1 Shortest Paths with Weights

Recall that we can extract a breadth-first search (BFS) tree T from a graph $G = (V, E)$ using a special case of the general `Search-Tree(G,s)` algorithm. In this version of the algorithm, we use a first-in-first-out (FIFO) queue to manage the order in which we examine vertices with at least one neighbor in the intermediate search tree T.

Thus, the algorithm explores *every* vertex of distance ℓ (measured in number of edges) from the source vertex s before it explores *any* vertex of distance $\ell + 1$, for all ℓ. The resulting BFS tree T will contain the union of "geodesic" or shortest paths from s to all other vertices $V - s$. (If G is connected, then T contains exactly $n - 1$ edges, while if G has multiple components, then $|T| < n - 1$. Do you see why?)

Because "distance" for a BFS is defined as a count of the number of edges, it can only be applied to unweighted graphs (but, it is still correct for multigraphs and graphs with self-loops; do you see why?). However, many graphs have edge weights and this property complicates the discovery of short paths.

The key problem is this: *in a weighted graph, we need to correctly choose a path composed of many edges that has a lower total weight than a path with fewer edges but more weight.*

The solution is to generalize our shortest-path algorithm by using *priority queue* to store a list of "pending" vertices, rather than a simple FIFO queue. Now, the position of a vertex $y \in V$ in the queue will be given by the edge weight w_{xy} of edge $(x \to y) \in E$ that we could add to T in order to reach y. This priority queue thus allow us to repeatedly choose the smallest-weight edge to add.

1.1 All-Pairs-Shortest-Path

Suppose that we are tasked with finding the shortest path between any pair of vertices in our graph $s, t \in V$. This problem is similar to what Google Maps solves when you ask it for driving directions.[1] We could precompute a pairwise distance matrix $d(s, t)$ for all pairs s, t, also storing

[1]In fact, Google Maps solves a problem that 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 vertices 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 properties. On the other hand, this problem is easier because A and B have distinct spatial locations, meaning that choices can often 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

the corresponding route. Then, to fulfill a query, we simply lookup the answer. This solution uses a one-time computation to produce the matrix $d(s,t)$—search queries are then fulfilled in $O(1)$ time—but requires $\Omega(V^2)$ space to store the matrix and routes. For moderate-sized graphs ($|V| > 10^4$), this approach is highly inefficient.[2]

We can now formalize this *all-pairs shortest paths* problem. The input is a directed graph $G = (V,E)$, and a weight function $w : e \to \mathbb{R}$ for all $(i,j) = e \in E$. The output is a set of paths $\{\sigma_{st}\}$, each with weight $w(\sigma_{st})$, for all pairs of vertices $s,t \in V$, such that we minimize

$$\sum_{s,t\in V} w(\sigma_{st}) = \sum_{s,t\in V} \left(\sum_{e\in\sigma_{st}} w(e) \right) \enspace .$$

where σ_{st} is a sequence of edges $s \to x \to \cdots \to y \to t$ in E. That is, find want to find the set of pairwise paths with smallest total weight.

A key insight is that the set of shortest paths from s to every $t \in V - s$ can be obtained without reference to the set for some other vertex $r \neq s$. Thus, we can solve the all-pairs problem by independently solving the *single-source shortest paths* (also called SSSP) problem for each vertex $s \in V$, and then combinine the results of these subproblems.

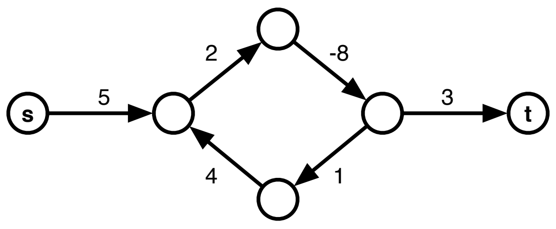
1.1.1 The problem of negative cycles

What happens if (i) G contains a cycle $\sigma_{xx} = x \to y \to \cdots \to z \to x$, and (ii) that cycle contains a negative weight edge $\exists e \in \sigma_{xx}$ where $w(e) < 0$? If the weight of the cycle is positive $w(\sigma_{xx}) > 0$, then nothing bad happens: a single pass around the loop still increases the distance from some s to some t (see schematic below), and a correct algorithm will never add a cycle to some shortest path because adding it would increase the total weight, while omitting it would produce a small-weight path.

