CMSC 451: Lecture 5
Graph Shortest Paths: Dijkstra and Bellman-Ford

Tuesday, Sep 12, 2017

Reading: Sects. 4.4 and 6.8 in KT.

Shortest Paths: Today we consider the problem of computing shortest paths in a directed graph. We are given a digraph $G = (V, E)$ and a source vertex $s \in V$, and we want to compute the shortest path from s to every other vertex in G. This is called the *single source shortest path problem*. The algorithms we will present work for undirected graphs as well, by simply assuming that each undirected edge consists of two directed edges going in opposite directions.

In general we assume that the graph is *weighted*, meaning that each edge $(u, v) \in E$ has a numeric edge weight $w(u, v)$. The *cost* of a path is the sum of edge weights along the path. Define the *distance* from any vertex u to any vertex v to be the minimum cost over all the paths from u to v. We denote this by $\delta(u, v)$. Thus, we are interested in computing $\delta(s, v)$ for all $v \in V$. We assume that every vertex has a trivial path of cost zero to itself, and hence $\delta(v, v) = 0$, for all $v \in V$.

Background: In earlier classes you may have learned about *breadth-first search*. This algorithm runs in $O(n + m)$ algorithm for finding shortest paths from a single source vertex to all other vertices, assuming that the graph has no edge weights, or equivalently, that the weights are equal to unity. Thus, distance is just the number of edges on a path. This algorithm uses a first-in, first-out queue to process the vertices, visiting s first, then all the vertices accessible by a single edge, then all the vertices accessible by two edges, an so on, until all the vertices reachable from s have been processed. Today we will consider two algorithms that work for weighted graphs.

Since edge weights usually correspond to distances or travel times, it is typically the case that edge weights are positive, or at least, they are nonnegative. However, there are applications where negative edge weights make sense. For example, if an edge denotes a financial trade (buying or selling commodities), the cost may either be positive (a loss) or negative (a gain). In this context, the shortest path corresponds to a sequence of transactions that minimizes loss (or equivalently, maximizes gain).

In general it is possible to define shorest paths even for graphs with negative edge weights, but it should be noted that shortest paths are not well defined if the graph has *negative-cost cycles*. The reason is that the cost could be made arbitrarily small by traversing the cycle over and over. We will assume that the graphs we will be working with have no negative-cost cycles. Since allowing negative edge weights makes the problem more difficult to solve, we will consider these two variants separately.

Dijkstra's Algorithm: We first present a simple greedy algorithm for the single-source problem, which assumes that the edge weights are nonnegative. The algorithm, called *Dijkstra's algorithm*, was invented by the famous Dutch computer scientist, Edsger Dijkstra in 1956 (and published later in 1959). It is among the most famous algorithms in Computer Science.

In our presentation of the algorithm, we will stress the task of computing just the distance from the source to each vertex (not the path itself). We will store a *predecessor link* with each vertex, which points to way back to the source. Thus, the actual path can be found by traversing the predecessor links and reversing the resulting path. Since we store one predecessor link per vertex, the total space needed to store all the shortest paths is just $O(n)$.

Shortest Paths and Relaxation: The basic structure of Dijkstra's algorithm is to maintain an *estimate* of the shortest path for each vertex, call this $d[v]$. Intuitively $d[v]$ stores the length of the shortest path from s to v *that the algorithm currently knows of.* Indeed, there will always exist a path of length $d[v]$, but it might not be the ultimate shortest path. Initially, we know of no paths, so $d[v] = \infty$, and $d[s] = 0$. As the algorithm proceeds and sees more and more vertices, it updates $d[v]$ for each vertex in the graph, until all the $d[v]$ values "converge" to the true shortest distances.

The process by which an estimate is updated is sometimes called *relaxation*. Here is how relaxation works. Intuitively, if you can see that your solution is not yet reached an optimum value, then push it a little closer to the optimum. In particular, if you discover a path from s to v shorter than $d[v]$, then you need to update $d[v]$. This notion is common to many optimization algorithms.

