Colorado CSCI 5454: Algorithms **Fall 2020**

Shortest Paths

Instructor: Bo Waggoner *Lecture 3*

This lecture covers shortest-path algorithms: breadth-first search (for unweighted graphs), Dijkstra's, and Bellman-Ford. Recall that we use n for the number of vertices and m for the number of edges in our graphs.

Objectives:

- Understand breadth-first search (BFS) and compare to DFS.
- Understand weighted graphs, Dijkstra's algorithm, and Bellman-Ford.
- Know under which conditions the above algorithms are correct and why.

1 Breadth-First Search

Recall that we studied depth-first search (DFS), a general algorithm with many uses; in particular, we used it to solve reachability and to construct topological sorts. Similarly, *breadth-first search (BFS)* is a general algorithm that solves many problems – including some of the same ones as DFS, but in a different way.

First, we will use BFS to solve the previous problem of reachability: can one follow edges from s and arrive at t? We'll need a data structure called a *queue.* A *queue* is just a list that supports the following operations, and we will ask for the following time complexities:

Operation	**Time complexity**	**Meaning**
append(v)	$O(1)$	add v to the end of the list
pop()	$O(1)$	remove the first item of the list and return it
size()	$O(1)$	return current size of list

This queue is *first-in first-out (FIFO)*, meaning that items are guaranteed to be *pop*pped in the order they are *append*ed.

Remark. To see that such a queue is possible in the word RAM model, note that we can use an array along with two pointers, one to the beginning and one to the end. To append v, increment the end-pointer and put v in the final slot; also add one to the current size. To pop from a nonempty list, just return the element at the beginning-pointer and increment the beginning-pointer; also decrement the current size. This isn't necessarily the most space-efficient implementation, though. One can also use linked lists, trees, or variants of the array approach.

The BFS approach to reachability is given in Algorithm 1. Notice that line 8 is quite abstract: it must look up the list of neighbors of u, iterate through them, and only execute the body of the loop if the neighbor v has `is_marked`$[v]$ = False. We will often describe algorithms at a high-level in this way, leaving the implementation to fill out these details.

Exercise 1. Sketch a proof that BFS-Reachability is correct. (The proof is very similar to that of DFS.)

Exercise 2. Show that the running time of BFS-Reachability is $O(n + m)$. (If stuck, recall the proof for DFS; this is easier.)

Algorithm 1 BFS-Reachability

```
1: Input: Directed graph $G = (V, E)$ as adjacency list, vertices $s, t$
2: Set is_marked[$u$] = False for all $u$
3: Create queue $Q$
4: $Q$.append($s$)
5: Set is_marked[$s$] = True
6: while $Q$ is not empty do
7:     Let $u = Q$.pop()
8:     for each unmarked out-neighbor $v$ of $u$ do
9:         $Q$.append($v$)
10:        Set is_marked[$v$] = True
11:    end for
12: end while
13: Return is_marked[$t$]
```

2 Shortest Paths on Unweighted Graphs

Soon, we will consider *weighted* graphs where edges have numbers, called weights, associated with them. But for now, let's stick with the *unweighted* directed graphs we've considered so far. (As before, it will be easy to adapt this algorithm to the undirected case.)

The **unweighted shortest-path problem** is:

- Input: A directed, unweighted graph G; two vertices s and t.
- Output: The length of the shortest path from s to t; and optionally, the path itself.

The length of the shortest path is also called the *distance* from s to t. If there is no path from s to t, we can indicate this by returning the symbol ∞ for the distance.

Layers. The idea is that BFS proceeds through the graph in *layers*. In the zeroth layer is s. In the first layer are all out-neighbors of s. These are all added to the queue while s is processed. In the second layer are all vertices that are out-neighbors of out-neighbors of s, that are not s itself nor a neighbor of s. These are all added to the queue while the first layer is being processed.

And so on. In the dth layer are all the vertices at distance d from s. This makes BFS for shortest paths easy, intuitively. One trick is that instead of using `is_marked`, we can just check if `dist`$[u] = \infty$ or not.

The only slightly tricky part is reconstructing the shortest path itself after we've processed all the vertices. We do this using an array `prev` to keep track, for each vertex on the path, of the previous vertex. As usual, we suppose the vertices are numbered $1, \ldots, n$.

Correctness - proof sketch. We prove correctness of the length by induction using the "layers idea". In particular, we show the algorithm processes all vertices at distance d before all vertices of higher distance; and that it sets the distances correctly.