Now what happens if there is some combination of negative-weight edges on a cycle such that the total weight of that cycle is negative $w(\sigma_{xx}) < 0$:

distance $d(x,B)$. But, this kind of greedy algorithm is not how people actually navigate road networks. Can you think of the difference?

[2]For many real-world problems, the *shortest* path is not required, and instead we simply want *some* short path, where perhaps short is defined as some stretch factor c times the length of the shortest path. These can be found using a hybrid approach, precomputing shortest paths between a subset of vertices and then stitching these locally-optimal paths together into an approximately-good global solution, on demand. The tradeoff is that for some types of inputs, this heuristic can fail badly. In this lecture, we aim for a more general solution, one that works correctly on all inputs.



Now, the minimum weight path from s to t has weight $-\infty$ (and length ∞). In this example, the weight of the loop is $1+2+4-8=-1$, and each pass around this loop reduces the total weight of the path $s \to \cdots \to t$. Any algorithm that seeks to minimize a path's weight can do so by passing around the loop an infinite number of times. More generally, any G that contains a negative-weight cycle will produce such paths of infinite length. Worse, our algorithm will never terminate because it can always get a better solution by running more cycles.

This is clearly pathological behavior that we want to avoid. Avoiding it algorithmically would require having a way to detect the presence of and then avoid a negative-weight cycle, but including this capability will over-complicate our exposition on SSSP algorithms (at first). Instead, we simply redefine *shortest path* to mean a path with minimum weight *and* that never touches a negative cycle. If no such path from s to t exists, then we say there is no shortest path between them. (Suppose G is undirected and contains a single negative-weight edge; can SSSP be applied? Why or why not?)

1.1.2 A general SSSP algorithm

As with our `Search-Tree` algorithm, we will describe a general framework for solving the SSSP problem, and then show how particular SSSP algorithms are special cases.

Formally, a SSSP algorithm takes as input a graph $G(V, E)$ and a source vertex $s \in V$. Let $w(u, v)$ return the weight of the edge $(u, v) \in E$. As with the `Search-Tree` algorithm, we need to store two pieces of auxiliary information about each vertex, which we store in two arrays:[3]

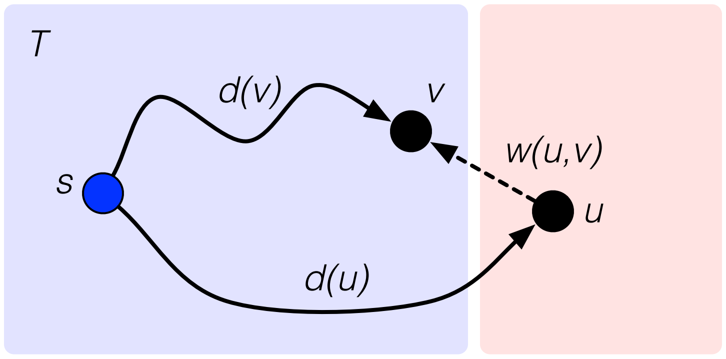
dist(v)	returns the length of the tentative shortest path between s and v
pred(v)	returns the predecessor of v in this tentative shortest-path (the set of these represent the single-source shortest path tree)

Their initialization is simply

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

[3]For notational convenience, dist(x) will also be referred to as $d(x)$, and pred(x) as $p(x)$.

Now, we make a crucial observation. Assume we have some tree T on the vertices of G, but that T is not a solution to the SSSP problem. What does this imply about the structure of T? It means that for some vertex v, there exists a shorter path from s to v than is contained in T. More precisely, it implies the existence of some edge (u, v) such that $\text{dist}(u) + w(u, v) < \text{dist}(v)$. Call such a (u, v) a *tense* edge. Thus, a correct solution T to the SSSP problem is a tree with *no tense edges*.



If (u, v) is tense, then the existing path from s to v in T is incorrect (i.e., cannot be part of a solution to the SSSP problem) 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 G and relax *it (make the path run through u instead).*

To relax an edge, we must update the tree T and the distance to v 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 yet specified exactly how we detect which edges are tense or in what order we choose to relax them, any algorithm that relaxes tense edges will eventually halt so long as relaxing an edge does not always produce more tense edges. The output of such an algorithm must then be a correct SSSP tree because such a tree, by definition, can have no tense edges.

