1 Fixing Dijkstra’s Algorithm with Negative Weights

Dijkstra’s algorithm doesn’t work on graphs with negative edge weights. Here is one attempt to fix it:

1. Add a large number $M$ to every edge so that there are no negative weights left.
2. Run Dijkstra’s to find the shortest path in the new graph.
3. Return the path found by Dijkstra’s, but with the old edge weights (i.e. subtract $M$ from the weight of each edge).

Show that this algorithm doesn’t work by finding a graph for which it must give the wrong answer. 

**Solution:** The above algorithm doesn’t work when the actual shortest path has more edges than other potential shortest paths. In this case, the paths with more edges have their weights increased more than the paths with fewer edges. We can see this in the following counterexample:

The shortest path is “down-right-up” (weight $-7$). After adding $M = 5$ to each edge, we increase the actual shortest path by 15. The path “right” only increases by 5 and so the algorithm returns this path as the shortest path.

2 Bellman-Ford Practice

(a) Run the Bellman-Ford algorithm on the following graph, from source $A$. Relax edges $(u, v)$ in lexicographic order, sorting first by $u$ then by $v$. 

---

Note: Your TA may not get to all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. The discussion worksheet is also a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.
(b) What problem occurs when we change the weight of edge $(H, A)$ to 1? How can we detect this problem when running Bellman-Ford? Why does this work?

(c) Let $G = (V, E)$ be a directed graph. Under what condition does the Bellman-Ford algorithm return the same shortest path tree (from source $s \in V$) regardless of the ordering on edges?

**Solution:**

(a) The resulting shortest path tree (dashed edges are non-tree edges):

Remember that you can terminate the Bellman-Ford algorithm as soon as a relaxation step does not change any distances.
(b) Changing the weight of \((H, A)\) to 1 introduces a negative cycle. We can detect this by checking whether, after making \(|V| - 1 = 9\) passes over the edges (relaxation steps), we can still update the distances. Try it out! The reason it works is that after \(k\) passes, we have found the length of all shortest paths from \(s\) with at most \(k\) edges. Since a simple path can only have at most \(|V| - 1\) edges, if we can update the distances on the \(|V|\)-th pass, the new shortest path contains a cycle.

A cycle can only be part of a shortest path if it is of negative weight.

(c) If and only if the shortest path tree rooted at \(s\) is unique, i.e. if for each \(v \in V\) there exists a unique shortest path from \(s\) to \(v\). Note that this is not the case in the example: there are two paths of length \(-1\) from \(A\) to \(D\).

3 Service scheduling

A server has \(n\) customers waiting to be served. Customer \(i\) requires \(t_i\) minutes to be served. If, for example, the customers were served in the order \(t_1, t_2, t_3, \ldots, t_n\), then the \(i\)-th customer would wait for \(t_1 + t_2 + \cdots + t_i\) minutes.

We want to minimize the total waiting time

\[
T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i).
\]

Given the list of the \(t_i\)'s, give an efficient algorithm for computing the optimal order in which to serve the customers.

**Solution:** We use a greedy strategy, by sorting the customers in increasing order of service times and serving them in this order. The running time is \(O(n \log n)\).

To prove correctness, for any ordering of the customers, let \(s(j)\) denote the \(j\)-th customer in the ordering. Then

\[
T = \sum_{i=1}^{n} \sum_{j=1}^{i-1} t_{s(j)} = \sum_{i=1}^{n} (n - i)t_{s(i)}.
\]

For any ordering, if \(t_{s(i)} > t_{s(j)}\) for \(i < j\), then swapping the positions of the two customers gives a better ordering. Since we can generate all possible orderings by swaps, an ordering which has the property that \(t_{s(1)} \leq \ldots \leq t_{s(n)}\) must be the global optimum. This is exactly the ordering that we output.

4 MST Basics

For each of the following statements, either prove or give a counterexample. Always assume \(G = (V, E)\) is undirected and connected. Do not assume the edge weights are distinct unless specifically stated.

(a) Let \(e\) be any edge of minimum weight in \(G\). Then \(e\) must be part of some MST.

(b) If \(e\) is part of some MST of \(G\), then it must be a lightest edge across some cut of \(G\).

(c) If \(G\) has a cycle with a unique lightest edge \(e\), then \(e\) must be part of every MST.

(d) For any \(r > 0\), define an \(r\)-path to be a path whose edges all have weight less than \(r\). If \(G\) contains an \(r\)-path from \(s\) to \(t\), then every MST of \(G\) must also contain an \(r\)-path from \(s\) to \(t\).