Consider an edge from a vertex u to v whose weight is $w(u, v)$. Suppose that we have already computed current estimates on $d[u]$ and $d[v]$. We know that there is a path from s to u of weight $d[u]$. By taking this path and following it with the edge (u, v) we obtain a path to v of length $d[u] + w(u, v)$. If this path is better than the existing path of length $d[v]$ to v, we should update $d[v]$ to the value $d[u] + w(u, v)$ (see Fig. 3.) We should also remember that the shortest path to v passes through u, which we do by setting pred$[v]$ to u (see the code block below).

```
relax(u, v) {
    if (d[u] + w(u, v) < d[v]) {    // is the path through u shorter?
        d[v] = d[u] + w(u, v)       // yes, then take it
        pred[v] = u                 // record that we go through u
    }
}
```

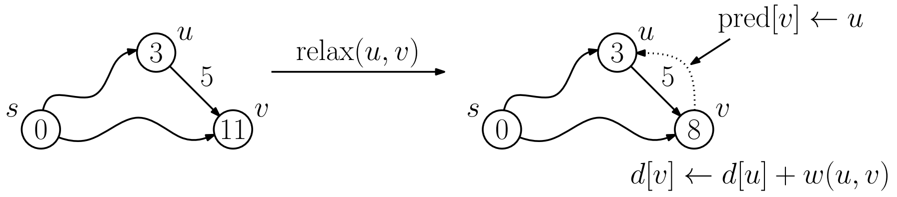

Fig. 1: Relaxation.

Observe that whenever we set $d[v]$ to a finite value, there is always evidence of a path of that length. Therefore $d[v] \geq \delta(s, v)$. If $d[v] = \delta(s, v)$, then further relaxations cannot change its value.

It is not hard to see that if we perform relax(u, v) repeatedly over all edges of the graph, the $d[v]$ values will eventually converge to the final true distance value from s. (This remark will reemerge when we discuss the Bellman-Ford algorithm later.) The cleverness of any shortest path algorithm is to perform the updates in a judicious manner, so the convergence is as fast as possible. In particular, the best possible would be to order relaxation operations in such a way that each edge is relaxed exactly once. Assuming that the edge weights are nonnegative, Dijkstra's algorithm achieves this objective.[1]

Dijkstra's Algorithm: Dijkstra's algorithm operates by maintaining a subset of vertices, $S \subseteq V$, for which we claim we "know" the true distance, that is $d[v] = \delta(s, v)$. Initially $S = \emptyset$, the empty set, and we set $d[s] = 0$ and all others to $+\infty$. One by one, we select vertices from $V \setminus S$ to add to S. (If you haven't seen it before, the notation "$A \setminus B$" means the set A excluding the elements of set B. Thus $V \setminus S$ consists of the vertices that are not in S.)

The set S can be implemented using an array of vertex marks. Initially all vertices are marked as "undiscovered," and we set $\text{mark}[v] = \text{finished}$ to indicate that $v \in S$.

How do we select which vertex among the vertices of $V \setminus S$ to add next to S? Here is where greedy selection comes in. Dijkstra recognized that the best (greediest) way is to process the vertex with the smallest d-value. That is, we take the unprocessed vertex that is closest (by our estimate) to s. This way, whenever a relaxation is being performed, it is possible to infer that result of the relaxation yields the final distance value. Later we will justify why this is the proper choice.

In order to perform this selection efficiently, we store the vertices of $V \setminus S$ in a *priority queue* (e.g. a heap), where the key value of each vertex u is $d[u]$. We will need to make use of three basic operations that are provided by the priority queue:

Build: Create a priority queue from a list of n elements, each with an associated key value.

Extract min: Remove (and return a reference to) the element with the smallest key value.

Decrease key: Given a reference to an element in the priority queue, decrease its key value to a specified value, and reorganize if needed.

For example, using a standard binary heap (as in heapsort) the first operation can be done in $O(n)$ time, and ther other two can be done in $O(\log n)$ time each. Dijkstra's algorithm is given in the code block below, and see Fig. 1 for an example.

Notice that the marking is not really used by the algorithm, but it has been included to make the connection with the correctness proof a little clearer.

