1 Shortest Paths

Recall from Lecture 6 that BFS is a simple variant of the general `Search-Tree` algorithm in which we store the edges we explore in a first-in-first-out (FIFO) queue. The version we considered only worked on undirected and unweighted graphs, and that BFS returns a tree that is composed of the shortest paths from the source vertex s to all other vertices $V - s$. If we wanted to generalize BFS to work on weighted networks, we can simply use a data structure called *priority queue*, which maintains its contents in an ordered way, from largest to smallest weight edges, so that each time we dequeue an item, we get the smallest edge.

It can be useful to think of the behavior of BFS as growing a tree outward in layers from the source. Tree edges always cross from a layer ℓ to a layer $\ell + 1$, and all nodes in a layer ℓ are at distance ℓ from s. Further, each edge in the graph that is not a tree edge must connect a pair of nodes with the same distance from s. The tree that BFS produces is a kind of *spanning tree* for all the nodes reachable in G from s, although it may not be the minimum spanning tree.[1]

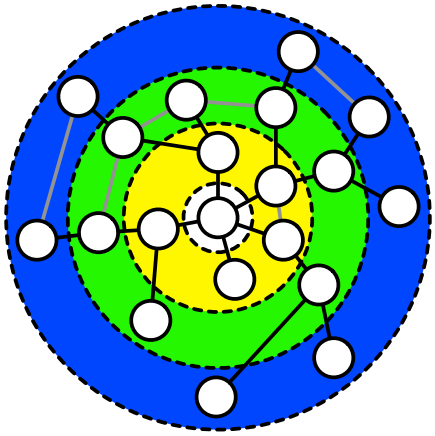


Figure 1: A BFS tree on a simple network, showing the way BFS explores the network in layers; tree edges are shown in black and non-tree edges are shown in grey.

1.1 All-Pairs-Shortest-Path

Suppose we wanted an algorithm that could tell us the shortest path from point A to B in a graph, for any pair of choices? This problem is very similar to what Google Maps solves when you ask

[1] We'll cover minimum spanning trees next week.

it for driving directions.[2] If we can precompute a pairwise distance matrix $d(A, B)$ for all pairs A, B, and also store the corresponding route, we can simply look up the answer when we're given a query. This is not an efficient solution in terms of space or time since it takes $\Omega(V^2)$ time to construct the matrix. Most real-world solutions take a hybrid approach, precomputing some short paths and then stitching them together into an approximately-good solution, on demand.

Algorithmically, the problem is as follows: given a weighted, directed graph $G = (V, E)$ and two vertices s (source) and t (target), find the path σ_{st} from s to t, which minimizes the function

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

That is, find the path with smallest total weight. All solutions to this problem solve, for each possible choice of s, the *single-source shortest paths* or SSSP problem. That is, given s, find the shortest path from s to t, for each $t \in V - s$. If the network were undirected, we can simply use BFS because the BFS tree is a shortest path tree, rooted at s.

1.1.1 SSSP tree $\neq$ MST

We haven't discussed them yet, but next week we'll cover minimum spanning trees (MST). A minimum spanning tree is the subset of edges of G with minimum weight conditioned on their forming a spanning tree, i.e., every pair of vertices $u, v \in V$ is reachable within the MST if the pair is reachable in G.

For a given graph G, a single-source shortest path tree rooted at s is not necessarily the same as the minimum spanning tree for G. You can see this quickly by noting that there are at most $|V|$ SSSP trees—one for each $s \in V$—while there's at least one MST. They *can* be the same, but they don't have to be. Here's an example.

[2] Actually, the problem Google Maps solves is both harder and easier. It is harder because the graph is more complicated. In a road network, "intersections" where several roads come together are nodes and edges are the stretches of pavement between intersections. Edges are decorated with several types of information: spatial length (distance), speed limit, toll-road or not, one-way or not, and position in the road "hierarchy," e.g., side street, city street, major artery, state highway, major highway, etc. Further, driving times may be different in different directions, so the network is directed. Identifying the "best" route between A and B depends on how you define best: it could be shortest distance (length of trip), shortest time (duration of trip), fewest toll roads, or even some complicated combination of these or other functions. This problem is easier because A and B have distinct spatial locations, which means that choices can be made in a spatially greedy fashion: from a particular location x in the network, we can choose from among the edges originating there the one that most reduces the remaining distance $d(x, B)$. On the other hand, this kind of greedy algorithm is not how people actually navigate road networks. Can you think of the difference?

