12 Shortest Paths (March 4)

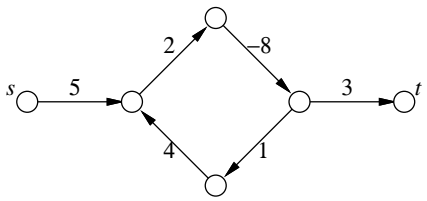
12.1 Introduction

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 we want to answer the question "What's the fastest way to drive from my apartment in Champaign, Illinois to my wife's 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. The graph is directed since the driving times along the same road might be different in different directions.[1]

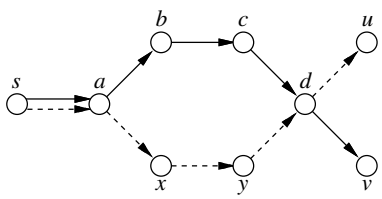
Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, since the presence of a negative cycle might mean that there is no shortest path. In general, a shortest path from s to t exists if and only if there is *at least one* path 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 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.

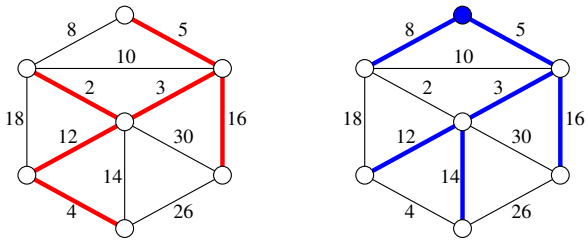
It's not hard to see that if the shortest paths are unique, then they form a tree. To prove this, it's enough to observe that sub-paths of shortest paths are also shortest paths. If there are multiple shortest paths to the same vertices, we can always choose one path to each vertex so that the union 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.

[1] For example, there might be a speed trap on I-70 just inside the Ohio border, but only for westbound traffic.

I should emphasize here that shortest path trees and minimum spanning trees may be very different. For one thing, there is only one minimum spanning tree, but there may be a different shortest path tree for every source vertex.



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

12.2 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. Each vertex v in the graph stores two values, which describe a *tentative* shortest path from s to v.

- $\text{dist}(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path.
- $\text{pred}(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path.

Notice that the predecessor pointers automatically define a tentative shortest path tree. We already know that $\text{dist}(s) = 0$ and $\text{pred}(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $\text{dist}(v) = \infty$ and $\text{pred}(v) = \text{NULL}$ to indicate that we do not know of *any* path from s to v. We then repeatedly apply the following *relaxation* step:

```
find an edge u → v such that
      dist(u) + w(u → v) < dist(v)
and relax it:
      dist(v) ← dist(u) + w(u → v)
      pred(v) ← u
```

If we successfully find an edge $u \to v$ to relax, then we know that our tentative shortest path to v is incorrect, since the path $s \rightsquigarrow u \to v$ is shorter than our old path $s \rightsquigarrow v$. If we cannot find an edge to relax, the 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 $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of the predecessor chain ending at v:

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

   This is easy to prove by induction on the number of relaxation steps. (Hint, hint.)

2. If the algorithm halts, then $\text{dist}(v) \leq w(s \rightsquigarrow v)$ 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. One direction is obvious. The other direction follows from the fact that every relaxation step reduces either the number of vertices with $\text{dist}(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by some positive amount.

I haven't said anything about how we detect which edges can be relaxed, or what order we relax them in. 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" it, attempting to relax every edge out of u. Whenever we successfully relax and edge $u \to v$, we put v in the bag.

```
GENERICSSSP(s):
  dist(s) ← 0;  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞;  pred(v) ← NULL

  put s in bag
  while bag is not empty
      take u from bag
      for all edges u → v
          if dist(u) + w(u → v) < dist(v)
              dist(v) ← dist(u) + w(u → v)
              pred(v) ← u
              put v in bag (unless it's already there)
```

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 $\Theta(2^E)$ relaxation steps in the worst case! The other two possibilities are much more efficient.