To analyze Dijkstra's algorithm, recall that $n = |V|$ and $m = |E|$. We account for the time spent on each vertex after it is extracted from the priority queue. It takes $O(\log n)$ to extract this vertex from the queue. For each incident edge, we spend potentially $O(\log n)$ time if we need to decrease the key of the neighboring vertex. Thus the time is $O(\log n + \deg(u) \cdot \log n)$ time. The other steps of the update run in constant time. Recalling that the sum of degrees

[1]Note, by the way that while this objective is optimal in the worst case, there are instances where you might hope for much better performance. For example, given a cartographic road map of the entire United States, computing the shortest path between two locations near Washington DC should not require relaxing every edge of this road map. A better approach to this problem is provided by another greedy algorithm, called A^*-search.

Dijkstra's Algorithm

```
dijkstra(G,w,s) {
    for each (u in V) {                     // initialization
        d[u] = +infinity
        mark[u] = undiscovered
        pred[u] = null
    }
    d[s] = 0                                // distance to source is 0
    Q = a priority queue of all vertices u sorted by d[u]
    while (Q is nonEmpty) {                 // until all vertices processed
        u = extract vertex with minimum d[u] from Q
        for each (v in Adj[u]) {            // relax all outgoing edges from u
            if (d[u] + w(u,v) < d[v]) {
                d[v] = d[u] + w(u,v)
                decrease v's key in Q to d[v]
                pred[v] = u
            }
        }
        mark[u] = finished
    }
    [The pred pointers define an ``inverted'' shortest path tree]
}
```

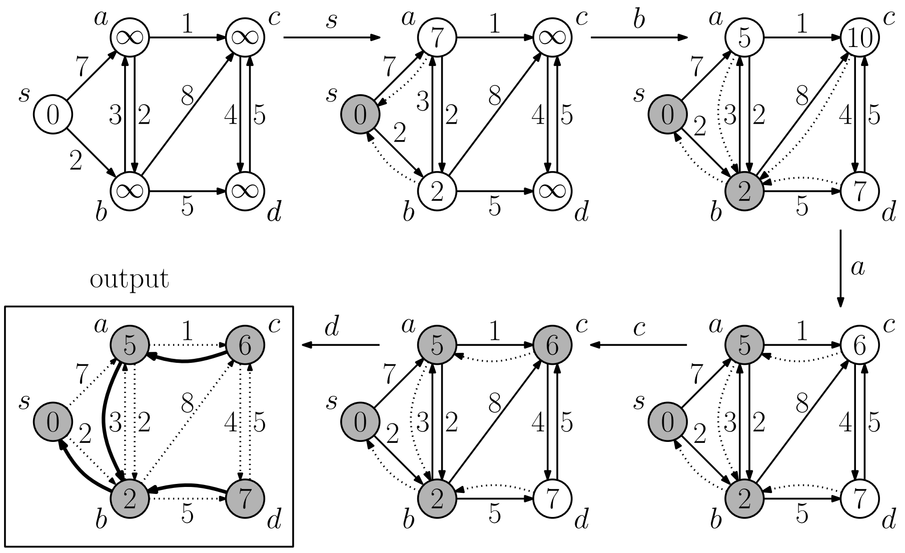

Fig. 2: Dijkstra's Algorithm example.

of the vertices in a graph is $O(m)$, the overall running time is given by $T(n,m)$, where

$$\begin{aligned} T(n,m) &= \sum_{u\in V}(\log n + \deg(u)\cdot\log n) = \sum_{u\in V}(1+\deg(u))\log n \\ &= \log n\sum_{u\in V}(1+\deg(u)) = (\log n)(n+2m) = \Theta((n+m)\log n). \end{aligned}$$

Since G is connected, n is asymptotically no greater than m, so this is $O(m\log n)$. If you use a "smarter" heap (in particular, a Fibonacci heap, which supports very fast decrease-key operations, the running time is only $O(n\log n + m)$.

Correctness: Recall that $d[v]$ is the distance value assigned to vertex v by Dijkstra's algorithm, and let $\delta(s,v)$ denote the length of the true shortest path from s to v. To establish correctness, we need to show that $d[v] = \delta(s,v)$on termination. This is a consequence of the following lemma, which states that once a vertex u has been added to S (i.e., has been marked "finished"), $d[u]$ is the true shortest distance from s to u.

Lemma: When a vertex u is added to S, $d[u] = \delta(s,u)$.