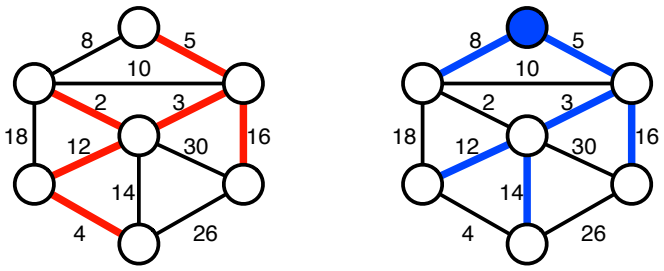
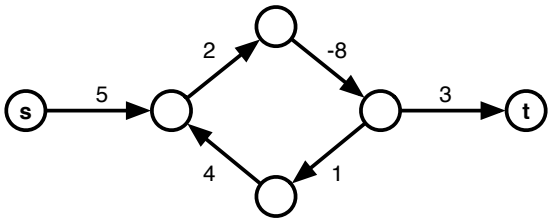


Figure 2: The left-hand figure shows a minimum spanning tree while the right-hand figure shows a single-source shortest path tree rooted at the blue node.

1.1.2 Negative cycles

What happens if a cycle in G contains a *negative* weight edge? For instance, consider this small graph:



In this case, the minimum weight path from s to t has weight $-\infty$ (but length ∞), because each cycle through the central loop adds -1 to the path's weight. This is inconvenient. To prevent this, while preserving generality, we simply define "shortest path" to mean any path with minimum weight *and* which does not touch a negative cycle. If no such path from s to t exists, then there is no shortest path between them. (For applications of SSSP algorithms to undirected graphs, this requirement remains because a single negative weight edge constitutes a negative weight cycle.)

1.1.3 A general SSSP algorithm

Like with our `Search-Tree` algorithm, this will be a general framework, and particular SSSP algorithms will be special cases. Remember that the SSSP takes as input a graph G and a source vertex s; let $w(u, v)$ return the weight of the edge $(u, v) \in E$. Also like the `Search-Tree` algorithm, we need to store two pieces of information about each vertex:

$dist(v)$ returns the length of the tentative shortest path between s and v

$pred(v)$ returns the predecessor of v in this tentative shortest-path
(the set of these represent the single-source shortest path tree)

As with the `Search-Tree` algorithm from Lecture 6, these are initialized as:

$$\begin{aligned} dist(s) &= 0 \\ dist(v) &= \infty, \text{ for all } v \neq s \end{aligned}$$

$$pred(v) = NULL, \text{ for all } v \in V$$

Now, we make a crucial observation. What does a *bad* solution to the SSSP problem look like? A bad solution means that for some vertex v, there exists a shorter path from s to v than is contained in our SSSP tree. This implies the existence of some edge edge (u, v) where $dist(u)+w(u, v) < dist(v)$. We call (u, v) a *tense* edge.

If (u, v) is tense, then the tentative shortest path from s to v is incorrect because the path weight first from s to u plus the cost of (u, v) is less than the weight of the current path to v. This observation implies an algorithmic solution for the SSSP problem: repeatedly find a tense edge in the graph and then *relax* it, i.e., we make the path run through u instead. To relax an edge, we simply update the tree and the distances to reflect the incorporation of (u, v) into the tree:

```
Relax(u,v) {
   dist(v) = dist(u) + w(u,v)
   pred(v) = u
}
```

Although we have not unspecified exactly how we detect which edges are tense or in what order we choose to relax them, we can see that an algorithm that relaxes tense edges will halt so long as relaxing an edge does not produce more tense edges and that its output will be a correct SSSP tree because a SSSP tree, by definition, has no tense edges.

There are many possible choices we could make about how to find and relax tense edges. To avoid being too specific yet (we'll get there), here's a generic approach: we will maintain a *set*[3] of vertices S; whenever we remove some vertex u from S, we examine each of its outgoing edges (u, v) and check if it can be relaxed. (This punts on the issue of the ordering for now, but we will revisit that in a moment.) If we successfully relax some edge (u, v), then we place v in the set. That's it.

[3]This should be an immediate clue about both the running time and future implementation choices: a "set" is a generic type of data structure and its internal organization determines the time and space required to find, remove or add items. Dictionary ADTs can be used to maintain a set.

