CPSC 366—Problem Set #2

February 25, 2021

1 Problem 1: Minimum Weight Path

1.1 Algorithm Design

1. Construct directed graph $G' = (V, E', w')$ in the following manner:
   (a) Iterate over each edge E_i in the original set of edges E. E_i is an undirected edge between two vertices $u, v \in V$
   (b) Add the directed edge $e' = (u, v)$ (edge from vertex u to v) to E' and set $w'(e') = w(v)$
   (c) Add the directed edge $e' = (v, u)$ to E' and set $w'(e') = w(u)$
2. Run Dijkstra's on graph G' with source vertex s and destination vertex t with the exception that $\delta(s)$ is initialized to $w(s)$ instead of 0.

Running Time: $O((n + m) \log n)$

1.2 Proof of Correctness

Proof. We proved in class that Dijkstra's algorithm computes the shortest path from a source vertex (let's call it v_1 in this proof) to all other vertices in V, including the destination vertex (let's call it v_n). To prove the correctness of my algorithm, I just need to show that the minimum weight path between v_1 and v_n that was calculated on G' is still the minimum weight path on G.

Dijkstra's algorithm returns a path of vertices $(v_1, v_2, ..., v_n), v_i \in V$ as well as the total weight of this path from v_1 to v_n. The edges used on this path are $\{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$. According to how I constructed the directed edges in G', the sum of the weights of these edges that make up the path is $w(v_2) + w(v_3) + ... + w(v_n)$.

However, we also have to consider the modification I made to Dijkstra's algorithm, which is that $\delta(v_1)$ is initialized to $w(v_1)$ instead of 0. This modification adds $w(v_1)$ to the total weight of any path returned by Dijkstra's because when processing a vertex u and updating it's neighbors v, the $\delta(u)$ term in this calculation $\delta(v) = \delta(u) + w(v)$ will always either be $\delta(v_1)$ or have already included it since v_1 is the first vertex that is processed.

Thus, the total weight of the path $\{(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)\}$ in G' becomes $w(v_1) + w(v_2) +$

$w(v_3) + ... + w(v_n)$. This is precisely the total weight of a path involving vertices $v_1, v_2, ...v_n$ in G.

The path calculated on G' is still the minimum weight path on G using proof by contradiction. Suppose $\exists v_j \in V$ such that v_i to v_j to v_{i+2} has less weight than v_i to v_{i+1} to v_{i+2}. In other words, $w(v_j) < w(v_{i+1})$. However, suppose for the sake of contradiction that Dijkstra returns the path including v_i, v_{i+1}, v_{i+2}. Since $\{(v_i, v_j), (v_j, v_{i+2})\} \in E$, $\{(v_i, v_j), (v_j, v_{i+2})\} \in E'$ by definition of how I constructed E'. Thus, the path (v_i, v_j, v_{i+2}) exists in G', and it obviously has a smaller weight than (v_i, v_{i+1}, v_{i+2}). Dijkstra, being a correct shortest path algorithm, would have chosen the path including v_j instead of v_{i+1}, and so a contradiction has been reached. Hence, the path on G' returned by Dijkstra must also be the minimum weight path on G.

This algorithm runs in $O((n + m) \log n)$ time, where $n = |V|$ and $m = |E|$. This is because the preprocessing step of constructing G' only takes $O(m)$ time since it just requires iterating over each edge. After that, the algorithm runs Dijkstra's on n vertices and $2m$ edges, which has a running time of $O((n + m) \log n)$ if implemented with a binary min-heap priority queue. □

2 Problem 2: Feedback Edge Set

2.1 Algorithm Design

1. Construct undirected graph $G' = (V, E, w')$ where $w'(e) = -w(e), \forall e \in E$.
2. Run Kruskal's algorithm on G' to obtain F', the set of edges that make up the Minimum Spanning Tree for G'
3. The minimum-weight feedback edge set of G is then just $F = E \backslash F'$

Running Time: $O(m \log n)$

2.2 Proof of Correctness

Lemma: Adding an edge between 2 existing vertices in a spanning tree $T = (V, E)$ will create a cycle.

