CS 3510 Algorithms 2/06/2024

Lecture 8: Bellman-Ford and Floyd-Warshall

Lecturer: Abrahim Ladha *Scribe(s): Richard Zhang*

1 Single Source Shortest Path (with Negative Edges)

Previously, we were dealing with finding the shortest path from a single source for a graph with nonnegative edge weights. We used Dijkstra's algorithm for this, which worked because the paths to any node v would go through nodes closer to the source than v. This allows the algorithm to mark the node on the top of the priority queue as visited and not consider any more paths to this node. However, with negative edges, this useful property is broken! Consider the following graph:

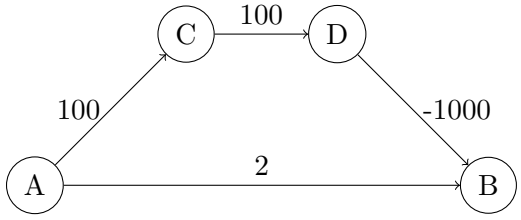

Figure 1: Graph with negative edge weights.

Dijkstra's will pop node B off the priority queue and mark its shortest distance to be 2 since the path from A to C seems to be much larger. However, the path with the least total weight would actually be $A \to C \to D \to B$ with a total weight of -800 (which is much lower than 2). Dijkstra's has not accounted for shorter paths found in the future.

Another issue arises with negative edges. Consider the graph below:

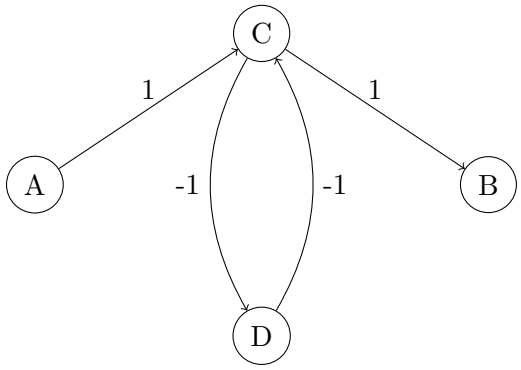

Figure 2: Graph with negative cycle

What would be the shortest path from A to B? The path from A to C to B with length 2 can be reduced to 0 by going through node D. However, we can go through node D multiple times and further decrease the path length possibly to $-\infty$! C and D in this graph form a **negative cycle** since the sum of edges in the cycle is negative. There can be positive edge weights in a negative cycle, but the total sum of all the cycle edges is negative. Negative cycles make the shortest path problem ill-defined because of this, so we need a way to somehow detect this when designing a shortest path algorithm that accounts for negative edges.

1.1 Bellman-Ford

The Bellman-Ford algorithm can help us with finding the shortest path from a single source with negative edge weights. The Bellman-Ford algorithm utilizes the same sort of distance update operation/relaxation that Dijkstra's uses:

$$dist(v) := min(dist(v), dist(u) + l(u, v))$$

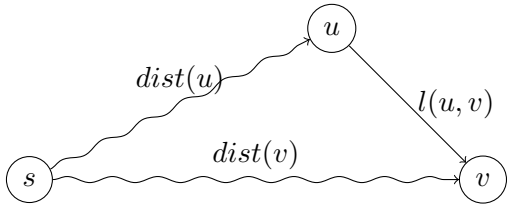

Figure 3: Distance update visualization

Dijkstra has the privilege to stop performing this update operation for a node v when it is popped from the priority queue since a shorter path would to v would never happen in the future. However, with negative edges, Bellman-Ford may need to apply this update operation may more times to account for future shorter paths. By applying this update operation multiple times for all the vertices, Bellman-Ford can find the shortest paths (although without Dijkstra's shortcuts).

How does Bellman-Ford know when to stop updating the distance array? Let's first assume that there are no negative cycles. If we were to apply this update operation once for all the edges in the graph, the nodes that are one edge away from the source are guaranteed to be updated. Apply another round of this, and the nodes one or two edges away are guaranteed to be updated. Essentially, the ith round would have found the shortest paths with at most i edges. We can keep this going until we reach the maximum number of edges that a shortest path in a graph can have, which is $|V| - 1$ (visiting all $|V|$ vertices). This is because a path with more edges would visit a vertex more than once, an indication of an unnecessary cycle that only increases the path length.

With these insights, we can implement the algorithm as follows:

```
def bellmanFord(G, l, s):
    for all v in V:
        dist(u) = infinity

    dist(s) = 0
    for i=1...(|V| - 1)
        for each e=(u,v) in E
            dist(v) = min{dist(v), dist(u) + l(u, v)}
```

Figure 4: Bellman-Ford pseudo-code

There is no longer a priority queue, and this algorithm applies the same distance update over and over again until the maximum possible number of updates are reached. The runtime would be $O(|V||E|)$ since we looping through all the edges $|V|-1$ times. This is significantly slower than Dijsktra's runtime of $O((|V| + |E|) \log V)$ with a binary heap.

