CSE 101 Homework 3

Winter 2021

This homework is due on gradescope Friday February 5th at 11:59pm pacific time. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in LaTeXis recommend though not required.

Question 1 (Public Transit on a Budget, 40 points). *Lars is trying to get around town. He has various options for transportation with the possible routes represented by edges of a directed graph G. Each edge e has a positive integer cost* $\text{cost}(e)$ *dollars and a time it takes to traverse* $\text{time}(e)$. *Lars has a limited number N of dollars and would like to get between two locations (s and t) in as little total time as possible.*

(a) Give an algorithm that given G, s, t, the functions cost *and* time *and the total budget N, determines the shortest time to get from s to t under the budget. For full credit your algorithm should run in time $O(N(|V| + |E|))$ or better. [20 points]*

(b) Suppose that some routes are allowed to have a cost of 0. *Provide an algorithm to solve the new version of this problem with runtime $O(N(|V|\log(|V|) + |E|))$ or better. [20 points]*

Solution 1.

(a) We propose an algorithm similar to Bellman-Ford. Let $d_k(w)$ be the length of the shortest path from s to w with a total cost of at most k dollars. Then we have $d_k(w) = \min[d_{k-\text{cost}(v,w)}(v) + \text{time}(v, w)]$ for all edges $(v, w) \in E$. We compute this in order for $k = 1..n$ for every vertex and return $d_N(t)$. Pseudocode for our algorithm is as follows (note the similarity to Bellman-Ford):

```
StingyTransit(G, s, t, cost, time, N)
    d_0(v) = inf for all v
    d_0(s) = 0
    For k = 1 to N
        For w in V
            d_k(w) = min[d_{k-cost(v,w)}(v)+time(v,w)]
        d_k(s) = min(d_k(s), 0)
    Return d_N(t)
```

Each round takes $O(|V| + |E|)$ time and we run our algorithm for $O(N)$ rounds, so our algorithm is $O(N(|V| + |E|))$ in total, as desired.

Since our algorithm computes $d_k(w)$ in order from $k = 1$ to $k = n$, and $\text{cost}(v, w)$ is always positive, we know that for every incoming edge (v, w) our algorithm has already computed $d_{k-\text{cost}(v,w)}(v)$, so our algorithm at the very least runs without error. We approach a proof of correctness through strong induction. For the base case, note that clearly $d_0(w) = \infty$ for all $w \neq s$, and $d_0(s) = 0$. Now, assume that $d_j(w)$ is computed correctly for all w for all $1 \leq j \leq k - 1$. We want to show that $d_k(w)$ is computed correctly for all w. First, the shortest path to any non-source vertex w must always pass through a neighboring vertex with an incoming edge to that vertex, so the length (time) of such a path with cost $\leq k$ is simply $\min[d_{k-\text{cost}(v,w)}(v) + \text{time}(v, w)]$. Since $\text{cost}(v, w)$ is always positive, $d_{k-\text{cost}(v,w)}(v)$ must have already been computed correctly by the inductive hypothesis, so $d_k(w)$ is correct. For the source, the only difference is that the shortest path to the source could be the path that just stays at the source (which has time 0), so we simply run $d_k(s) = \min(d_k(s), 0)$, and so $d_k(s)$ is also correctly computed. Thus $d_k(v)$ is computed correctly for all v, and so our algorithm correctly returns $d_N(t) =$ the shortest time to get from s to t with cost $\leq N$, as desired.

(b) Note that our algorithm above fails because it is possible that $d_{k-\text{cost}(v,w)}(v)$ has not yet been computed on the k^{th} round (that is, if $\text{cost}(v,w) = 0$), and there is no order in which we can compute $d_k(w)$ that guarantees that $d_k(v)$ has already been computed for all $(v,w) \in E$ (since cycles could exist in the graph). Thus, we run the above algorithm, but for each round (for each value of k), when we compute $d_k(w) = \min[d_{k-\text{cost}(v,w)}(v) + \text{time}(v,w)]$, we only consider edges (v,w) such that $\text{cost}(v,w) > 0$. Then at the end of the round, we run a modified Dijkstra's starting from s, with $\text{dist}(v)$ initialized to $d_k(v)$ for all v, but only considering edges where $\text{cost}(v,w) = 0$. Pseudocode for our algorithm is as follows (note that the pseudocode for Dijkstra's is heavily based off of the pseudocode from the slides):

```
STWithFreeRides(G, s, t, cost, time, N)
    d_0(v) = inf for all v
    d_0(s) = 0
    For k = 1 to N
        For w in V
            d_k(w) = min[d_{k-cost(v,w)}(v)+time(v,w)] if cost(v,w) > 0
        d_k(s) = min(d_k(s), 0)
        Dijkstra's(G, s, cost, time, d, k)
    Return d_N(t)

Dijkstra's(G, s, cost, time, d, k)
    Initialize Priority Queue Q // sorted by d_k(v)
    For v in V
        Q.Insert(v) // with key d_k(v)
    While (Q not empty)
        v = Q.DeleteMin()
        For (v,w) in E, where cost(v,w) = 0
            If d_k(v) + time(v,w) < d_k(w)
                d_k(w) = d_k(v) + time(v,w)
                Q.DecreaseKey(w)
```

