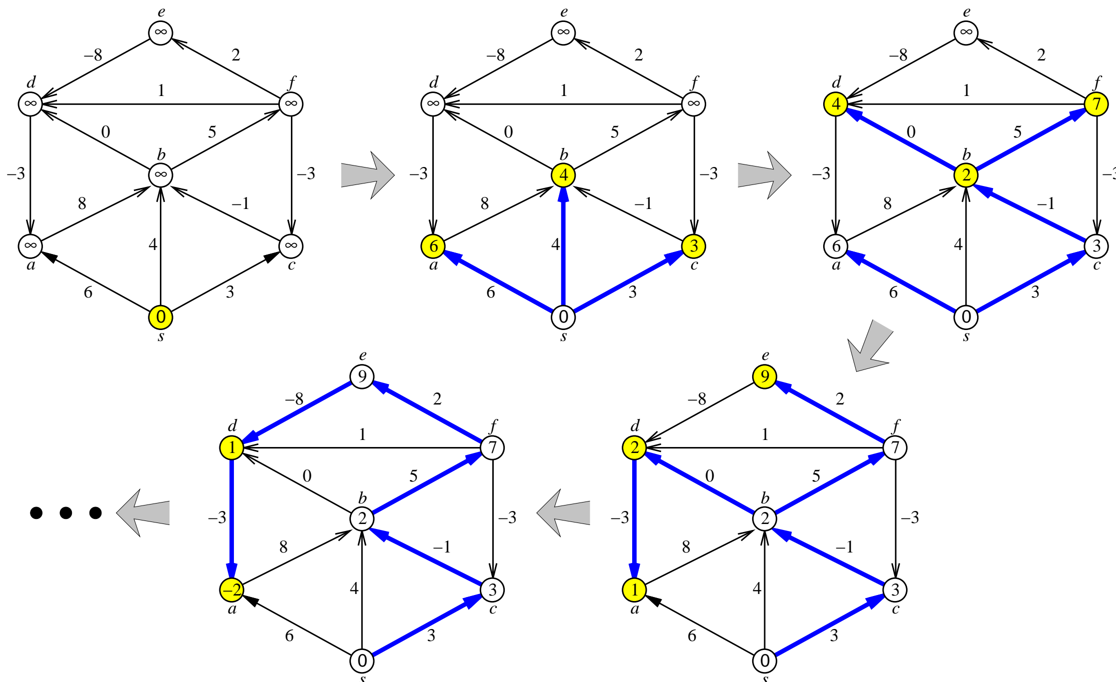
If we replace the heap in Dijkstra's algorithm with a FIFO queue, we obtain an algorithm first sketched by Shimbel in 1954, described in more detail by Moore in 1957, then independently rediscovered by Woodbury and Dantzig in 1957, and by Bellman in 1958. Because Bellman explicitly used Ford's formulation of relaxing edges, this algorithm is usually called 'Bellman-Ford', although some early sources refer to 'Bellman-Shimbel'. 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.)

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 at most the length of the shortest path $s \rightsquigarrow v$ consisting of at most i 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 $\boldsymbol{O(VE)}$.



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 end-of-phase token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

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→v
          if u→v is tense
              RELAX(u→v)
  for every edge u→v
      if u→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 have to show that our earlier invariant holds; as before, this can be proved by induction on i.

19.7 Shimbel's Algorithm as Dynamic Programming

Shimbel's algorithm can also be recast as a dynamic programming algorithm. Let $dist_i(v)$ denote the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges. It's not hard to see that this function obeys the following recurrence:

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

For the moment, let's assume the graph has no negative cycles; our goal is to compute $dist_{V-1}(t)$. We can clearly memoize this two-parameter function into a two-dimensional array. A straightforward dynamic programming evaluation of this recurrence looks like this:

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

Now let us make two minor changes to this algorithm. First, we remove one level of indentation from the last three lines. This may change the order in which we examine edges, but the modified algorithm still computes $dist_i(v)$ for all i and v. Second, 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 is still correct.

```
SHIMBELDP2(s)
  dist[0, s] ← 0
  for every vertex v ≠ s
      dist[0, v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i, v] ← dist[i − 1, v]
      for every edge u→v
          if dist[i, v] > dist[i, u] + w(u→v)
              dist[i, v] ← dist[i, u] + w(u→v)
```

[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.

Now notice that the iteration index i is completely redundant! We really only need to keep a one-dimensional array of distances, which means we don't need to scan the vertices in each iteration of the main loop.

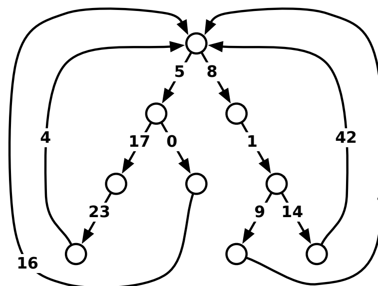
```
SHIMBELDP3(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

The resulting algorithm is (almost) identical to our earlier algorithm SHIMBEL! The first three lines initialize the shortest path distances, and the last two lines check whether an edge is tense, and if so, relaxes it. The only thing missing is the explicit maintenance of predecessors, but that's easy to add.

Exercises

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

2. 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.

3. 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.

4. 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.

*5. 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). Specifically, 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.]*

*6. 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.]*

7. (a) Describe and analyze 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.

   *(b) Describe and analyze 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.

8. 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.

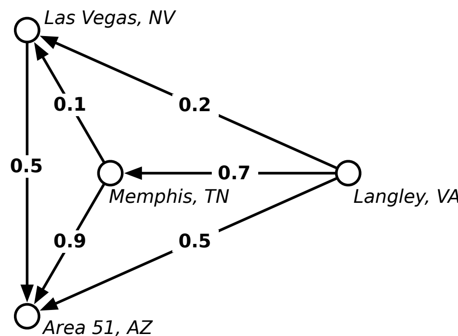
   Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend traveling. Assume that the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

9. 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.]*

10. 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.)

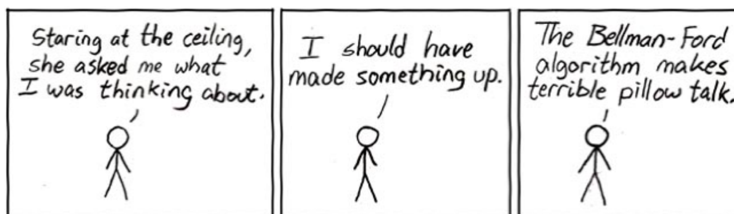
11. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he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



— Randall Munroe, *xkcd* (http://xkcd.com/69/)
Reproduced under a Creative Commons Attribution-NonCommercial 2.5 License

© Copyright 2012 Jeff Erickson. Released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License (http://creativecommons.org/licenses/by-nc-sa/3.0/).
Free distribution is strongly encouraged; commercial distribution is expressly forbidden. See http://www.cs.uiuc.edu/~jeffe/teaching/algorithms for the most recent revision.