1.2 Example

Let's run Bellman-Ford on the following graph:

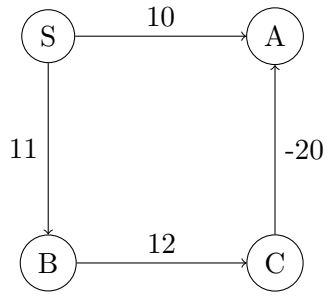

Figure 5: Bellman-Ford example graph

The current distance values for each round is presented in the table below:

Round	dist(S)	dist(A)	dist(B)	dist(C)
0	0	∞	∞	∞
1	0	10	11	∞
2	0	10	11	23
3	0	3	11	23

Figure 6: Bellman-Ford example distance values

There are 3 rounds since $|V| - 1 = 3$. At round 0, only the starting node S will have a distance value with the rest being infinity. As the rounds progress, notice how the distance values are each of the vertices are decreasing. Round 1 includes the shortest path distances with at most one edge, so the nodes A and B would have their distance values updated.

Round 2 includes shortest path distances with at most two edges, so C would have its distance value updated. The key observation here is the update on dist(A) in Round 3:

$$dist(A) := min(dist(A), dist(C) + l(C, A))$$

This would evaluate to $min(10, 23 - 20) = 3$, demonstrating how the distance value for a vertex can change multiple times (unlike Dijsktra's). Since Round 3 considers the shortest paths with at most 3 edges, the path $S \to B \to C \to A$ is accounted for.

1.3 Negative Cycle Detection

Bellman-Ford can also be used to detect if a negative cycle exists in a graph. With one more round, the shortest paths with $|V|$ edges would be considered. However, this would mean that a vertex must have been visited at least twice in this shortest path if it used $|V|$ edges. This means that there is a cycle that starts and ends at this vertex. If this cycle was a nonnegative one, then the shortest path value would not change since the cycle would only increase the total weight or keep it the same. However, if the cycle was negative, the shortest path distance would decrease.

With these insights, we can do the following to detect a negative cycle:

1. Run one more round of Bellman-Ford.
2. If a distance value has decreased for some vertex, then there must be a negative cycle.

2 Shortest Paths in Directed Acyclic Graphs

Without the presence of cycles, finding the shortest path in a directed acyclic graph (DAG) becomes much easier. Even with negative edges, the shortest path for a node is entirely based on the shortest paths from its incoming nodes. With the shortest paths calculated and fixed for the incoming nodes, the shortest path to the current node would be the minimum of the shortest paths of the incoming nodes plus their respective incoming edges. With the guaranteed topological order from the Top-Sort algorithm, we can compute the shortest paths from the bottom-up with the same update operation used in Dijkstra's and Bellman-Ford. Here is the algorithm:

```
def shortestPathInDag(G, l, s):
    for all v in V:
        dist(u) = infinity
    dist(s) = 0
    Topsort G
    for u in V in topsort order
        for edge (u,v) in E:
            dist(v) = min(dist(v), dist(u) + l(u,v))
```

Figure 7: Shortest Path in DAG pseudo-code

Let's run this algorithm on the following DAG:

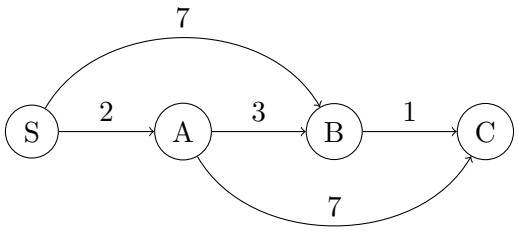

Figure 8: Shortest Path in DAG example graph

For each current node in the outer loop, here are the following current distance values after the relaxation operations are applied in the inner loop:

Current Node	dist(S)	dist(A)	dist(B)	dist(C)
Before outer loop	0	∞	∞	∞
S	0	2	7	∞
A	0	2	5	9
B	0	2	5	6
C	0	2	5	6

Figure 9: Shortest Path in DAG example values

Going in topological order ensures that all the incoming vertices of the current vertex were already processed. All the incoming edges to the vertex will be considered before the vertex is reached in the outer loop, which ensures that all possible shortest paths are considered and no possible path shorter is missed.

The runtime of this algorithm would be $O(|V| + |E|)$ since only the outgoing edges of each current vertex are processed once for the relaxation operation. This runtime is much more efficient than Dijkstra's.

3 All Pairs Shortest Path

Unlike the Single Source Shortest Path problem, the All Pairs Shortest Path problem looks for the shortest paths for all possible pairs of vertices. Instead of only considering one vertex to be the starting vertex, this problem now has every vertex in the graph be a possible start.

One native solution is to run Dijsktra's for each vertex in the graph and concatenate the distance arrays together. This would result in a runtime of $O(|V| \cdot (|V| + |E|) \log |V|) = O((|V|^2 + |V||E|) \log |V|)$, which does not seem terrible. However, with a relatively dense graph, where $|E| \sim |V|^2$, the runtime would be $O(|V|^3 \log |V|)$, which does not look as good. Of course, an algorithm for this problem would need to be $\Omega(|V|^2)$ since there are $|V|^2$ possible pairs that need to be iterated on.

