CS577: Intro to Algorithms — Shortest paths revisited

Recall the shortest paths problem:

Given: Weighted graph $G = (V, E)$ with cost function $c : E \to \mathbb{R}$, and a distinguished vertex $s \in V$.

Goal: Find the shortest path from node s to all other vertices.

Earlier in the course we discussed Dijkstra's algorithm for this problem. The algorithm can be implemented in $O(|V||E|)$ time using priority queues. Unfortunately the algorithm fails to give the correct answer if the graph contains any negative cost edges. (Negative costs might not seem natural in the context of computing distances, but shortest path problems come up in a variety of situations where negative costs may be present. We will discuss one such example in the context of network flow.)

Can we compute shortest path in graphs with negative cost edges? Dynamic programming gives an answer. Recall that in a dynamic program our main goal is to try to express the optimal solution in the form of a solution to a simpler subproblem. What could such a problem be in the context of shortest paths? Let us focus on the simpler case where we want to find the shortest path between s and t alone. One approach may be to guess the vertex on the shortest path immediately preceding t. Suppose this vertex is u. Then the problem reduces to finding the shortest path between s and u. However, this doesn't immediately seem simpler than the original problem itself. Can we attribute some property to the path between s and u that shows that we have made some progress towards making the problem simpler? Here are two ways:

1. The path between s and u is shorter (in terms of the number of "hops" or edges in the path) than the one between s and t.
2. The path between s and u visits a smaller set of vertices than the one between s and t.

Following the first approach leads to the Bellman-Ford algorithm, while the second leads to the Floyd-Warshall algorithm. We will see that both approaches have unique strengths. We will discuss the single-source multiple-target version of the problem first and then extend it to all-pairs shortest paths. As an aside, Richard Bellman is the inventor of dynamic programming.

We first note an important assumption—in order for the two algorithms to work G must not contain cycles of negative weight. If it does, a shortest path is not well-defined as one can reduce the distance between any two nodes to negative infinity by going to the cycle, traversing it an infinite number of times and then going to the destination. One may ask, in this context, for the shortest path that visits each node at most once. The two algorithms mentioned above cannot find such paths. In fact no polynomial time algorithms are known for this problem. We will discuss this further towards the end of the course. Towards the end of this section we discuss the matter of negative cycles further.

We will start with the Bellman-Ford algorithm and discuss both the single-source and all-pairs variants; the Floyd-Warshall algorithm is slower in the single-source case but faster in the all-pairs case, and will be discussed later.

1.1 Bellman-Ford

The key observation for the first approach comes as a result of noticing that any shortest path in G will have at most $|V|-1$ edges. Thus if we can answer the question "what's the shortest path from s to each other vertex t that uses at most $|V|-1$ edges," we will have solved the problem. We can answer this question if we know the neighbors of t and the shortest path to each node that uses at most $|V|-2$ edges, and so on.

Computing the shortest path from s to t with at most 1 edge is easy: if there is an edge from s to t, that's the shortest path; otherwise, there isn't one. (In the recurrence below, such weights are recorded as ∞.)

Computing the shortest path from s to t with at most 2 edges is not much harder. If u is a neighbor of both s and t, there is a path from s to t with two edges; all we have to do is take the minimum sum. By convention, we will assume that missing edges have weight ∞. Then we need not restrict the minimum to vertices u that are neighbors of s and can just take the minimum over all nodes adjacent to t. (Or even over all of V.) We must also consider the possibility that the shortest path is still just of length one.

A similar process is repeated for 3 edges; we look at each neighbor u of t, add the weight of going from s to u with at most 2 hops to the weight of going from u to t, then taking the minimum such weight. Again, we must consider the possibility that the best path doesn't change when adding the 3rd edge.