There are many ways we could go about finding tense edges to relax. To avoid being too specific yet, here is a generic approach: we will maintain a *set*[4] of vertices Q; whenever we remove some vertex u from Q, we examine each of its outgoing edges (u, v) and check if it can be relaxed. (We defer the question of the ordering for now, which we will return to below.) If we successfully relax some edge (u, v), then we place v in the set.

Initially, $Q = \{s\}$ and the distances from s to all vertices $V - s$ are set to infinity; thus, all of the edges $(s, x) \in E$ are *tense*. (Or more specifically, 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 is pseudocode for this generic algorithm:

```
Generic-SSSP(G,s) {
   dist(s) = 0                  // initialize distance and tree auxiliary arrays
   pred(s) = NULL               //
   for all vertices v != s {    //
      dist(v) = INF             //
      pred(v) = NULL            //
   }                            //
   Q = emptySet()               // initialize 'set' data structure

   Q.add(s)                     // add source vertex to the set
   while Q.notempty() {         // grow the tree
      u = Q.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)          //
            Q.add(v)            // add v to the set
}}}}
```

1.1.3 Correctness

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

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

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

[4]This is a clue about both the running time and the deferred implementation choices: a "set" is an abstract data structure and hence the cost in time and space to fulfill find, remove or add item operations depends on the particular implementation. Dictionary ADTs can be used to maintain a set.

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 $\text{dist}(v) \leq w(s \rightsquigarrow v)$ 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). □

Thus, `Generic-SSSP` is a correct algorithm for solving the SSSP problem. What remains is to decide how to manage the set of vertices Q. Different choices produce 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 using a min-heap data structure[5] 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., $\text{dist}(v)$, from s to v. This implies that the algorithm evaluates the vertices in increasing order of their tentative shortest-path distance from s and thus each vertex is evaluated at most k_{in} times. Each time we evaluate a vertex, we then evaluate each of its outgoing edges; thus, we relax each edge at most once. (The figure below 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.

[5]Recall that 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. And, recall that heaps can be easily implemented in an array if you know the maximum size it will have (or, if you don't, you can use dynamic re-sizing, so long as you amortize that cost appropriately). In general, heaps are trees, and the two main flavors are Binomial and Fibonacci heaps (see CLRS Chapters 19 and 20). 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.

If we use a classic binary min-heap, then each heap update takes $O(\log V)$ and so the running time is $O(V + E \log V)$. 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.

