CMSC 451: Lecture 4
Graph Shortest Paths: Dijkstra and Bellman-Ford

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.

We assume that each edge $(u, v) \in E$ is associated with a numerical edge weight $w(u, v)$ (see Fig. 1(a)). (If the graph is not weighted in this manner, then we assume that each edge has a weight of 1.) The *cost* of a path is defined to be 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 $v \in V$ (see Fig. 1(b)). We assume that every vertex has a trivial path of cost zero to itself, and hence $\delta(v, v) = 0$, for all $v \in V$. The collection of shortest paths defines a directed tree rooted at s, called the *shortest-path tree* (see Fig. 1(c)).

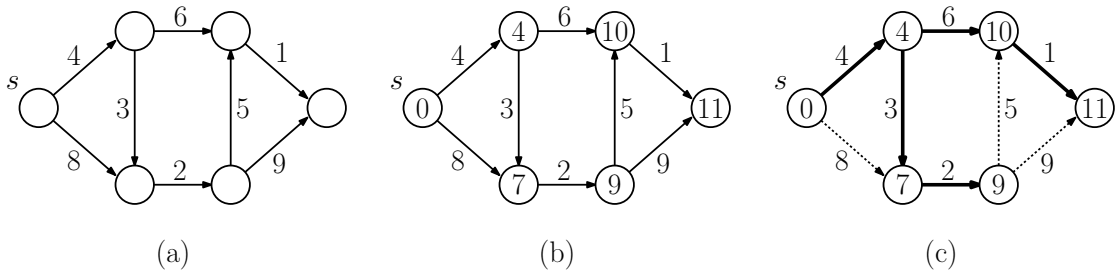

Fig. 1: (a) A weighted digraph G, (b) with vertices labeled with their distance $\delta(s, v)$, and (c) the shortest-path tree.

Preliminaries: If the edges of the graph are not weighted, there is an efficient algorithm, called *breadth-first search*, which solves the single-source shortest path problem. This algorithm runs in $O(n+m)$ time, where $n = |V|$ and $m = |E|$. This algorithm uses a first-in, first-out (FIFO) queue to process the vertices. The algorithm is very simple. It starts with a queue containing just the source vertex s, setting $d[s] \leftarrow 0$. It repeatedly dequeues the next vertex u. It then enqueues all of its neighbors v that have not already been visited and sets $d[v] \leftarrow d[u] + 1$. It stops when the queue is empty, and $d[u]$ is the distance from s to u.

In this lecture, we will focus on graphs with weighted edges. Since edge weights usually correspond to distances or travel times, it is common to assume that edges weights are positive, or at least, they are nonnegative. However, there are applications where negative edge weights make sense (see Fig. 2(a)). 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 shortest paths even for graphs with negative edge weights, but it should be noted that shortest paths may not be well defined if the graph has *negative-*

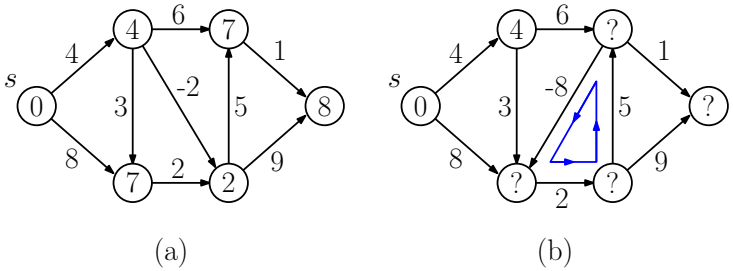

Fig. 2: (a) A weighted digraph with negative-cost edges and (b) a negative cost cycle. Observe that the shortest path from s to any of the vertices labeled "?" is undefined, because the distance can be made arbitrarily low by repeatedly looping through this cycle.

cost cycles. The reason is that the distance could be made arbitrarily small by traversing the cycle over and over (see Fig. 2(b)). You might wonder whether we could solve this by adding the constraint that it is not allowed to repeat vertices (that is, the path must be *simple*). This, however, makes the problem much harder solve.

We will discuss two algorithms. One that assumes nonnegative edge weights (Dijkstra) and one that allows for negative edge weights, but no negative-cost cycles (Bellman-Ford).

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 graph algorithms.