Proof. Suppose vertices $u, v \in V$ are the vertices between which you will add an edge and that edge $(u, v) \notin E$. Since T is a spanning tree, there is already a path from u to v. Adding an edge between u and v will create another path between u and v, thus creating a cycle. □

Proving My Algorithm's Correctness

Proof. By definition, a graph $G'' = (V, E \backslash F, w)$ has to be acyclic. So, if we want to minimize the total weight of all edges in F, we want G'' to have as many edges as possible—while still remaining acyclic—and also have the biggest edge weights possible.

The "as many edges as possible" criteria can be satisfied by spanning trees (or forests of spanning trees if G is disconnected), which cover all vertices with minimum possible number of edges. Thus, spanning trees have no cycles, and in fact, by the Lemma above, it can be shown that adding an extra edge would create a cycle in the graph.

The "maximum edge weights as possible" criteria can be satisfied by maximum spanning trees, which are similar to minimum spanning trees except that the weights are maximum.

By negating the weights to form G', we make it so that edges that used to have the largest weights in G now have the smallest weights in G' and vice versa. So, when Kruskal's algorithm picks out the edges F' that form the minimum spanning tree of G', it is actually picking out the edges that would form the maximum spanning tree of G. Since Kruskal's greedily inserts edges until no more edges can be inserted, we know that no more edges can be added to F'.

Also, we don't have to worry about negative weights because when these weights are negated, they become positive and are processed after all of the previously positive but now negative weights.

Finally, we also don't have to worry about G being disconnected because unlike Prim's, Kruskal's algorithm still will find the minimum spanning tree for each connected component of the graph. This algorithm ensures that no more edges can be added to F' and that F' has the maximum weight possible. Thus, Feedback Edge Set $F = E \backslash F'$ has the smallest weight possible because the sum of the total weights in F and F' must sum to the total weights of all edges in E; $F \cap F' = \emptyset$

The algorithm runs in $O(m \log n)$ time because preprocessing just takes $O(m)$ time since you are just iterating over all edges in E once, while Kruskal's algorithm takes $O(m \log n)$ time. □

3 Problem 3: Shortest Paths with Negative Edges

3.1

Proof. The key difference between algorithm 1 and Dijkstra's is that algorithm 1 allows the program to process the same node more than one time. In the worst case scenario, this algorithm will have to check every single possible path from source vertex s to destination vertex t. Assuming in the worst case that the shortest path from s to t involves all N vertices in V, then there are at most $(N-1)!$ permutation of paths to consider. Each path consists of N vertices to process, so the max number of iterations over the while loop is $N * (N-1)! = N!$.

Algorithm 1 only updates a vertex i if it has found a path that is shorter than what was originally the path from s to vertex i. In the worst case, Algorithm 1 considers every single path from s to t, and if it only considers another path if that path is better, the very last path to a vertex i that Algorithm 1 computes must be the shortest path to i in the graph; there is no other possible path that could be shorter because the algorithm already processed all of them and still decided to move on.

If Algorithm 1 terminates before considering all possible paths to each vertex i, then that must mean that it skipped suboptimal paths that would lead to longer weights. In the algorithm, the edge (u,v) is only relaxed if $\delta(v) > \delta(u) + l(u,v)$. If $\delta(u) + l(u,v) > \delta(v)$, then there is no point including (u,v) because for all edges after vertex v, the path that includes (u,v) will always have a bigger weight than the optimal path that reached vertex v without using edge (u, v) $\square$

