Design and Analysis of Algorithms
Lecture 7
Prof. Aaron Clauset

CSCI 5454
9 February 2011

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. The generalization to directed graphs is trickier; do you see why?

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. (We'll see more about spanning trees next week.)

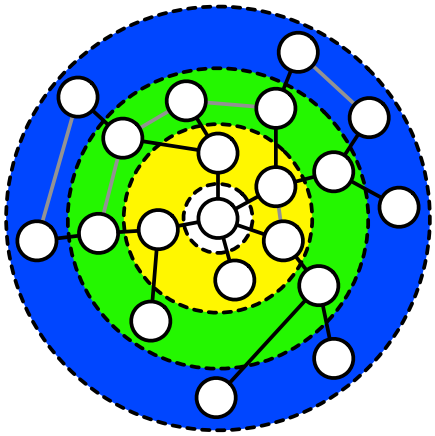


Figure 1: A BFS tree on a simple network, show the way BFS explores the network in layers; tree edges are shown in black and non-tree edges are shown in grey.

1.1 All-Pairs-Shortest-Path

Suppose we wanted to know the shortest path from point A to B in a graph, but we didn't know beforehand what vertices are A and B? This problem is very similar to what Google Maps solves when you ask it for driving directions.[1] 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 shortest path σ_{st} from s to t, which minimizes the function

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

All solutions to this problem solve, for each possible choice of s, the *single-source shortest paths* or SSSP problem, so the running time is simply V times the running time of the SSSP algorithm. That is, given s, find the shortest paths from s to t, for all $t \in V$. If the network were undirected, we can simply use BFS to do this because the BFS tree is a shortest path tree, rooted at s. (When there are multiple shortest paths from s to some t, we the set of all shortest paths is not a tree; can you prove that if the shortest path from s to t is unique that the set forms a tree?)

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 tree, i.e., every pair of vertices $u, v \in V$ is reachable in 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

[1] 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) or it could be shortest time (duration of trip) or even some other function. 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 that reduce the remaining distance $d(x, B)$. But, that's not how people navigate road networks at all. What's the difference?

trees while there's at least one MST. They *can* be the same, but they don't have to be. Here's an example.

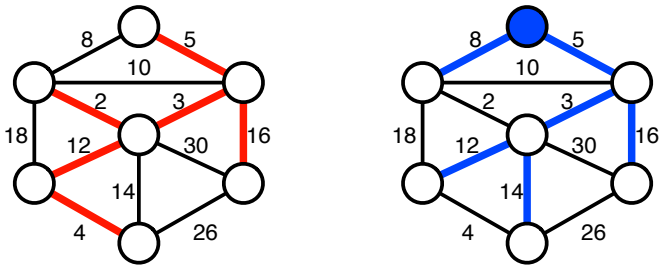
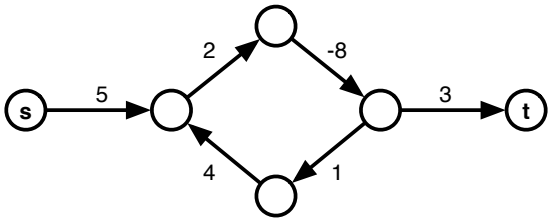


Figure 2: The left-hand figure shows a minimum spanning tree while the right-hand figure shows a single-source shortest path tree rooted at the blue node.

1.1.2 Negative cycles

What happens if one of our edges has *negative* weight? For instance, consider this small graph:



In this case, the minimum weight path from s to t has weight $-\infty$, because we can run around the loop as many times as we like, and the net weight of the loop is -1. This is inconvenient. To circumvent this problem, while still preserving generality, we can redefine "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, we need to preserve this requirement, i.e., we disallow cycling back and forth across a negative-weight edge.)