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 (see Fig. 3(b)). 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 $O(n)$.

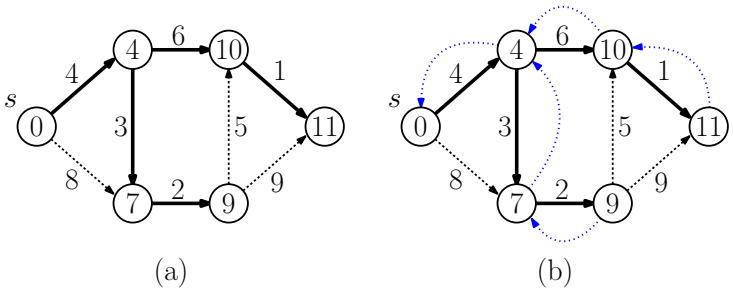

Fig. 3: (a) The shortest-path tree and (b) showing predecessor links (in blue).

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 we initialize $d[s] \leftarrow 0$, and for all other vertices v, we set $d[v] \leftarrow \infty$. 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*. Intuitively, if you can see that the distance estimate can be improved along an edge, then do so. Consider any edge (u, v). We know that there exists 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 current estimate $d[v]$, then we use it instead (see Fig. 6.)

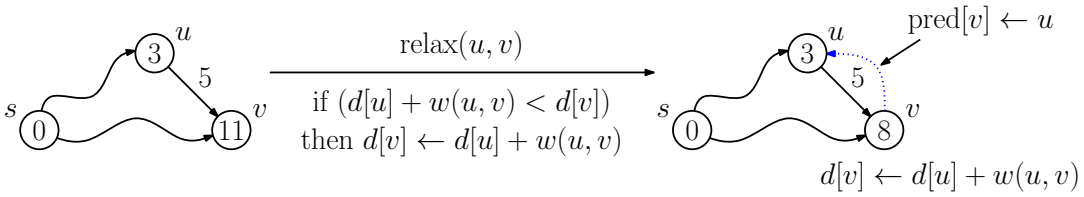

Fig. 4: Relaxation.

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 $d[v] = \delta(s, v)$, 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. Dijkstra's algorithm does this in a simple, greedy manner and is optimal in the worst case. There are more practical variants, such as A^*-search, which can do better for typical instances.

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.)

Paradoxically, the algorithm does not explicitly store the set S. Rather, we store all the vertices that are *not* in S in a priority queue, sorted by d-values. We greedily extract the "closest" vertex from the queue to be processed next, that is, the vertex with the smallest d-value. (Later we will justify why this is the proper choice.) We then propagate distance information forward by applying the relax operation on all of the outgoing edges from this selected vertex. We assume that our priority queue supports the following operations efficiently:

Build: Create a priority queue from a set 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 build operation can be performed in $O(n)$ time, and the other two can be done in $O(\log n)$ time each. The algorithm is given in the code block below. An example is shown in Fig. 4.

Dijkstra's Algorithm

```
dijkstra(G=(V,E), w, s) {
    for each (u in V) {                     // initialization
        d[u] = +infinity;  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]) {     // relax(u,v)
                d[v] = d[u] + w(u,v)
                pred[v] = u
                decrease v's key in Q to d[v]
            }
        }
    }
    [The pred pointers define an inverted shortest-path tree]
}
```