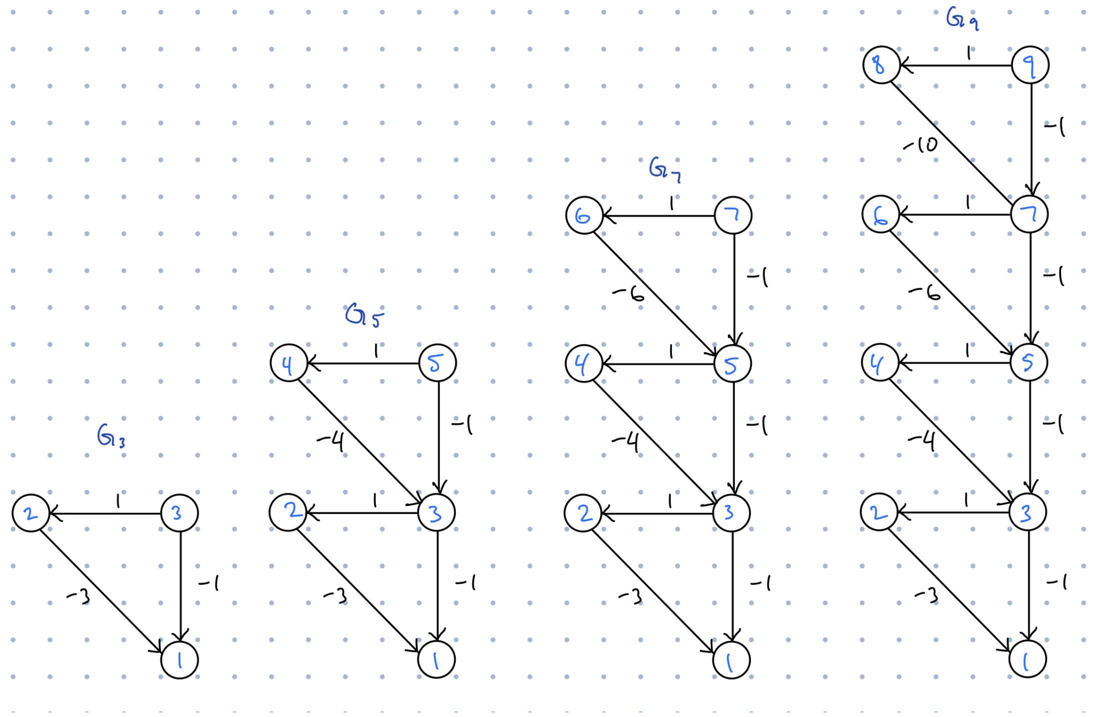
Running Time: $O(n \log n)$

3.2

Proof. The idea for a graph G_n on which Algorithm 1 takes at least 2^{cn} iterations to terminate is to set $c = \frac{1}{2}$. Then, we just need to show that there is a way to add 2 nodes to G_n to obtain a G_{n+2} that takes double the amount of iterations that G_n takes to terminate. Consider the following sequence given by $a_i = -(2^{i-1}) - 2$ for $i \in \mathbb{N}$

$$\{-3, -4, -6, -10, -18, ...\}$$

Then, construct graphs $G_3, G_5, G_7, G_9, ...$ in the following manner:



In the graphs above, the source of G_i is vertex i, and the destination is vertex 1. For each "triangle", there are 2 paths downwards. You can either take the right path (call this R), which has the locally optimal weight $w(R_i) = -1$ or the left path (call this L), which has a globally optimal weight $w(L_i) = 1 + a_i \leq -2$. Let $N = \frac{i-1}{2}$ for G_i be the number of "triangles". Finally, represent paths from source to destination vertex like so: $(m_1, m_2, ..., m_N)$ where $m_i \in L, R$.

Algorithm 1's first path will be $(R_1, R_2, ..., R_N)$ because the -1 weight edges seem optimal compared to the 1 weight edges that branch off to the left. The goal of this method of graph construction is then to have Algorithm 1 try every combination of Right and Left paths starting from

$(R_1, R_2, ..., R_N)$ and working down to $(L_1, L_2, ..., L_N)$. If R moves are represented by 1's and L moves are represented by 0's, this is equivalent of going from $2^N - 1$ to 0 in binary. This means that we need to process vertex 1 2^N times. Path $(R_1, ..., R_N)$ has a total weight of $-N$. In order to process vertex 1 2^N times, the weight of the path that ends at vertex 1 must be updated $2^N - 1$ times. The most efficient way to do this is to decrement the weight $2^N - 1$ times from $-N$. In other words, the weight of the paths must be $-N, -N-1, -N-2, ..., -N-2^N+1$. We just said that the final path that Algorithm 1 will try is $(L_1, ..., L_N)$, so this path must have weight $-N - 2^N + 1$. We can prove that the sequence I stated above works by induction.