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:

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

Now, define an edge (u, v) to be *tense* if $dist(u) + w(u, v) < dist(v)$. If (u, v) is tense, then the tentative shortest path from s to v is incorrect because the path from s to u and then (u, v) is shorter. Our strategy for solving the SSSP problem will be to find a tense edge in the graph and then *relax* it, i.e., we make the path run through u instead:

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

If there are no tense edges, then the algorithm is finished and we have a shortest path tree.

Importantly, we haven't unspecified exactly how we detect which edges are tense and what in order we choose to relax them. There are many possible choices we could make here, and to avoid being too specific yet, here's a generic approach: let us maintain a *set* of vertices[2] (initially, this set will contain only the source vertex s); whenever we remove some vertex u from the set, 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 can revisit that in a moment.) If we successfully relax some edge (u, v), then we place v in the set. That's it.

Here's pseudocode for the algorithm:

[2]This should be an immediate clue about both the running time and future implementation chocies: 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.

```
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(u)             // 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 does what we claim. In order to do the running time of the algorithm, however, we'll need to specify *which* edges we relax.

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, 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 some positive amount.

So, `Generic-SSSP` does what we claim, and now we need to decide how to manage the set of vertices. Different choices here lead to different algorithms. 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[3] then we obtain Dijkstra's algorithm. In this case, we can show by induction that the vertices are scanned in increasing order of their shortest-path distance from s, and thus it follows that each vertex is scanned at most once, and that each edge is relaxed at most once. Figure 3 shows an example of running Dijkstra's algorithm on a small graph.