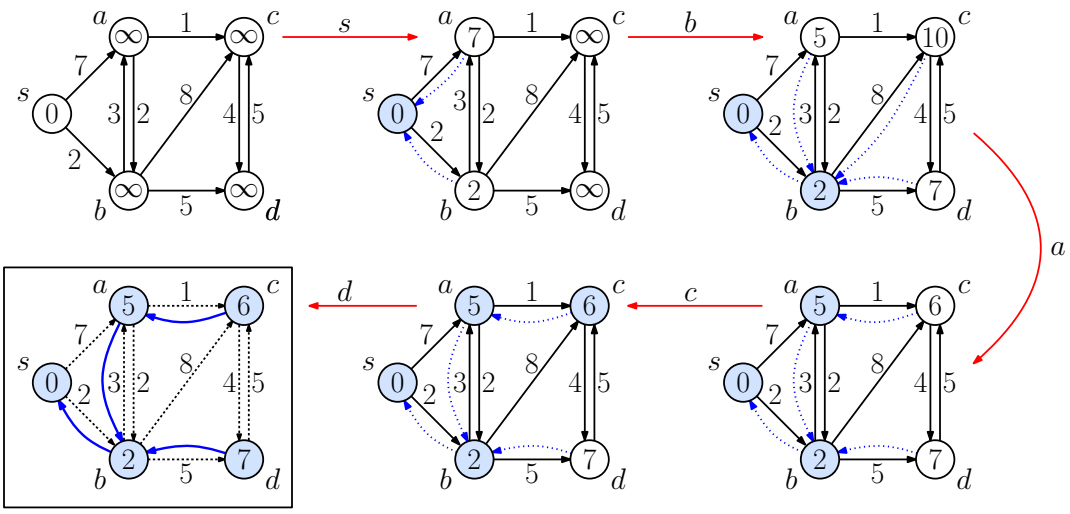

Fig. 5: Dijkstra's Algorithm example. Processed vertices are shaded in blue and broken blue links are predecessor links. The red arrows indicate the vertex being processed, which is always the unprocessed (white) vertex having the smallest d-value.

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 on termination, $d[v] = \delta(s, v)$. This is a consequence of the following lemma, which states that when we finish processing a vertex, its distance value is correct.

Lemma: At all times and for all vertices $u \in V$, if $d[u] \neq \infty$, there exists a path of this cost. After u has been processed, $d[u] = \delta(s, u)$.

Proof: The first assertion follows by induction and the fact that $d[u]$ values change through the relax operation, which propagates a d-value (which represents an actual path cost) through one more edge. Thus, there is a path of cost $d[u]$. This implies that $d[u] \geq \delta(s, u)$.

For the second assertion, suppose to the contrary that this is not true. Consider the *first* finished vertex u for which this fails to hold, that is, $d[u] \neq \delta(s, u)$.Since $d[u] \geq \delta(s, u)$, we have $d[u] > \delta(s, u)$. Let S denote the set of processed vertices, just prior to processing u. The true shortest path from s to u must exit S along some edge (x, y), where $x \in S$ and $y \notin S$ (see Fig. 6). (Note that it may be that $x = s$ and/or $y = u$).

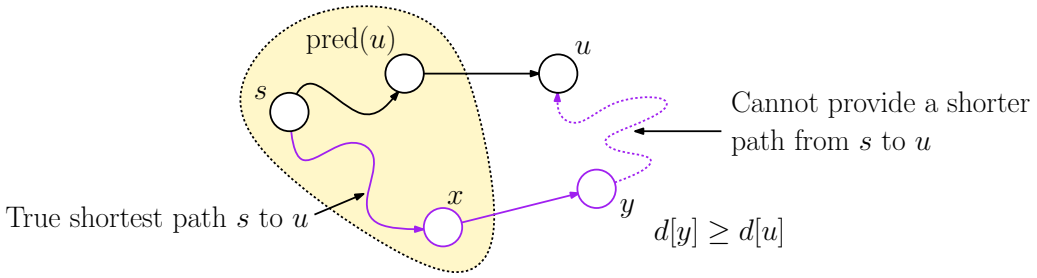

Fig. 6: Correctness of Dijkstra's Algorithm.

Because u is the first vertex where we made a mistake, and since x was already processed, we have $d[x] = \delta(s, x)$. When the algorithm processes x, it performs relax(x, y), which implies that

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

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.

Running Time: 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 $O(\log n + \text{out-deg}(u) \cdot \log n)$ time. The other steps of the update run in constant time. Recalling that the sum of degrees of the vertices in a graph is $O(m)$, the overall running time is given by $T(n, m)$, where

$$\begin{aligned} T(n, m) \;&=\; \sum_{u \in V} (\log n + \text{out-deg}(u) \cdot \log n) \;=\; \sum_{u \in V} (1 + \text{out-deg}(u)) \log n \\ &=\; \log n \sum_{u \in V} (1 + \text{out-deg}(u)) \;=\; (\log n)(n + m) \;=\; O((n + m) \log n). \end{aligned}$$

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 the decrease-key operation in $O(1)$ amortized time, the running time decreases to only $O(n \log n + m)$.