Base case: `dist`$[s] = 0$ which is correct, and it is processed in the first iteration of the while loop, before all others. Inductive step: suppose we have just processed all vertices with distance at most d and set their distances correctly. This implies that the all remaining vertices have distance at least $d+1$. By definition, a vertex is at distance $d+1$ if and only if it is still remaining *and* is a neighbor of a distance d vertex. But we just iterated through all remaining neighbors of distance d vertices, so we added exactly the distance $d+1$ vertices to the queue and set their distances correctly.

It remains to show that the path itself is correct; this proceeds similarly. All distance 1 vertices have prev$[v] = s$, and all distance $d+1$ vertices have prev$[v] =$ a distance d vertex, and so on. So Reconstruct returns a path from s to t where the number of vertices (excluding s) is the distance to t, so it is a shortest path.

Algorithm 2 BFS-Path

```
1: Input: Unweighted directed graph $G = (V, E)$ as an adjacency list, vertices $s, t$
2: Set dist[$u$] $= \infty$ for all $u$
3: Set prev[$u$] $=$ NULL for all $u$
4: Create queue $Q$
5: Set dist[$s$] $= 0$
6: $Q$.append($s$)
7: while $Q$.size() $> 0$ do
8:     Let $u = Q$.pop()
9:     for each out-neighbor $v$ of $u$ do
10:        if dist[$v$] $= \infty$ then
11:            Set dist[$v$] =dist[$u$] $+ 1$
12:            Set prev[$v$] $= u$
13:            $Q$.append($v$)
14:        end if
15:    end for
16: end while
17: Run and return Reconstruct along with dist[$t$]
```

Subroutine 3 Reconstruct

```
1: Create list path
2: Let $v = t$
3: while prev[$v$] $\neq$ NULL do
4:     Add $v$ to beginning of path
5:     Set $v =$ prev[$v$]
6: end while
7: Return path
```

Running time - proof sketch. Just as with DFS, we can see that the inner `for` loop in line 9 executes in total at most $m = |E|$ times, and the outer `while` loop executes at most $n = |V|$ times, and each line is a constant-time operation. Reconstruct runs in $O(n)$ time, as it just iterates through a list of at most n vertices doing constant-time operations. So the overall running time is $O(n + m)$.

Exercise 3. What is the space usage of BFS-Path?

Exercise 4. (Optional) Can you utilize depth-first search to solve shortest paths on unweighted graphs? If so, give the algorithm and briefly argue correctness. If not, explain what goes wrong.

3 Positive Weighted Graphs and Dijkstra's

Now we suppose the input is a weighted graph with edge weights w_{uv} for each edge (u, v). The *length* of a path is now defined to be the sum of the weights of the edges in the path.

Note that we can easily modify the adjacency matrix and adjacency list representations in the word RAM model to allow for weights. For the adjacency list, the out-neighbor list of u now contains pairs (v, w_{uv}). For the adjacency matrix, the (u, v) entry of the matrix now contains w_{uv} if $(u, v) \in E$, or else a special marker, for example ∞, if $(u, v) \notin E$. We will assume that all numbers involved can be represented in the word size of the machine; for example, it will be enough if the sum of the absolute values of all input numbers is small enough to fit.

The **shortest-path problem** is:

- Input: A directed, weighted graph G; two vertices s and t.
- Output: The length of the shortest path from s to t; and optionally, the path itself.

Again the length of the shortest path is also called the *distance* from s to t.

For this section, we assume all edge weights are positive. Then BFS still feels close to the right algorithm. We want to expand out from s, processing vertices in order of distance. The problem is that once we've processed all vertices at distance $\leq d$, the next vertex could have any distance greater than d, not exactly $d + 1$. Also, it could be a neighbor of any previous vertex, not just the previous layer.

We'll need a new data structure: a *priority queue*. It needs to contain items along with a *value* for each item. It needs to support the following operations, when the maximum queue size is n items:

Operation	**Time complexity**	**Meaning**
create	$O(n \log n)$	create and insert n vertices with given values
update(v, d)	$O(\log n)$	update value for v to be d
popmin()	$O(\log n)$	remove the item with smallest value and return it
size()	$O(1)$	return number of items currently in the queue

The above time complexities are for the *binary tree* implementation, with which many students may be familiar. The Fibonacci heap, an advanced data structure, has a better asymptotic time complexity. But for concreteness we will just use the more-familiar binary tree implementation.

Correctness - proof sketch. Let $d(u)$ denote the true distance from s to u, i.e. length of the shortest path. There are several key facts we need.