Initially, $S = \{s\}$ and the distance to all vertices $V - s$ is infinite; thus, all of the edges $(s, x) \in E$ are *tense*. (In fact, all edges that reach from the vertices in the tree to vertices not yet in the tree are tense.) Each time we relax a tense edge, we either grow the tree to include a new vertex or we reduce the cost of the path to some x already in the tree.

Here's pseudocode for this generic algorithm:

```
Generic-SSSP(G,s) {
   dist(s) = 0                   // initialize distances and
   pred(s) = NULL                // shortest-path tree arrays and
   for all vertices v != s {     // set data structure
      dist(v) = inf              //
      pred(v) = NULL             //
   }                             //
   S = emptySet()                //

   S.add(s)                      // add source vertex to the set
   while S.notempty() {          // grow the tree
      u = S.get()                // get some vertex from the set
      for all edges (u,v) {      // examine all of its outgoing links
         if (u,v) is tense {     // relax the tense ones
            Relax(u,v)           //
            S.add(v)             // add u to the set
}}}}
```

Before we think about the particular choices left unspecified here, let's quickly prove that this algorithm is correct, i.e., it halts on all correctly specified inputs and it returns a solution that satisfies the target criteria.

Claim 1: If $dist(v) \neq \infty$, then $dist(v)$ is the total weight of the predecessor chain ending at v:

$$s \to \cdots \to pred(pred(v)) \to pred(v) \to v \ .$$

Proof: By induction on the number of edges in the path from s to v. (Do you see how?)

Claim 2: If the algorithm halts, then $dist(v) \leq w(s \rightsquigarrow v)$ for all paths $s \rightsquigarrow v$.

Proof: By induction on the number of edges in the path $s \rightsquigarrow v$. (Do you see how?)

Claim 3: The algorithm halts if and only if there is no negative cycle reachable from s.

Proof: The "only if" direction is easy to prove—if there is a reachable negative cycle, then after the first edge in the cycle is relaxed (i.e., our path touches the cycle), the cycle always has at least one tense edge and thus the algorithm will never halt. The "if" direction follows from the fact that every relaxation step reduces either the number of vertices with $dist(v) = \infty$ by 1 (because we add a novel vertex to our SSSP solution) or reduces the sum of the finite shortest path lengths by some positive amount (because we find a shorter path to a vertex already in our SSSP solution). □

So, `Generic-SSSP` does what we claim. Now, it remains to decide how to manage the set of vertices. Different choices here lead to different algorithms, which have different running times. Some obvious choices for data structures are the usual suspects: a stack, a queue and a heap. If we use a stack, we need to perform $\Theta(2^{|E|})$ relaxation steps at worst. (Do you see why?)

1.1.4 Dijkstra's algorithm

If we implement the set as a min-heap data structure[4] then we obtain Dijkstra's algorithm. In this case the key in the min-heap for some vertex v is the tentative distance, i.e., $dist[v]$, from s to v. This implies that the algorithm evaluates the vertices in increasing order of their shortest-path distance from s and thus each vertex is evaluated at most once. Each time we evaluate a vertex, we then evaluate each of its attached edges; thus, we evaluate each edge at most once. (Figure 3 shows an example of Dijkstra's algorithm on a small graph.)

Each time an edge (u, v) is relaxed, we have to update the key associated with v. At worst, we will relax every edge, and thus perform E updates to the heap keys. And, there are at most V times when we add or remove a key to the heap.

If we use a classic binary min-heap, then each heap update takes $O(\log V)$ and so the running time is $O(E \log V)$. But, if we use a Fibonacci min-heap, which has lower amortized cost for updates, the running time is only $O(E + V \log V)$. This is because each of the E updates now costs amortized $O(1)$ but removing each of the V keys from the heap (and we remove one key each time we evaluate a node) costs $V \log V$. The running time is whichever of these is larger.

[4]We haven't covered heaps in the lectures, but they're a very important type of data structure. A "min-heap" allows us to find the smallest key in the set very quickly while other operations are more expensive; a "max-heap" is just the same, but for the largest key in the set. Heaps can be implemented in an array fairly easily, but have a maximum size unless you implement dynamic re-sizing. Heaps are most naturally implemented as trees; see Chapters 19 and 20 on Binomial and Fibonacci heaps. Fibonacci heaps have better amortized running time than Binomial heaps: add, find minimum, decrease key, and merge heaps work in amortized $O(1)$ time, while remove works in amortized $O(\log n)$ time.