Proof: Suppose to the contrary that at some point Dijkstra's algorithm *first* attempts to add a vertex u to S for which $d[u] \neq \delta(s,u)$. By our observations about relaxation, $d[u]$ is never less than $\delta(s,u)$, thus we have $d[u] > \delta(s,u)$. Consider the situation just prior to the insertion of u into S, and consider the true shortest path from s to u. Because $s \in S$ and $u \in V \setminus S$, at some point this path must first jump out of S. Let (x,y) be the first edge taken by the shortest path, where $x \in S$ and $y \in V \setminus S$ (see Fig. 3). (Note that it may be that $x = s$ and/or $y = u$).

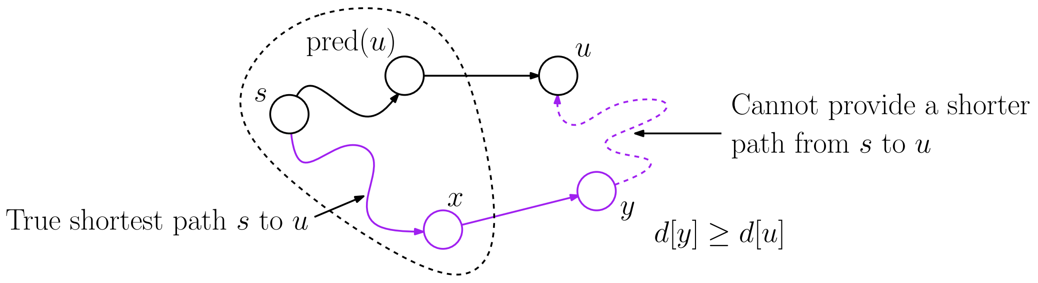
pred(u)
u
s
Cannot provide a shorter path from s to u
x
y
True shortest path s to u
$d[y] \geq d[u]$

Fig. 3: Correctness of Dijkstra's Algorithm.

Because u is the first vertex where we made a mistake and since x was already processed, we have $d[x] = \delta(s,x)$. Since we applied relaxation to x when it was processed, we must have

$$d[y] = d[x] + w(x,y) = \delta(s,x) + w(x,y) = \delta(s,y).$$

Since y appears before u along the shortest path and edge weights are nonnegative, we have $\delta(s,y) \leq \delta(s,u)$. Also, because u (not y) was chosen next for processing, we know that $d[u] \leq d[y]$. Putting this together, we have

$$\delta(s,u) < d[u] \leq d[y] = \delta(s,y) \leq \delta(s,u).$$

Clearly we cannot have $\delta(s,u) < \delta(s,u)$, which establishes the desired contradiction.

Bellman-Ford Algorithm: Let us now consider the question of how to solve the single-source shortest path problem when negative edge costs are allowed. Recall that we need to assume that the graph has no negative-cost cycles. (As an exercise, you are encouraged to think about how to detect whether this assumption is violated.)

We shall present the *Bellman-Ford algorithm*, which solves this problem. This algorithm slightly predates Dijkstra's algorithm, which was discovered in 1956. The algorithm was originally due to Alfonso Shimbel in 1955. It was rediscovered by Ford in 1956 and then by Bellman in 1958. This algorithm is slower than the best implementations of Dijkstra's algorithm in the worst case. Dijkstra's algorithm runs in time $O(m \log n)$, whereas Bellman-Forst runs in time $O(nm)$.

The idea behind the Bellman-Ford algorithm is quite simple. We initialize $d[s] \leftarrow 0$ and $d[v] \leftarrow \infty$ for all other vertices. We know that for each edge $(u, v) \in E$, the operation relax(u, v) propagates shortest-path information outwards from s. So, let's simply apply this operation repeatedly along each edge of E until the d-values converge.

Bellman-Ford Algorithm

```
bellman-ford(G, w, s) {
    for each (u in V) {                 // initialization
        d[u] = +infinity
        pred[u] = null
    }
    d[s] = 0
    repeat {                            // repeat until convergence
        converged = true
        for each ((u, v) in E) {        // relax along each edge
            if (d[u] + w(u, v) < d[v]) {
                d[v] = d[u] + w(u,v)
                pred[v] = u
                converged = false
            }
        }
    } until (converged)
    [The pred pointers define an ‘‘inverted’’ shortest path tree]
}
```

