This homework is due on gradescope Friday October 26th at 11:59pm. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommend though not required.

Question 1 (Partial Dijkstra, 30 points). Suppose that when running Dijkstra’s algorithm we only need to be able to find the vertices within distance \(d\) of the source node.

(a) Show that you can modify Dijkstra to achieve this in \(O(|V_d| \log |V_d| + |E_d|)\) time where \(V_d\) is the set of vertices within distance \(d\) of the source and \(E_d\) is the collection of edges with at least one endpoint in \(V_d\).

Note: This is practically very useful, as it means that you don’t need to analyze the entire map of North America in order to find the shortest path to your local grocery store. \([15\text{ points}]\)

(b) Suppose that furthermore you know that all edge weights in the graph are non-negative integers. Show that the runtime can be improved to \(O(|V_d| + |E_d| + d)\). \([\text{Hint: you will want to modify the priority queue being used in the algorithm using the knowledge that all keys will be integers at most }d]\) \([15\text{ points}]\)

Solution 1. \((a)\)

**Algorithm.** We can modify Dijkstra to use a Fibonacci heap which will give us a runtime bound of \(O(|V| \log |V| + |E|)\).

To achieve the desired bounds of \(O(|V_d| \log |V_d| + |E_d|)\), we can make a further modification by building the priority queue as we discover new nodes, adding nodes to the priority queue if and only if they are within distance \(d\) of \(s\). Once the priority queue is empty, we return the closed set of vertices discovered by our modified Dijkstra.

**Complexity.** We can see that the priority queue will only contain at most \(|V_d|\) entries, giving us \(O(|V_d| \log |V_d|)\) for priority queue updates when moving items from the fringe to the closed set.

Since the algorithm only considers \(V_d \subseteq V\), we see that we only explore \(E_d\) edges, which will give us \(O(|E_d|)\) runtime for updating values in the priority queue.

Altogether this implies the runtime of our algorithm is \(O(|V_d| \log |V_d|) + O(|E_d|) = O(|V_d| \log |V_d| + |E_d|)\).

**Correctness.** When a vertex \(w\) is within distance \(d\) of \(s\), Dijkstra’s algorithm guarantees us that it will remain within that distance or improve.

When a vertex \(w\) is greater than distance \(d\) from \(s\), for some vertex \(v\) with \(e = (v, w)\), \(w\) is effectively unreachable from \(v\) and is akin to a modified \(G = G'\) where \(E' = E - e\). Because the path from \(s\) to \(v\) was minimal, Dijkstra’s algorithm also guarantees us that \(w\) will remain unreachable from \(v\), implying that we never need to revisit \(e\).

So we see that our algorithm explores a vertex if and only if that vertex is within distance \(d\) from \(s\).

(b)

**Algorithm.** We can modify Dijkstra by initiating a priority queue consisting of \(d+1\) buckets represented by dynamic arrays and ordered by priority (i.e. \(0, 1, \ldots, d\), with bucket \(k\) corresponding to priority \(k\)). The other modification is, as above, we add vertices to the queue if and only if they are within distance \(d\) of \(s\). Once the priority queue is empty, we return the closed set of vertices discovered by our modified Dijkstra.
Complexity. Building \(d\) buckets takes \(O(d + 1) = O(d)\) time.

If each bucket contains lists of occupied and unoccupied indices, and we have a pointer to the next non-empty bucket, every iteration of the algorithm can then pull the next element with the highest priority in \(O(1)\) time.

If we keep a table containing bucket indices and indices within buckets for all vertexes in the queue, we can remove an item from the queue and add it to the closed set in \(O(1)\) time. This implies the algorithm takes \(O(|V_d|)\) time for this (since we must do it for each vertex in \(V_d\)).

As we have a list of unoccupied indices, add operations take \(O(1)\) time. Since we can remove elements in \(O(1)\) time, this implies that updating a vertex's priority value and moving it to another bucket is also \(O(1)\) time which further implies that the algorithm takes \(O(|E_d|)\) time for this step (since we may do it for each edge in \(E_d\)).

Altogether this implies the runtime of our algorithm is \(O(|V_d|) + O(|E_d|) + O(d) = O(|V_d| + |E_d| + d)\).

Correctness. The proof follows similarly as the one in (a.) The only difference is that we are adding nodes to buckets with the same priority.

Since we run our algorithm until all buckets are empty, this algorithm will also explore a vertex if and only if that vertex is within distance \(d\) from \(s\).

**Question 2** (Street Race Planning, 30 points). Leonard and Amy are street racers. Their city is given by a directed graph \(G\). For each edge \(e\), they have average times \(L(e)\) and \(A(e)\) that it takes them to traverse that edge. Howard is trying to fix a race in Amy’s favor. To do so he needs to find a course for this race (which must start and end at the same place) so that Amy’s average time to complete the course is smaller than Leonard’s. Give an algorithm to determine whether or not this is possible to do. For full credit, your runtime should be \(O(|V||E|)\) or better.