Theorem: $w((L_1, ..., L_N)) = -N - 2^N + 1$

Base Case: N=1, $w((L_1)) = 1 - 3 = \boxed{-2} = -1 - 2^1 + 1$

Inductive Step: Assume the theorem holds for $N = K, K > 1$. Show that it still holds for $N = K + 1$.
Given that

$$w((L_1, ..., L_K)) = -K - 2^K + 1$$

Consider

$$\begin{aligned} w((L_1, ..., L_{K+1})) &= w((L_1, ..., L_K)) + w(L_{K+1}) \\ &= -K - 2^K + 1 + 1 - 2^{K+1-1} - 2 \\ &= -K - 2^{K+1} \\ &= -(K+1) - 2^{K+1} + 1 \end{aligned}$$

Thus, the inductive step holds true, and the theorem is proven.

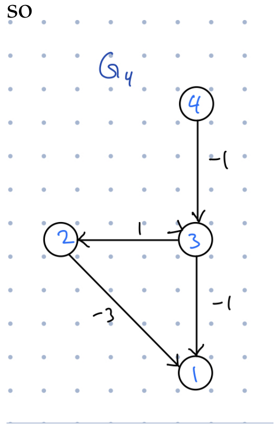
Now, we need to show that Algorithm 1 truly does compute all 2^N paths from $(R_1, ..., R_N)$ to $(L_1, ..., L_N)$. This can once again be done with induction.

Theorem: Algorithm 1 computes all 2^N paths from $(R_1, ..., R_N)$ to $(L_1, ..., L_N)$.

Base Case: N=1, Algorithm 1 first takes path (R_1) from vertex 3 to 1 because it has edge weight -1. Then, it finds path (L_1) from vertex 3 to 2 to 1 because it has total weight of -2.

Inductive Step: Assume the theorem holds for $N = K, K > 1$. Show that it still holds for $N = K + 1$.
Given that a graph G_i with $N = \frac{i-1}{2} = K$ "triangles" discovers 2^N paths, we see that adding another 2 vertices to get graph G_{i+2} with $N = \frac{i+1}{2} = K + 1$ doubles the number paths available. This is because that algorithm 1 will always take path R_1 over L_1 first because the first edge has weight -1. This will go into the $N = K$ subgraph, which generates 2^N paths. After this terminates, algorithm 1 will then consider path L, which we previously proved has a small enough weight to once again call the $N = K$ subgraph. This generates another 2^N paths for a total of $2^N + 1$ paths. Thus, the inductive step holds and the theorem is proven true.

It is easy to see that a graph G_i takes at least $2^{ci} = 2^{\frac{i}{2}}$ iterations to terminate when i is odd because vertex 1 on its own gets processed $2^N = 2^{\frac{(i-1)}{2}}$ times. In order for vertex 1 to be processed $2^{\frac{(i-1)}{2}}$ times, vertices 2 and 3 must have been each processed $2^{\frac{(i-1)}{2}-1}$ times. Summing just these 3 numbers together gives you $2^{\frac{i+1}{2}}$ iterations. For an even i, you can just place the extra vertex like so



In this case, vertices 1, 2 and 3 are still processed $2^{\frac{i}{2}}$ times, and the extra vertex at the top adds one more iteration. Thus, for even i's, the total number of iterations is $\geq 2^{\frac{i}{2}} + 1 > 2^{\frac{i}{2}}$. Hence, for any $i \geq 3$, G_i takes at least $2^{\frac{i}{2}}$ iterations to terminate. $\square$