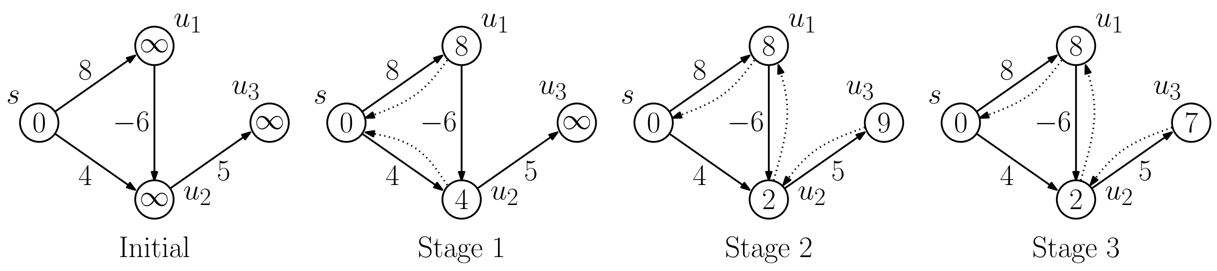

Fig. 4: Bellman-Ford Algorithm. (The algorithm will run for one more stage, but nothing will change, and it will terminate.)

Correctness: The following lemma establishes the correctness of the algorithm.

> **Lemma:** On termination of the Bellman-Ford algorithm, for all $v \in V$, $d[v]$ contains the correct distance from s to v.
>
> **Proof:** We assert first that *if* the algorithm converges, then $d[v] = \delta(s, v)$ for all $v \in V$. As observed earlier, $d[v]$ contains the cost of *some* path from s to v, so we have $d[v] \geq \delta(s, v)$. We will prove that $d[v] \leq \delta(s, v)$ by induction on the length of the shortest path from s to v. In particular, if the shortest path from s to v consists of i edges, then after the ith iteration of the repeat-loop, $d[v] = \delta(s, v)$.
>
> For the basis case ($i = 0$) the only vertex whose shortest path is of length zero is s itself. By our initialization code, $d[s] = 0$, and by definition of shortest paths $\delta(s, s) = 0$, so we're done.
>
> For the general case ($i \geq 1$), consider any vertex v be any vertex such that the length of the shortest path from s to v consists of i edges. Let u be the vertex that immediately precedes v along this shortest path. It follows that (1) the shortest path from s to u is of length $i - 1$, and (2) $\delta(s, v) = \delta(s, u) + w(u, v)$. By the induction hypothesis, we know that after $i - 1$ iterations of the repeat-loop, we have $d[u] = \delta(s, u)$. After the ith iteration of the repeat-loop, when we consider the edge (u, v) we will update the value of $d[v]$ to
>
> $$d[v] \;\leq\; d[u] + w(u, v) \;=\; \delta(s, u) + w(u, v) \;=\; \delta(s, v),$$
>
> In conclusion, we have shown that $\delta(s, v) \leq d[v] \leq \delta(s, v)$, which implies that $d[v] = \delta(s, v)$. Which completes the proof.

Running time: Of course, all of the above is based on the assumption that the algorithm does indeed terminate. Next, we show that this will happen after at most n iterations of the repeat loop. We will make use of the following observation.

> **Lemma:** If G has no negative cost cycles, then there is a shortest path from s to any vertex v that does not repeat any vertex.
>
> **Proof:** Suppose to the contrary that the shortest path from s to v did repeat a vertex. Then there would be a cycle in the path. Since G has no negative cost cycles, we can remove this cycle from the path without increasing the path's total cost. If we repeat this for every repeated vertex, we will eventually have a path that contains no repetitions.
>
> **Corollary:** For each $v \in V$, there is a shortest path from s to v consisting of at most $n - 1$ edges (where $n = |V|$).

It follows immediately from the corollary that after $n - 1$ iterations, the d-values of all the vertices will have achieved their final values. After one more iteration, we will discover this fact, and the algorithm will terminate.

Since each relax operation required $O(1)$ time, each iteration of the repeat-loop takes time $O(m)$. Since the algorithm terminates after n iterations, the total running time is $O(nm)$. Note that the algorithm may actually terminate earlier.

Now, looping back to the question we asked early, can we modify this algorithm to detect whether the graph has a negative cost cycle? Hopefully the easy answer will occur to you. (If not, think about it a bit more.)