Formally, we define $A[i][k]$ to be the length of the shortest path from the source node to node i that uses at most k edges, or ∞ if such a path does not exist. Then the following recurrence will compute the weights of the shortest paths, where $N(i)$ is the set of nodes adjacent to i and min returns ∞ if $N(i)$ is empty:

$$A[i][k] = \min \begin{cases} \min_{j \in N(i)} \left(A[j][k-1] + c(j,i)\right) \\ A[i][k-1] \end{cases}$$

The first option corresponds to the case where the path described by $A[i][k]$ actually involves all k edges; the second option is if allowing a kth edge doesn't change the solution. The second option can be removed if we consider an implicit, zero-weight loop at each node (that is, $c(v,v) = 0$ for all v).

The bottom-up dynamic programming approach to this problem uses an $|V| \times |V|$ matrix.

$$\begin{array}{cc} & \begin{array}{ccc} 1 & \dots & |V| \end{array} \\ \begin{array}{c} |V| \\ \vdots \\ \vdots \\ 2 \\ 1 \end{array} & \begin{bmatrix} & & \\ & \square & \\ \swarrow & \downarrow & \searrow \\ & & \\ & & \end{bmatrix} \end{array}$$

Rows represent the maximum number of edges that are allowed for a given instance; the rows are ordered so that the final answer (with a maximum of $|V|-1$ edges) is at the top of the graph. Columns represent vertices in the graph.

When computing $A[i][k]$, represented by the boxed square, the algorithm looks at all of the boxes in the next row down for nodes that are adjacent to i, as well as i itself. Therefore each square takes $O(|V|)$ time. Since there are $|V|^2$ squares, this immediately gives us an upper bound of $O(|V|^3)$ for the running time. However, we can do better.

Consider a traversal of one entire row in the above matrix. During this pass, each edge (u, v) will be considered twice: once from node u to node v, and once from v to u. (To see this, note that the diagonal arrows in the schematic above correspond to edges.) Thus the work done in each *row* is $O(|E|)$. There are $|V|$ rows, so this gives a total bound of $O(|V| \cdot |E|)$ for the algorithm.

This presentation of these shortest-path algorithms only explains how to determine the weight of the shortest path, not the path itself, but it is easy to adapt the algorithm to find the path as well. This is left as an exercise for the reader.

All Pairs Shortest Paths

The all pairs shortest path problem constitutes a natural extension of the single source shortest path problem. Unsurprisingly, the only change required is that rather than fixing a particular start node and looking for shortest paths to all possible end nodes, we instead look for shortest paths between all possible pairs of a start node and end node. Formally, we may define the problem as follows:

Given: A graph $G = (V, E)$. A cost function $c : E \to \mathbb{R}$.

Goal: For every possible pair of nodes $s, t \in V$, find the shortest path from s to t.

Just as the problem itself can be phrased as a simple extension of the single source shortest paths problem, we may construct a solution to it with a simple extension to the Bellman-Ford algorithm we used for that problem. Specifically, we need to add another dimension (corresponding to the start node) to our table. This will change the definition of our recurrence to

$$A[i, j, k] = \min \begin{cases} A[i, j, k-1] \\ \min\limits_{l} \{A[i, l, k-1] + c(l, j)\} \end{cases} .$$

If we take $c(l, l) = 0$ for all $l \in V$, we may write this even more simply as

$$A[i, j, k] = \min_{l} \{A[i, l, k-1] + c(l, j)\}.$$

It is easy to see how this modification will impact the running time of our algorithm — since we have added a dimension of size $|V|$ to our table, our running time will increase from being $O(|V| \cdot |E|)$ to being $O(|V|^2 \cdot |E|)$.