**Solution:**

1. True, \(e\) will belong to the MST produced by Kruskal.
2. True, suppose \((u, v)\) is the edge. Let one side of the cut be everything reachable in the MST from \(u\) without using the edge \((u, v)\). If this cut has an edge lighter than \((u, v)\) then we could add this edge to the MST and remove \((u, v)\). We know this edge is not already in the MST because otherwise both its endpoints would be reachable from \(u\) without using \((u, v)\).

3. False. Let \(e\) be also the heaviest edge of a different cycle; then, we know that \(e\) can’t be part of the MST. Concretely, in the following graph, edge \((B, E)\) satisfies this condition, but will not be added to the graph.

![Graph](image)

4. True. Let \(v_1, v_2, \ldots, v_n\) denote the \(r\)-path in \(G\) from \(s = v_1\) to \(t = v_n\). Consider the greatest \(i\) such that the MST has an \(r\)-path from \(s\) to \(v_i\) and assume for contradiction that \(i < n\). Then the edge \((v_i, v_{i+1})\) is not in the MST. Adding this edge to the MST forms a cycle, which must have edges of length less than \(r\) since otherwise we could replace one of them with \((v_i, v_{i+1})\). Thus there is an \(r\)-path from \(v_i\) to \(v_{i+1}\) and hence from \(s\) to \(v_{i+1}\), contradicting our initial assumption.

5 Updating a MST

You are given a graph \(G = (V, E)\) with positive edge weights, and a minimum spanning tree \(T = (V, E')\) with respect to these weights; you may assume \(G\) and \(T\) are given as adjacency lists. Now suppose the weight of a particular edge \(e \in E\) is modified from \(w(e)\) to a new value \(\hat{w}(e)\). You wish to quickly update the minimum spanning tree \(T\) to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each, give a description of an algorithm for updating \(T\), a proof of correctness, and a runtime analysis for the algorithm. Note that for some of the cases these may be quite brief.

(a) \(e \notin E'\) and \(\hat{w}(e) > w(e)\)
(b) \(e \notin E'\) and \(\hat{w}(e) < w(e)\)
(c) \(e \in E'\) and \(\hat{w}(e) < w(e)\)
(d) \(e \in E'\) and \(\hat{w}(e) > w(e)\)

Solution:

(a) Main Idea: Do nothing.

Correctness: \(T\)’s weight does not increase, and any other spanning tree’s weight either stays the same or increases, so \(T\) must still be an MST.

Runtime: Doing nothing takes \(O(1)\) time.
(b) **Main Idea:** Add $e$ to $T$. Use DFS to find the cycle that now exists in $T$. Remove the heaviest edge in the cycle from $T$.

**Correctness:** The heaviest edge in a cycle is safe to exclude from the MST (because if it is in the MST, you can remove it from the MST and add some other edge to the MST, and the MST’s cost will not increase), and any edge not in an MST is the heaviest edge in some cycle (in particular, the cycle formed by adding it to the MST). For any edge not in $T$ except for $e$, decreasing $e$’s weight does not change that it is the heaviest edge in the cycle, so it is safe to exclude from the MST. By adding $e$ to $T$ and then removing the heaviest edge in the cycle in $T$, we remove an edge that is also safe to exclude from the MST. Thus after this update, all edges outside of $T$ are safe to exclude from the MST.

**Runtime:** This takes $O(|V|)$ time since $T$ has $|V|$ edges after adding $e$, so the DFS runs in $O(|V|)$ time.

(c) **Main Idea:** Do nothing.

**Correctness:** $T$’s weight decreases by $w(e) - \hat{w}(e)$, and any other spanning tree’s weight either stays the same or also decreases by this much, so $T$ must still be an MST.

**Runtime:** Doing nothing takes $O(1)$ time.

(d) **Main Idea:** Delete $e$ from $T$. Now $T$ has two components, $A$ and $B$. Find the lightest edge with one endpoint in each of $A$ and $B$, and add this edge to $T$.

**Correctness:** Every edge besides $e$ in the MST is a lightest edge in some cut prior to changing $e$’s weight, and increasing $e$’s weight cannot affect this property. So all edges besides $e$ are safe to keep in the MST. Then, whatever edge we add is also the lightest edge in the cut $(A, B)$ and is thus also safe to include in the MST.

**Runtime:** This takes $O(|V| + |E|)$ time, since it might be the case that almost all edges in the graph might have one endpoint in both $A$ and $B$ and thus almost all edges will be looked at.