3.1 Floyd-Warshall

We can achieve a faster runtime than running Dijkstra's $|V|$ times with the Floyd Warshall algorithm, which has a runtime of $O(|V|^3)$. Floyd-Warshall works by defining the following subproblem:

$$dist(i, j, k) = \text{ shortest path from } v_i \text{ to } v_j \text{ only considering } v_1 ... v_k$$

The nodes being considered are the possible vertices that could be in between v_i and v_j in the shortest path from v_i to v_j. Floyd-Warshall computes the shortest paths efficiently by reusing the shortest paths of possibly other pairs with the consideration of a smaller set of vertices. The algorithm inductively builds up the larger subproblems from smaller subproblems until all vertices are considered. We can store the answer to these smaller subproblems in an array for fast lookups.

One base case would be where we are considering the shortest paths from any vertex v_i to itself without considering any vertices in between. In this case, the shortest path distance for these cases would be 0, so $dist(i, i, 0) = 0$ for $v_i \in V$. Another base case is where two vertices are connected by an edge. Without any intermediate vertices being considered, the shortest path would have the distance value be the edge weight itself, so $dist(i, j, 0) = l(v_i, v_j)$ for all $(v_i, v_j) \in E$

Suppose by the induction hypothesis that we have computed all the answers for $dist(i, j, k-1)$ considering the vertices $v_1 ... v_{k-1}$. Now let's also consider the node v_k.

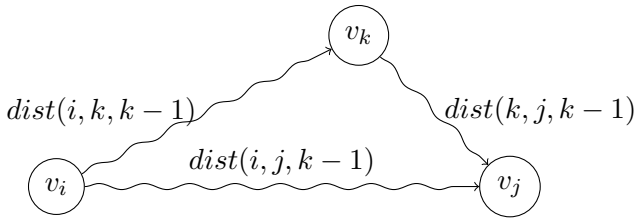

Figure 10: Inductive case in Floyd-Warshall

There are two cases. One case is where the shortest path still does not include v_k, so the shortest path would be what it was before v_k was being considered ($dist(i, j, k-1)$). The other case is where v_k is in the shortest path between v_i and v_j when now considering v_k. The shortest path from v_i to v_k can be appended with the shortest path from v_k to v_j, resulting the shortest path from v_i to v_j now considering vertices $v_1 ... v_k$. We can simply take the minimum of these two cases to get the answer to the larger subproblem.

With these ideas, here is the pseudo code:

```
def floydWarshall(G, l):
    # Base cases
    for i in range(|V|):
        dist(i, i) = 0
    for i in range(|V|):
        for j in range(|V|):
            dist(i, j) = l(i, j) if defined, infinity otherwise

    # Inductive cases
    for k in range(|V|)
        for i in range(|V|)
            for j in range(|V|)
                dist(i,j) = min(dist(i,j), dist(i,k) + dist(k, j))
```

Figure 11: Floyd-Warshall pseudo-code

Notice how a 2D array is being used in the implementation even though our *dist* subproblem has three inputs. This is because only the *dist* values from the previous round (where $v_1 ... v_{k-1}$ are being considered) were needed, so there is no need to store all the distance values before $k-1$.

The runtime for this algorithm would be $O(|V|^3)$ due to the triple nested for loop.

3.2 Example

Let's run Floyd-Warshall on the graph below:

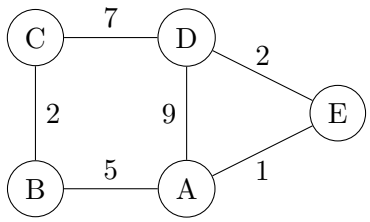

Figure 12: Floyd-Warshall example graph

Below are the distance array values when initialized with the base case values:

dist	A	B	C	D	E
A	0	5	∞	9	1
B	5	0	2	∞	∞
C	∞	2	0	7	∞
D	9	∞	7	0	2
E	1	∞	∞	2	0

Figure 13: Floyd-Warshall base case distance values

When A is being considered, we now consider shortest paths that have the node A in between. The distance array values become updated to the following:

dist	A	B	C	D	E
A	0	5	∞	9	1
B	5	0	2	14	6
C	∞	2	0	7	∞
D	9	14	7	0	2
E	1	6	∞	2	0

Figure 14: Floyd-Warshall first round distance values

When B is also being considered, we now consider shortest paths that have node A and/or node B in between (i.e. E to A to B to C). The distance array values become the following:

dist	A	B	C	D	E
A	0	5	7	9	1
B	5	0	2	14	6
C	7	2	0	7	8
D	9	14	7	0	2
E	1	6	8	2	0

Figure 15: Floyd-Warshall second round distance values

The distance table will continue to update until all the vertices have been considered, resulting in the shortest path distances between all pairs of nodes.

4 Dynamic Programming Sneak Peek

Bellman-Ford and Floyd-Warshall employ a technique called **dynamic programming**. Dynamic programming breaks the problems down into smaller, overlapping subproblems. The answers to these subproblems are cached in some data structure and are used to solve bigger problems. You will learn more about dynamic programming in the next unit, so stay tuned!