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. The algorithm was originally due to Alfonso Shimbel in 1955. (This slightly predates Dijkstra's algorithm, which dates to 1956.) It was rediscovered by Ford in 1956, and then by Bellman in 1958. (And for some reason, poor Shimbel was not credited with the name!) 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-Ford runs in time $O(nm)$. Note, however, that this is the worst case for Bellman-Ford. It can run as fast as $O(n)$ time, if you are extremely lucky.

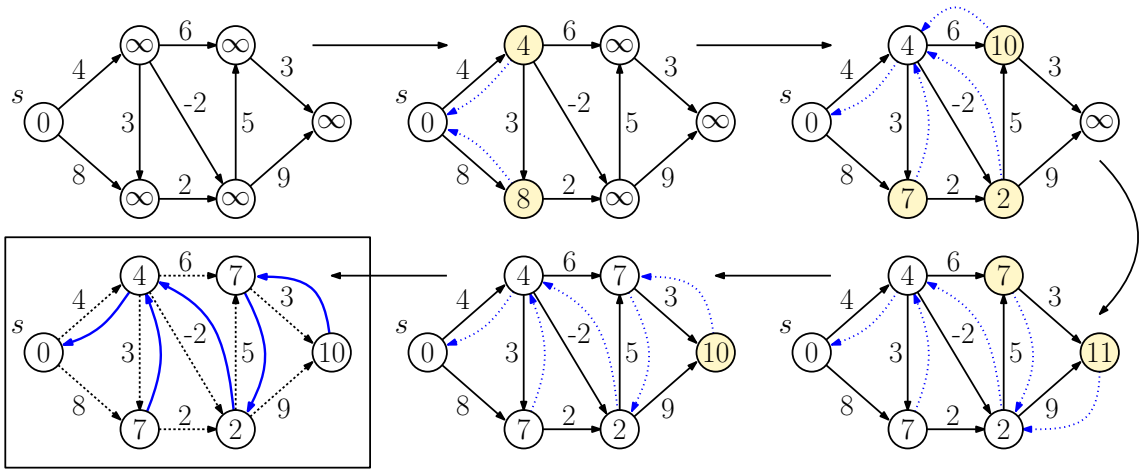

Fig. 7: Bellman-Ford Algorithm. (In this example, we assume that all relaxations happen "in parallel," which means that they are based on the d-values from the previous stage. The algorithm does not do this, but it does not affect its correctness. Vertices whose d-values were changed are shaded in yellow.)

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 shorest-path information outwards from s. So, let's simply apply this operation repeatedly along each edge of E until the d-values converge. The algorithm is shown in the following code block. See Fig. 7 for an example.

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

Lemma: If G has no negative-cost cycles, Bellman-Ford will terminate, and on termination of the Bellman-Ford algorithm, for all $v \in V$, $d[v]$ contains the correct distance from s to v.

Bellman-Ford Algorithm

```
bellman-ford(G=(V,E), 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]
}
```

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 $d[v] = \delta(s, v)$.

To see that the algorithm converges, observe that if a vertex is not reachable from s, then its d-value never changes, and it does not affect convergence. Otherwise, when we first reach a vertex on a path from s, the relax process sets its distance to a finite value. After this, each time the algorithm fails to converge, at least one vertex's d-value has strictly decreased. This can happen only a finite number of times, so the algorithm eventually converges.

Running time: Observe that time through the for-loop of the algorithm visits each edge once and

spends constant time, so the for-loop takes $O(m)$ time, where $m = |E|$. To show that the algorithm runs in $O(nm)$ time, it suffices to show that we go through the outer while loop at most $n = |V|$ times.

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 some vertex u. Thus, the path has the form $s \rightsquigarrow u \rightsquigarrow u \rightsquigarrow v$. Since G has no negative cost cycles, we can remove the cycle $u \rightsquigarrow u$ 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 $n = |V|$).

Now, go back and review the induction from the correctness proof for Bellman-Ford. It shows that after i iterations of the outer while-loop, all the vertices whose shortest paths consist of i or fewer edges have the correct d-values. By the corollary, the algorithm terminates in at most $n - 1$ iterations, so the overall running time is $O((n - 1)m) = O(nm)$. As observed earlier, this the worst case. The running time can be smaller.