Fact 1. For all $(u, v) \in E$, we have $d(v) \leq d(u) + w_{uv}$, because one path to v is to take the shortest path to u, then follow the edge (u, v).

Fact 2. Furthermore, if $v \neq s$, then $d(v) = d(u) + w_{uv}$ for some in-neighbor u. Here the shortest path to v is of the form $s, \ldots, u, v$, and the fact holds because $s, \ldots, u$ must be a shortest path to u. (Otherwise we could take a shorter path to u, and this would yield a shorter path to v, a contradiction.)

Fact 3. At all steps of the algorithm, `dist`$[v] \geq d(v)$ for all v. This is certainly true at the start of the algorithm, where `dist`$[s] = 0$ and `dist`$[u] = \infty$ for $u \neq s$. Then, we make some update of the form

Algorithm 4 Dijkstra's Algorithm

```
1: Input: Weighted directed graph $G = (V, E)$ with positive weights $w_{uv}$, vertices $s, t$
2: Set dist$[v] = \infty$ for all $v$
3: Set dist$[s] = 0$
4: Set prev$[v]$ = NULL for all $v$
5: Create priority queue $Q$ with all vertices and distances
6: while Q.size() $> 0$ do
7:     Let $u = Q$.popmin()
8:     for each neighbor $v$ of $u$ do
9:         Call UpdateDist$(u, v)$
10:        Q.update($v$, dist$[v]$)                    // if $v \notin Q$, do nothing
11:    end for
12: end while
13: Run and return Reconstruct along with dist$[t]$
```

Subroutine 5 UpdateDist(u, v)

```
1: if dist$[v] >$ dist$[u] + w_{uv}$ then
2:     Set dist$[v]$ = dist$[u] + w_{uv}$
3:     Set prev$[v] = u$
4: end if
```

`dist`$[v]$ =`dist`$[u] + w_{uv} \geq d(u) + w_{uv} \geq d(v)$. And so on: at each step, this invariant is maintained for all v.

Fact 4. At each step of the algorithm, `dist`$[v]$ can only decrease, never increase. This follows immediately from the UpdateDist subroutine. Note that along with Fact 3, this implies that once we set `dist`$[v] = d(v)$, we never change it again.

Now, we prove by induction that vertices are processed in order of distance from s and have `dist`$[u] = d(u)$ by the time they are processed. As just mentioned, this implies that `dist`$[u] = d(u)$ at the end of the algorithm for all vertices, since they are all eventually processed.

Base case: after the first iteration, we have processed only s, the closest vertex to itself, and we have correctly set its distance to zero.

Inductive step: Let T be the vertices remaining in the queue at the beginning of an iteration. Consider all paths starting at s that end at any vertex in T. We first claim the shortest such path contains only one vertex in T: otherwise, we could stop the path early at the first vertex in T, and this would shorten the path because all edge weights are positive.

Therefore, the shortest path from s to any not-yet-processed vertex is of the form $s, \dots, x, u$ where in particular x has already been processed. By Fact 2, we have $d(u) = d(x) + w_{xu}$. By induction hypothesis, when processing x, we had `dist`$[x] = d(x)$. Then we called UpdateDist(x, u) and this ensured that `dist`$[u] \leq$`dist`$[x] + w_{xu} = d(x) + w_{xu} = d(u)$. Since by Fact 3 we also have `dist`$[u] \geq d(u)$, we must have `dist`$[u] = d(u)$.

So at this time, `dist`$[u] = d(u) \leq d(y) \leq$`dist`$[y]$ for all other $y \in T$, using Fact 3. So the algorithm next processes a u that has smallest distance from s out of all remaining vertices, and `dist`$[u]$ is set correctly. (Note that the argument works correctly if there are ties; any u at smallest distance from s may be processed and the argument goes through.)

The above proves that the path length is correct. To show that the path returned by Reconstruct is correct, recall that when we updated the distance of v for the final time, we set `dist`$[v] = d(v)$ =`dist`$[u]+ w_{uv} = d(u)+w_{uv}$ where the shortest path to v is of the form $s, \dots, u, v$. At this point, we set `prev`$[v] = u$. This shows that for all vertices except s, `prev` points to a previous vertex along a shortest path to v. Since Reconstruct just follows these pointers from t back to s, it is correct.