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

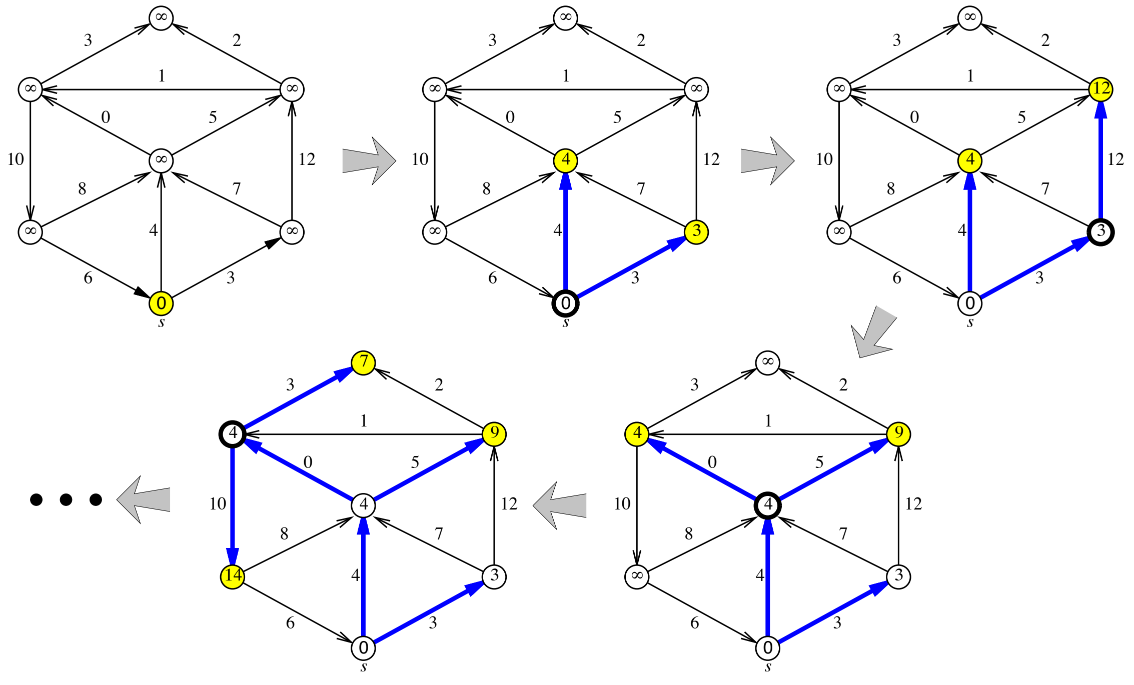


Figure 1: Four phases of Dijkstra's algorithm, run on a graph with no negative edges. At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned. The bold edges describe the evolving shortest path tree. (Note that there is one distance typo in this figure — can you find it? Image from Jeff Erickson'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. The figure below 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 $dist(v) + \texttt{Guess-Distance}(v, t)$. 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.

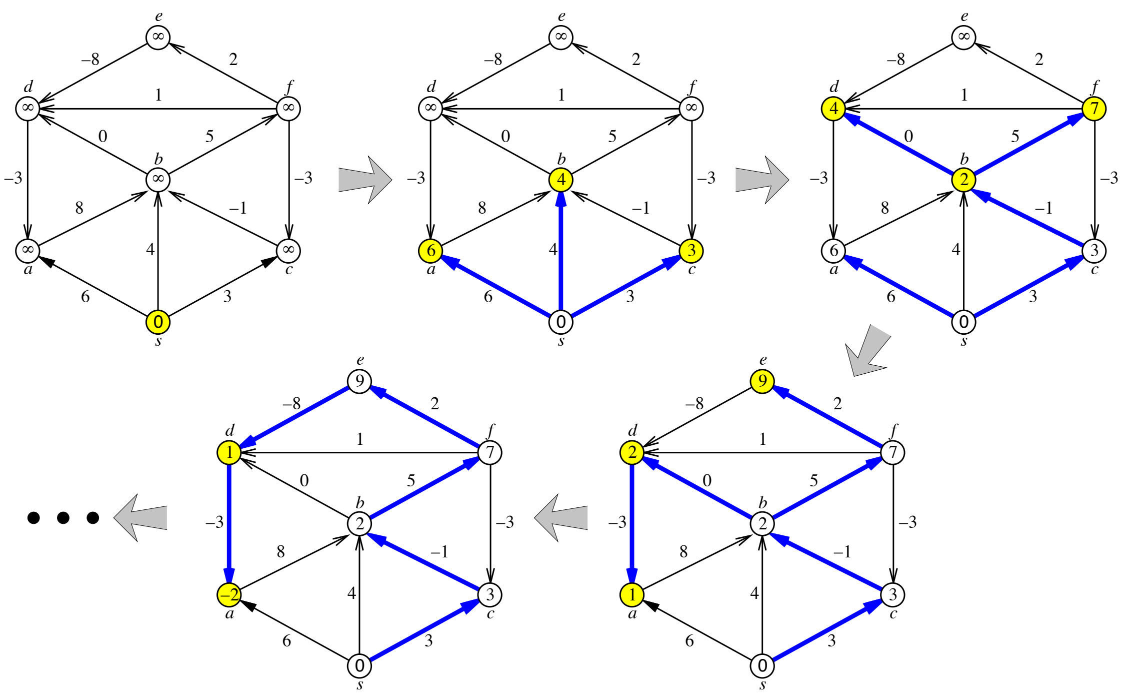


Figure 2: Four phases of the Bellman-Ford algorithm, run on a directed graph with some negative edges. Nodes are taken from the queue in the order $s \diamond abc \diamond dfb \diamond aed \diamond da \diamond \diamond$, where $\diamond$ is the token. Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path. (Note that there are two distance typos in this figure – can you find them? Image from Jeff Erickson's lecture notes.)

2 On your own

1. Read Chapter 24, Single-Source Shortest Paths