**Solution 2.**

Algorithm. Construct a new graph \(G’\) with the same vertices as \(G\). If there is an edge \(e\) in \(G\) from \(u\) to \(v\), then create an edge in \(G’\) from \(u\) to \(v\) with weight \(A(e) - L(e)\). Then, make a new vertex \(v’\) and add edges from \(v’\) to each vertex in \(G’\) with weight 0. Then, run Bellman-Ford choosing \(v’\) as the start vertex for \(|V| + 1\) rounds, and seeing if there exists any negative weight cycle. If there is a negative weight cycle, Howard can fix the race in Amy’s favor. If not, then Howard cannot.

Correctness. We need to prove that Howard can fix the race in Amy’s favor iff our algorithm finds a negative weight cycle.

If our algorithm finds a negative weight cycle, then Howard can fix the race in Amy’s favor. A negative weight cycle in \(G’\) would exactly correspond to a path \((v_1, v_2, v_3, ..., v_n, v_1)\) with edges \((e_1, e_2, ..., e_n)\) where for \(i < n, e_i\) is the edge from \((v_i, v_{i+1})\), and \(e_n\) is the edge from \((v_n, v_1)\), where

\[
\sum_{i=1}^{n} A(e_i) - L(e_i) < 0
\]

This means the average time Leonard needs to complete the race will be greater than Amy’s time, which is exactly a way for Howard to fix the race in Amy’s favor.

If our algorithm does NOT find a negative weight cycle, then Howard CANNOT fix the race in Amy’s favor. We know from above that a negative weight cycle would exactly correspond to a way for Howard to fix the favor in Amy’s favor. So, if we can show that \(G’\) has no negative weight cycle, we can show that Howard cannot fix the race in Amy’s favor. Bellman-Ford finds all negative weight cycles reachable from the source, and by definition our source \(v’\) can reach every other vertex. So, if Bellman-Ford does not find a negative weight cycle, there is no negative weight cycle in \(G’\). Thus, if our algorithm does not find a negative weight cycle, then Howard cannot fix the race in Amy’s favor.

So, our algorithm is correct.
**Complexity.** The graph construction takes linear time, and Bellman-Ford takes \( \Theta(|V||E|) \) time, so the overall complexity is \( \Theta(|V||E|) \).

**Question 3** (Number of Shortest Paths, 40 points). Let \( G \) be a directed graph with positive edge weights and two specified vertices \( s \) and \( t \).

(a) Find a way to compute the set \( H \) of directed edges of \( G \) that are part of some shortest path from \( s \) to \( t \). [Note: The solution to Q2 on Exam 1 from Spring 2018 might be useful here] [10 points]

(b) Given the above, find an algorithm to compute some particular shortest path from \( s \) to \( t \) in \( G \). [Hint: show that any path from \( s \) to \( t \) using only edges of \( H \) is a shortest path] [10 points]

(c) Find an algorithm to compute the number of shortest paths from \( s \) to \( t \) in \( G \). [Hint: you may want to show that \( H \) is a DAG] [20 points]

For full credit, all algorithms here should be near linear (up to log factors) time.

**Solution 3.**

(a)

**Algorithm.** Let \( G^R \) be the reversed graph of graph \( G = (V, E) \), i.e. \( G^R = (V, E^R) \), such that there exists an edge \((u, v)\) with weight \( x \) in \( E^R \), if and only if there exists an edge \((v, u)\) with weight \( x \) in \( E \) (we get \( G^R \) by reversing edges in \( G \)). Now, let’s run Dijkstra from \( s \) in \( G \) and find the shortest distances to all vertices and store them in array \( d_2 \). Then, let’s run Dijkstra in \( G^R \) from \( t \) and find all shortest distances to all vertices and store them in array \( d_1 \). We create an empty list and add some edge \((u, v)\) in \( G \) to it, if and only if \( d_1(t) = d_1(u) + d_1(v) + w(u, v) \). At the end this list will be equal to set \( H \).

**Correctness.** We must show, that edge \((u, v)\) belongs \( H \), if and only if \( d_1(t) = d_1(u) + d_1(v) + w(u, v) \). If \((u, v)\) belongs to \( H \), then there exists a shortest path \( p \) from \( s \) to \( t \), which contains \((u, v)\). Let \( p_1 \) be the sub-path of \( p \) from \( s \) to \( u \) and let \( p_2 \) be the sub-path of \( p \) from \( v \) to \( t \). We have, that \( d_1(u) \leq w(p_1) \) and \( d_1(v) \leq w(p_2) \). The first inequality is true by the definition of \( p_1 \). For the second one, we can show that the shortest distance from \( v \) to \( t \) in \( G \) equals the shortest distance from \( t \) to \( v \) in \( G^R \). It is indeed true, because by the definition of \( G^R \), for each path in \( G^R \) there will be one and only path \( G \) with the same vertices and total weight, but with each edge in the opposite direction (we can get one path by reversing the edges in the other one). This means that the shortest distance from \( v \) to \( t \) in \( G \) can’t be shorter than the shortest distance from \( t \) to \( v \) in \( G^R \), because otherwise we would have reversed all the edges in the shortest path from \( v \) to \( t \) in \( G \) and obtained a shorter path from \( t \) to \( v \) in \( G^R \) than the one which was claimed to be the shortest. Thus, we have:

\[
\begin{align*}
&d_1(u) \leq w(p_1) \\
&d_1(v) \leq w(p_2) \\
&w(p_1) + w(p_2) \leq d_1(u) + d_1(v)
\end{align*}
\]

which implies \( d_1(u) + d_1(v) = w(p_1) + w(p_2) \), or \( d_1(u) + d_1(v) - w(u, v) = w(p_1) + w(p_2) + w(u, v) = d_1(t) \).