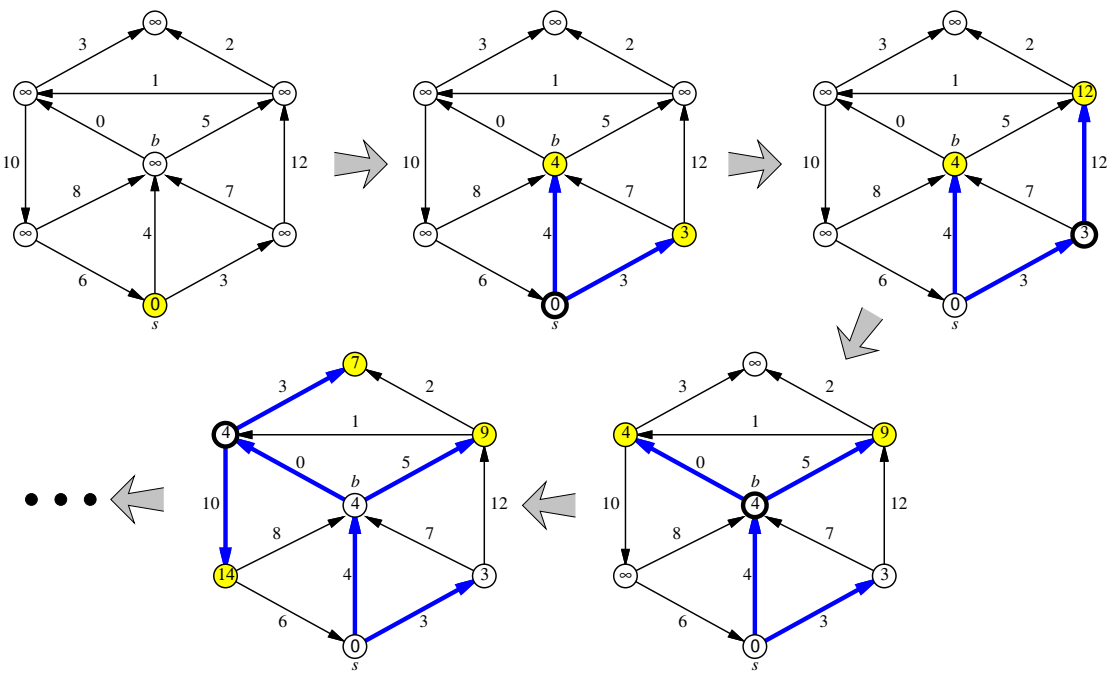
12.3 Dijkstra's Algorithm

If we use a heap, where the key of a vertex v is $\text{dist}(v)$, we obtain an algorithm first published by Edsger Dijkstra in 1959.

Dijkstra's algorithm 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 $O(E + V \log V)$.

This analysis assumes that no edge has negative weight. Dijkstra's algorithm is still *correct* if there are negative edges[2], but the worst-case running time could be exponential.

[2] At least, in the from I'm presenting it here. The version of Dijkstra's algorithm presented in CLR gives incorrect results for graphs with negative edges.



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

12.4 Moore's Algorithm ("Bellman-Ford")

If we replace the heap in Dijkstra's algorithm with a FIFO queue, we get an algorithm that was first published by Moore in 1957, and then independently by Bellman in 1958. (Since Bellman used the idea of relaxing edges, which was first proposed by Ford in 1956, this algorithm is usually called "Bellman-Ford".) Moore'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. Before we even begin, we insert a "token" into the queue. Whenever we take the token out of the queue, we begin a new phase by reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex s.

Figure!

Five phases of Moore's algorithm run on a directed graph with negative edges.
The bold edges describe the evolving shortest path tree.

A simple inductive argument implies that at the end of the ith phase, every $\text{dist}(v)$ is the length of the shortest path from s to 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 if $O(E)$. Thus, the overall running time of Moore's algorithm is $O(V \cdot E)$.

Now that we understand how the phases of Moore'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, let's just scan through the adjacency list directly and try to relax every edge in the graph.

```
MooreSSSP(s)
    dist(s) ← 0;  pred(s) ← Null
    for all vertices v ≠ s
        dist(v) ← ∞;  pred(v) ← Null

    repeat V times:
        for every edge u → v
            if dist(u) + w(u → v) < dist(v)
                dist(v) ← dist(u) + w(u → v)
                pred(v) ← u

    for every edge u → v
        if dist(u) + w(u → v) < dist(v)
            return "Negative cycle!"
```

This is closer to how CLR presents the "Bellman-Ford" algorithm. The $O(V \cdot E)$ 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 after the ith iteration of the outer loop, each $\text{dist}(v)$ is the length of the shortest path from s to v with less than i edges. Just like the earlier version of Moore's algorithm, this can be proved by induction on i.