Note: this analysis assumes no negative edge weights; if there are negative edge weights, the worst-case running time is exponential.

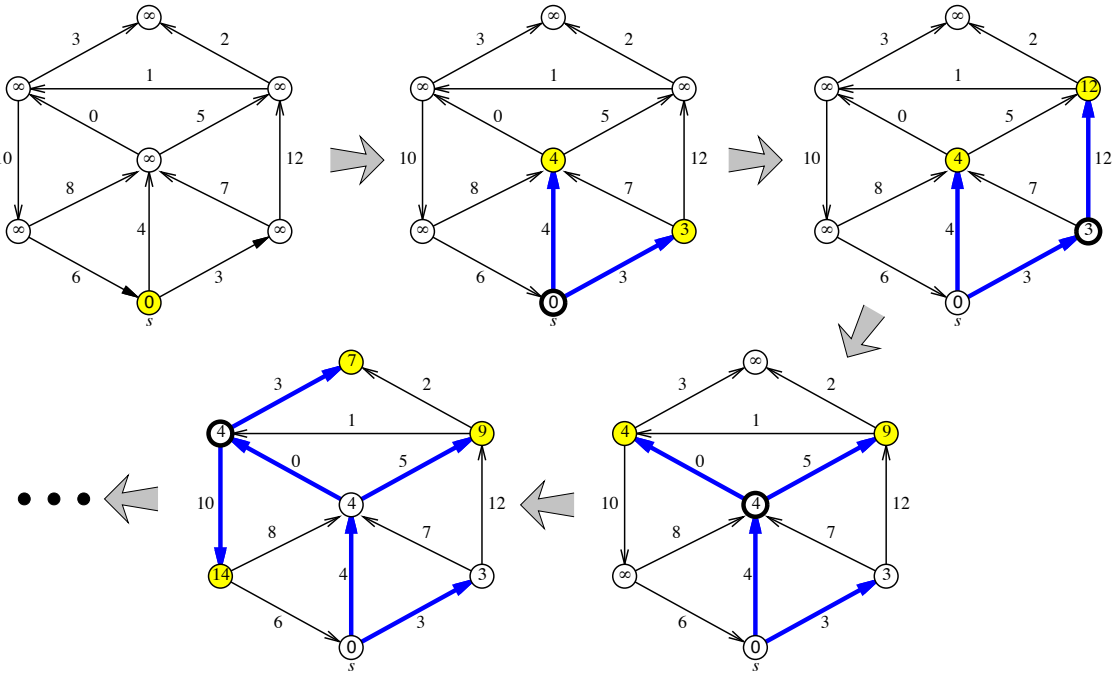


Figure 3: Four phases of Dijkstra's algorithm, run on a graph with no negative edges. At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned. The bold edges describe the evolving shortest path tree. (Taken from Jeff Ericson's lecture notes.)

1.1.5 Bellman-Ford algorithm

If we implement the set as a queue, then we obtain the Bellman-Ford algorithm. This algorithm is efficient even if there are negative edges (and thus we could use Bellman-Ford to *detect* their presence). The tradeoff, however, is a worse running time than Dijkstra; that is, if there are no negative edges, Dijkstra beats Bellman-Ford.

To analyze the running time of Bellman-Ford, let's break its behavior up into "phases," each of which is defined like this. At the beginning of the algorithm, we insert a special token into the queue. Whenever we remove the token from the queue, we begin a new phase and insert the token back into the queue. (The 0th phase consists only of scanning the source vertex s.) The algorithm

ends when the token is the only thing in the queue. Figure 4 shows an example of running the Bellman-Ford algorithm on a small graph.

Claim 4: At the end of the ith phase, for each vertex v, $dist(v)$ is less than or equal to the length of the shortest path $s \rightsquigarrow v$ consisting of i or fewer edges.

Proof: By induction (left as an exercise).

Since a shortest path can only pass through each vertex at most once, the algorithm either halts before the $|V|$th phase or the graph contains a negative cycle. In each phase, we scan each vertex at most once and we relax each edge as most once. Thus, the running time for a single phase is $O(E)$ and the running time of the entire algorithm is $O(VE)$.