Can we do any better? Let us consider the "levels" of this three dimensional array indexed by k. To compute the values of the $|V|^2$ cells in level k we use values at level $k - 1$ (corresponding to paths of length $k - 1$) and values at level 1 (corresponding to paths of length 1) and sum them up together. Another equivalent way of finding values at level k is to find values at level $k/2$ and then compose two paths of length $k/2$ to find paths of length k. Assuming that $|V| - 1$ is a power of 2 (and so the values of k we encounter will all be powers of 2), our recurrence relation becomes

$$A[i, j, k] = \min_{l} \{A[i, l, k/2] + A[l, j, k/2]\}, \text{ where } A[i, j, 1] = c(i, j).$$

The running time for this version of the algorithm may be seen to be $O(|V|^3 \log |V|)$. In each iteration, in order to compute each of the $|V|^2$ elements in our array, we need to check each possible midpoint, which leads to each iteration taking $O(|V|^3)$ time. Since we double the value of k at each step, we need only $\lceil \log |V| \rceil$ iterations to reach our final array. Thus, it is clear that the overall running time must be $O(|V|^3 \log |V|)$, a definite improvement over our previous running time of $O(|V|^2 \cdot |E|)$ when run upon a dense graph.

Negative Cycles

Throughout the preceding discussion on how to solve the shortest path problem — whether the single source variant or the all pairs variant — we have assumed that the graph we are given will not include any negative cost cycles. This assumption is critical to the problem itself, since the notion of a shortest path becomes ill-defined when a negative cost cycle is introduced into the graph. After all, once we have a negative cost cycle in the graph, we may better any proposed minimum cost for traveling between a given start node and end node simply by including enough trips around the negative cost cycle in the path we choose. While the shortest path problem is only well-defined in the absence of negative cost cycles, it would still be desirable for our algorithm to remain robust in the face of such issues.

In fact, the Bellman-Ford algorithm does behave predictably when given a graph containing one or more negative cycles as input. Once we have run the algorithm for long enough to determine the the best cost for all paths of length less than or equal $|V| - 1$, running it for one more iteration will tell us whether or not the graph has a negative cost cycle: our table will change if and only if a negative cost cycle is present in our input graph. In fact, we only need to look at the costs for a particular source to determine whether or not we have a negative cycle, and hence may make use of this test in the single source version of the algorithm as well. If we wish to find such a cycle after determining that one exists, we need only consider any pair of a start node and an end node which saw a decrease in minimum cost when paths of length $|V|$ were allowed. If we look at the first duplicate appearance of a node, the portion of the path from its first occurrence to its second occurrence will constitute a negative cost cycle.

1.2 Floyd-Warshall algorithm for shortest paths

A second way of decomposing a shortest path problem into subproblems is to reduce along the intermediate nodes used in the path. In order to do so, we first augment the graph we are given by labeling its nodes from 1 to n.

When we reduce the shortest paths problem in this fashion, we end up with the table

$$A[i, j, k] = \text{ shortest path from } i \text{ to } j \text{ using only intermediate nodes from } 1, \ldots, k,$$

which is governed by the recurrence relation

$$A[i, j, k] = \min \begin{cases} A[i, j, k-1] \\ A[i, k, k-1] + A[k, j, k-1] \end{cases}, \text{ where } A[i, j, 0] = c(i, j).$$

It is reasonably straightforward to see that this will indeed yield a correct solution to the all pairs shortest path problem. Assuming once again that our graph is free of negative cost cycles, we can see that a minimum path using the first k nodes will use node k either 0 or 1 times; these cases correspond directly to the elements we take the minimum over in our definition of $A[i, j, k]$. An inductive proof of the correctness of the Floyd-Warshall algorithm follows directly from this observation.

If we wish to know the running time of this algorithm, we may once again consider the size of the table used, and the time required to compute each entry in it. In this case, we need a table that is of size $|V|^3$, and each entry in the table can be computed in $O(1)$ time; hence, we can see that the Floyd-Warshall algorithm will require $O(|V|^3)$ time in total.

Although this algorithm gives a better bound on the running time in the all-pairs case, it fails to scale down nicely to the single-source version of the problem, and the Bellman-Ford algorithm provides the better bound for that version. The Floyd-Warshall algorithm can also be used to detect negative cycles – namely by determining whether $A[i, i, n]$ is negative for any node i. Moreover, it can be implement in quadratic space by reusing space from previous iterations.