Here, the key of some vertex v in the heap is represents the tentative distance from s to v. When we modify the distance $dist(v)$, we modify the key in the heap. Because we relax each edge at most once, we make at most $|E|$ modifications ("decrease key") to keys in the heap, each of which operation takes amortized $O(1)$ time. The $|V|$ inserts we make take amortized $O(\log |V|)$ time each. Thus, this efficient implementation of Dijkstra's algorithm has a running time of $O(E + V \log V)$. (How much space does it take?)

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

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, which makes it useful for detecting them. 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. (So, 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.

[3] We haven't covered heaps in the lectures, but they're a very important type of data structure. A "min-heap" allow us to find the smallest key in the set very quickly; 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.

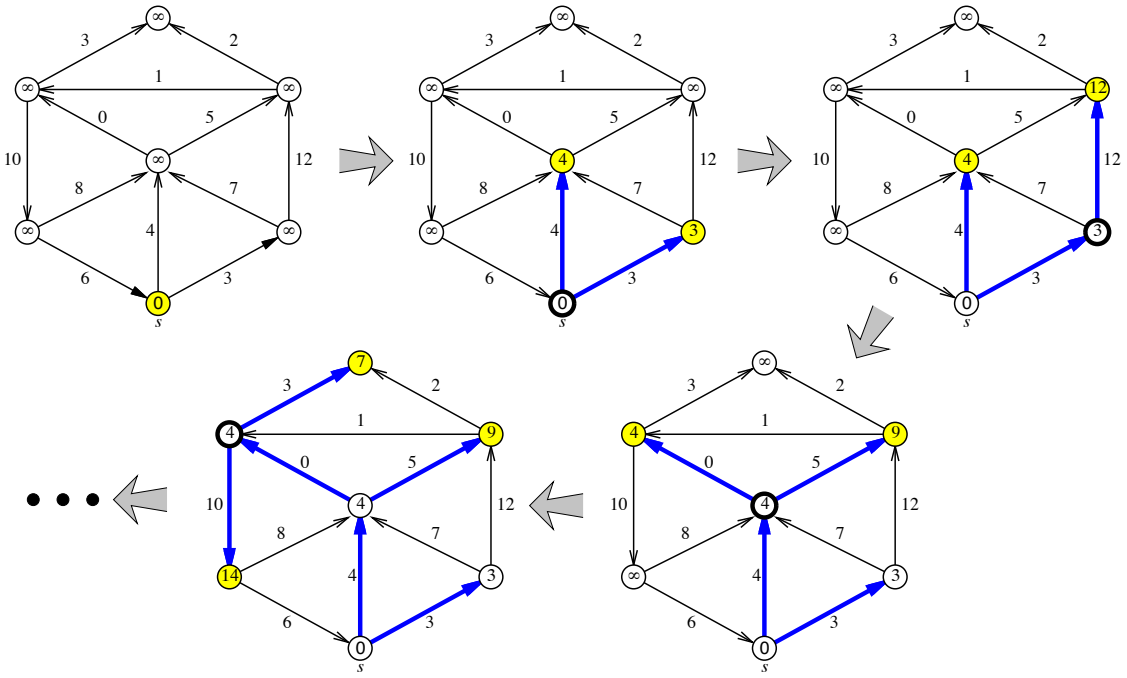


Figure 3: Four phases of Dijkstra's algorithm, run on a graph with no negative edges. At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned. The bold edges describe the evolving shortest path tree. (Taken from Jeff Ericson's lecture notes.)

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. Thus, in each phase, we scan each vertex at most once, and thus we relax each edge as most once, and 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.

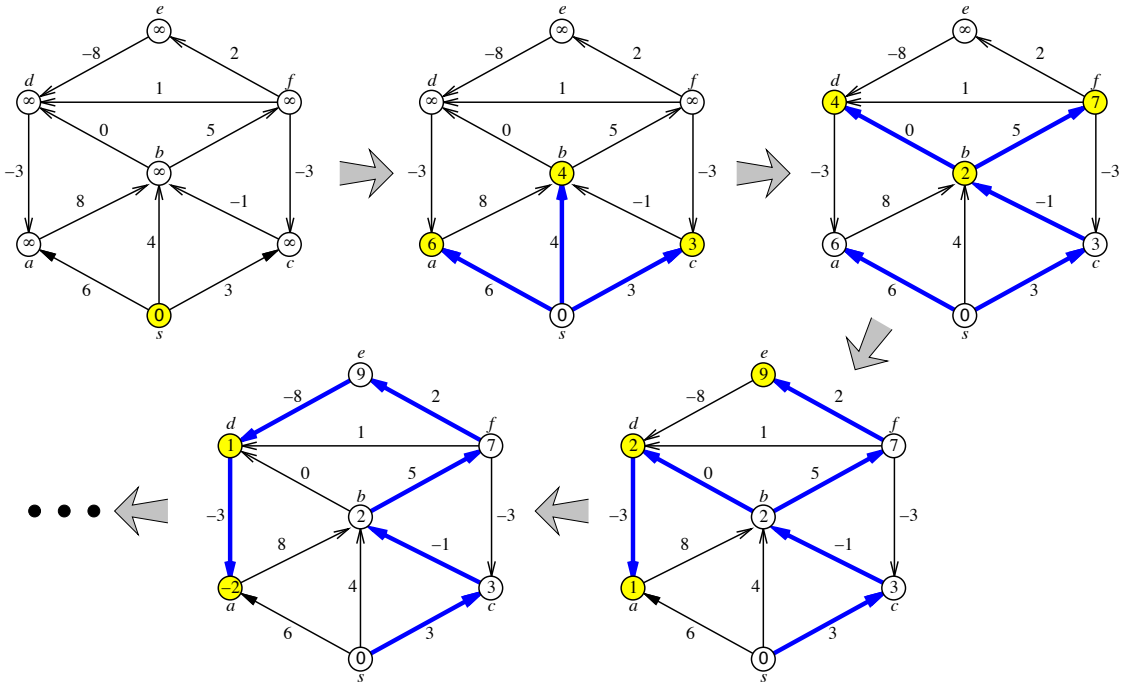


Figure 4: Four phases of the Bellman-Ford algorithm, run on a directed graph with some negative edges. Nodes are taken from the queue in the order $s \diamond abc \diamond dfb \diamond aed \diamond da \diamond \diamond$, where $\diamond$ is the token. Shaded vertices are in the queue at the end of each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

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)
3. Reminder: Email me your team's ranked preferences for lecture topics by Monday.