Here's an alternative way to construct Bellman-Ford, which has the same asymptotic behavior. Note that each phase of the queue-based version of the algorithm is basically trying to grow a BFS sourced at the vertex v. Instead of this, we could simply scan through the adjacency list directly and try to relax each edge. There are $|V|$ of these passes and each one takes worst $|E|$ time, so the running time is still $O(VE)$. This version can be shown correct by proving, by induction on i, that Claim 4 still holds. (What G leads to the worst case running time?)

1.1.6 The A^* heuristic

One slight generalization of Dijkstra's algorithm is frequently used to solve a related problem, which is to find a shortest path from some s to some particular t. The A^* heuristic, as it is commonly known, uses a function `Guess-Distance(v,t)` that returns an estimate of the distance from some vertex v to the target t. The difference between Dijkstra and A^* is that the key associated with a vertex is $dist(v) +$ `Guess-Distance`(v, t). Clearly, the more accurate `Guess-Distance(v,t)` is, the faster the algorithm runs, but in the worst case A^* still runs in $O(E + V \log V)$ time. One advantage of A^* is that it can be used even when the entire structure of the graph is not completely known (or quickly available), e.g., when crawling the World Wide Web or when searching a space of exponential size, as in many puzzle or game-solving algorithms, or in planning problems where the starting and target states are given, but the state space is not explicitly known.

2 For Next Time

1. Minimum Spanning Trees
2. Read Chapters 24 (Single-Source Shortest Paths) and 25 (All-Pairs Shortest Paths)

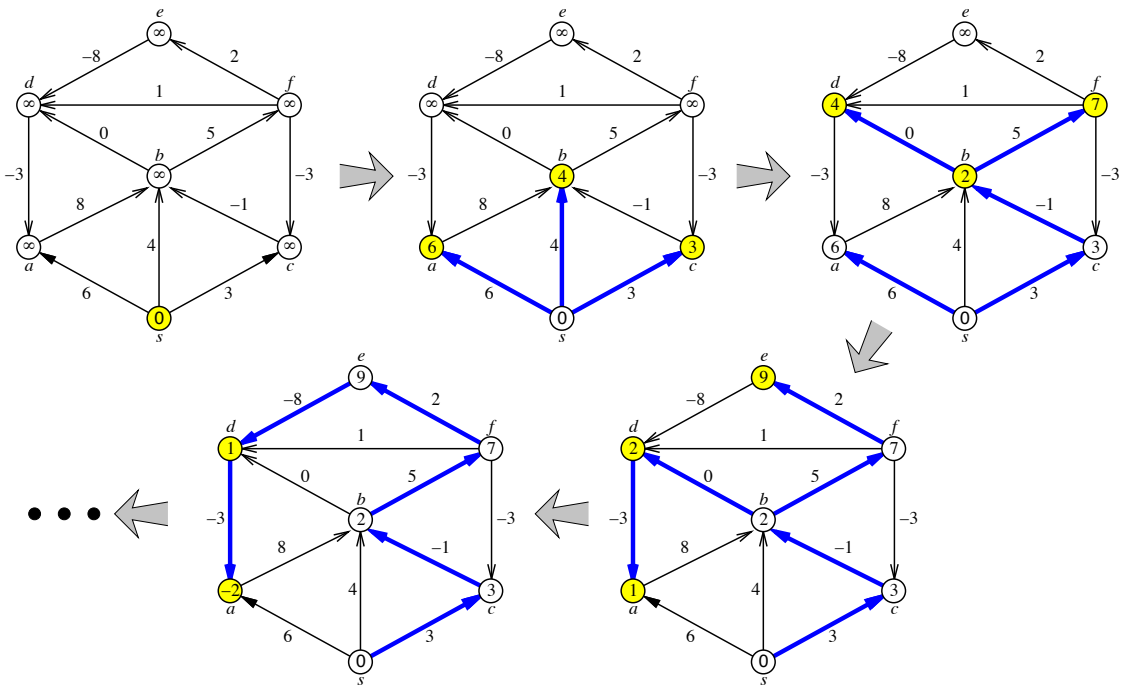


Figure 4: Four phases of the Bellman-Ford algorithm, run on a directed graph with some negative edges. Nodes are taken from the queue in the order $s \diamond abc \diamond dfb \diamond aed \diamond da \diamond \diamond$, where $\diamond$ is the token. Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path. (Taken from Jeff Ericson's lecture notes.)