Running time. We claim that Dijkstra's algorithm runs in time $O((n+m)\log n)$ when using a binary tree implementation of a priority queue discussed above. First, we must create the priority queue, which with a binary tree takes $O(n \log n)$ time. Again we go line-by-line. The outer while loop executes at most n times, so we execute line 7 n times. With a binary heap implementation, as discussed, this is $O(\log n)$ timer per execution, giving us a contribution of $O(n \log n)$. The for loop executes at most m times over the course of the algorithm, and inside it are all constant-time operations except Q.update(), which again with a binary heap is $O(\log n)$ per. This gives an $O(m \log n)$ contribution. Finally, Reconstruct takes $O(n)$ time (the loop executes n times and only performs constant-time operations). So we get a running time upper bound of $O\big((n+m)\log n\big)$.

Using a Fibonacci heap. Abstracting the analysis, we see that the running time of Dijkstra's algorithm is dominated by m update() operations and n popmin() operations of the priority queue. Using a Fibonacci heap, whose implementation we will unfortunately skip, the *total* time for the m update() operations is bounded by $O(m)$ and the *total* time for the n popmin() operations is bounded by $O(n \log n)$.

This allows Dijkstra's algorithm to run in time $O(m + n \log n)$.

Note that not every update() operation of the Fibonacci heap will run in $O(1)$ time and not every popmin() operation will run in $O(\log n)$ time. Instead it is only true that the totals are bounded by $O(m)$ and $O(n \log n)$ respectively. This is sometimes referred to as $O(1)$ and $O(\log n)$ "amortized" time per operation.

Exercise 5. What is an example graph with negative edge weights where Dijkstra's algorithm returns a wrong answer?

4 Possibly-negative edge weights

Now we allow some edge weights to be negative, but we disallow graphs with cycles whose total length is negative. Here Dijkstra's algorithm fails because it's not true that adding more steps to a path makes its length larger.

The key idea is the update step, where we set

$$\text{dist}[v] = \min\left\{\text{dist}[v]\ ,\ \text{dist}[u] + w_{uv}\right\}.$$

This is always a valid way to (weakly) improve the distance estimate to v, because it either keeps the curent path, or takes a path through u. So the idea of Bellman-Ford is to keep making these improvements for a while, and prove that after enough iterations the shortest paths have been found.

Algorithm 6 Bellman-Ford Algorithm

```
1: Input: Weighted directed graph $G = (V, E)$ with weights $w_{uv}$, vertices $s, t$
2: Set dist[$v$] = $\infty$ for all $v$
3: Set dist[$s$] = 0
4: Set prev[$v$] = NULL for all $v$
5: for $n - 1$ iterations do
6:     for each edge $(u, v)$ do
7:         Call UpdateDist($u, v$)
8:     end for
9: end for
10: Run and return Reconstruct along with dist[$t$]
```

Correctness. We assume no negative cycles. The idea is to prove by induction that after, iteration $j \in \{0, \ldots, n-1\}$ of the outer `for` loop, we have found and finalized all shortest paths that have at most j edges. Since there are no negative cycles, shortest paths have at most $n-1$ edges: any longer path must have a cycle, and removing the cycle from the path leads to the same destination with a shorter length. So this will prove that we've found all shortest paths from s.

Base case: before any iterations, we have the shortest path with zero edges, i.e. s, with its correct distance 0.

Inductive case: suppose we are in iteration $j + 1$. Suppose there exists such a shortest path from s to v with $j + 1$ edges, say $s, \dots, u, v$.

Even though edge weights can be negative, we still have from the proof of Dijkstra's a very important **Fact:** this path must begin with a shortest path from s to u of j edges. The proof is the same: If $s, \dots, u$ were not a shortest path, then we could use a shortest path to u to construct an even shorter path to v, a contradiction.

By inductive assumption, we have already found a shortest path to u, so in this iteration when we call UpdateDist(u, v), we must set `dist`$[v]$ equal to the distance from s to v.

Furthermore, as argued with Dijkstra's, we will never increase `dist`$[v]$ nor set it to a distance shorter than the shortest path, so it stays correct for the rest of the algorithm.

Efficiency. The running time is $O(n \cdot m)$ and space usage is $O(n)$; this is left as an exercise.

Benefits. A main benefit of Bellman-Ford is that, although the total number of operations in the running time is significantly higher than in Dijkstra's, it can be implemented in a distributed and robust way. Namely, consider a huge network of routers. In this example, they are all trying to help find the shortest path from s to t. Each router can periodically ask its neighbors for their current shortest distance, and then run UpdateDist() itself. It doesn't need a centralized controller coordinating the process. After enough time, all distances will be correct and we can run Reconstruct. We can also maintain these distances as the network changes or updates, propogating that information via distributed calls to UpdateDist().

Exercise 6. Show that Bellman-Ford's running time is $O(n \cdot m)$ and space usage is $O(n)$.