Now, if \( d_1(t) = d_1(u) + d_1(v) + w(u, v) \), then the path which goes from \( s \) to \( u \), uses edge \( u, v \) and then goes from \( v \) to \( t \) has the length which equals the length of the shortest path. Thus, it is a shortest path itself. Therefore, \((u, v)\) belongs to a shortest path.

**Complexity.** Reversing a graph takes \( O(|V| + |E|) \) time, as we only need to reverse each edges. We run Dijkstra twice in \( O(|V||\log|V| + |E|) \) and then for each vertex check whether the condition is satisfied, which adds up \( O(|V|) \). Therefore, overall time complexity is \( O(|V||\log|V| + |E|) \).

(b)
Algorithm. Run DFS from $s$. Each time when DFS visits a vertex $v$ from vertex $u$, make $\text{prev}[v] = u$. Define $\text{prev}[s] = -1$. When DFS finishes running, we should output $t, \text{prev}[t], \text{prev}\{\text{prev}[t]\}, \ldots, s$. We output this sequence in the following way:

while $t \neq -1$:
    • add $t$ to a list
    • $t = \text{prev}[t]$

reverse the list and return it

Correctness. Let’s consider any path from $p$ from $s$ to $t$ which only uses edges from $H$: $v_1, v_2, v_3, \ldots, v_{n-1}, v_n$, where $v_1 = s$ and $v_n = t$. By definition, we know that $w(v_{i-1}, v_i) = d_s(t) - d_s(v_{i-1}) - d_t(v_i)$ for all $2 \leq i \leq n$. Let’s sum all these equations by $i$:

$$\sum_{i=2}^{n} w(v_{i-1}, v_i) = (n - 1) \cdot d_s(t) - \sum_{i=2}^{n-1} (d_s(v_i) + d_t(v_i))$$

$$w(p) = d_s(t)$$

Thus, any path from $s$ to $t$ is the shortest path. To output some path from $s$ to $t$, we use a DFS algorithm.

Complexity. The time complexity is the time complexity of a DFS and a while loop. A while loop outputs a path and a path has $O(|V|)$ vertices. Each step in the while loop runs in a constant time. Thus, overall time complexity is $O(|V| + |E|)$.

(c) Algorithm. Topologically sort the graph. Let $f[v]$ be the number of paths from $s$ to $v$ in the graph. Let’s initialize it as $f[s] = 1$ and $f[v] = 0$ for any other $v$. Now, process vertices in topological order: for each vertex $v$ traverse all its edges $(v, u)$ and for each such edge make $f[u] + = f[v]$. The answer for the question is $f[t]$.

Correctness. It is easy to show that the graph that consists only from edges in $H$ will not have any cycles. We have shown above that any path from $s$ to $t$ will have the length $d_s(t)$. If we have a cycle, we can always choose a path from $s$ to $t$ that includes this cycle as a sub-path. The length of this path should equal $d_s(t)$, but this is not possible: if we remove the cycle from the path, then we still have a path from $s$ to $t$, but its length must be shorter than $d_s(t)$ (because the total length of the cycle is positive), but this is a contradiction.

Thus, we have a DAG and we can find its topological order. Now, let’s show that $f[v]$ will indeed be equal to the number of paths from $s$ to $v$ when our algorithm finishes processing $v$. We will prove it by induction:

(a) Base case: $f[s] = 1$. Indeed, there is one and only one path from $s$ to $s$: it is $s$ itself.

(b) Inductive hypothesis: assume for $v_i$, where $i < n$, the statement is true.

(c) Inductive step: Let’s prove it for $f[v_n]$. All paths from $s$ to $v_n$ can be divided into distinct subsets (that do not intersect with each other) by their last edge $(v_i, v_n)$, where $i < n$ (because of the topological order). But the number of paths from $s$ to $v_i$ equals $f[v_i]$. Thus, if we find the sum of all such $f[v_i]$, then $f[v_n]$ will be computed correctly. According to inductive hypothesis, when we visit $v_n$, value $f[v_i]$ has been computed correctly for any vertex $v_i$, $i < n$. Due to topological ordering, any vertex that has an edge to $v_n$ will be processed before $v_n$. Thus, by the time we visit $v_n$, we will have all the necessary values to compute $f[v_n]$.

Complexity. Topological sort takes $O(|V| + |E|)$ time. Then, we traverse each vertex and for every vertex we traverse its each edge. We process every edge in a constant time and each edge will be processed only once. Thus, the complexity will be $O(|V| + \sum_{v \in V} \text{deg}(v)) = O(|V| + |E|)$.

Question 4 (Extra credit, 1 point). Approximately how much time did you spend on this homework?