Each round takes $O(|V|\log|V| + |E|)$ time (since Dijkstra's is $O(|V|\log|V| + |E|)$) and we run our algorithm for $O(N)$ rounds, so our algorithm is $O(N(|V|\log|V| + |E|))$ in total, as desired.
The idea here is that for each round, the initial values of $d_k(v)$ (before running Dijkstra) represent the length of the best cost $\leq k$ path to v that does not end in a cost 0 edge. Thus, if we want to find the best overall cost $\leq k$ path, we need to think about paths that start with a path to some v taking time $d_k(v)$ and then continue with some sequence of cost 0 edges. This is equivalent to finding paths in a new graph with 1) a new vertex s_0 with edges of weight $d_k(v)$ to each vertex v and 2) the cost 0 edges from the original graph. Note that this is essentially the graph that our algorithm runs Dijkstra on, so at the end of each round, $d_k(v)$ is updated to accurately represent the actual distance of the shortest valid path to each vertex, and running the rest of the algorithm should give the correct result.

Question 2 (Negative Cycle Finding, 35 points). *We know how to use Bellman-Ford to determine whether or not a weighted, directed graph G has a negative weight cycle. Give an $O(|V||E|)$ time algorithm to find such a cycle if it exists. Hint: If there is such a cycle use Bellman-Ford to find a vertex v with $\text{dist}_{|V|-1}(v) > \text{dist}_{|V|}(v)$ and compute the paths involved. From this you should be able to find a cycle. You may also need to modify your graph some to deal with the possibility of a negative weight cycle not reachable from your chosen starting vertex s.*

Solution 2.

The standard Bellman Ford algorithm can be easily modified to find the negative cycle if it exists in the graph. First we note that in order to handle the case that a negative weight cycle exists in the graph but is not reachable from the source node, we can introduce an extra node in the graph and add an edge directed from this node to every other node in the graph with 0 weight. Then if we choose this node as the source node, we are guaranteed that if a negative weight cycle exists in the graph, it would be reachable from this source node. Therefore, without loss of generality, we assume that the negative weight cycle is reachable from the source node.

Now, we can easily modify the standard Bellman Ford algorithm to find the negative weight cycle if one exists by keeping track of the parent of a node whenever we update the distance of the end node of an edge. If we update the distance of the end node of an edge in the final iteration of Bellman Ford, that would mean that there exists a negative weight cycle in the graph. Then we find a node that is a part of the cycle and retrace the parents of this node to find the complete cycle. Pseudocode for this algorithm:

`FindNegativeCycle`(G, s):

 Initialize lists $p_k[]$ to store the parents of each node, $p_0(s) = s$ and $p_0(v) = -1$ for all $v \neq s$

 Initialize lists $d_k[]$ to store the distances of nodes from s, $d_0[v] = INF$ for all $v \neq s$ and $d_0[s] = 0$

 For $i = 1 \longrightarrow |V|$:

  For $w \in G.V$:

   $d_i(w) = min(d_{i-1}(v) + l(v, w))$

   $p_i(w) = argmin(d_{i-1}(v) + l(v, w))$

  $d_i(s) = min(d_i(s), 0)$

  $p_i(s) = s$

 $cycle\_node = -1$

 For v in $G.V$:

  If $d_{|V|}(v) < d_{|V|-1}(v)$:

   $cycle\_node = v$

 If $cycle\_node = -1$:

  No negative weight cycle exists in the graph and we return False

 Initialize list $path[]$ to store the $|V|-$length path from $cycle\_node$ to the source node, s

 For $i = |V| \longrightarrow 1$:

  $path$.add($cycle\_node$)

  $cycle\_node = p_i[cycle\_node]$

 There must be a repeated vertex in the path[] list. The negative weight cycle is composed of the vertices between the repeat occurrences of this vertex.

Time Complexity: Since building the $path[]$ list takes $O(|V|)$ time and running the Bellman Ford algorithm takes $O(|V||E|)$ time the overall time complexity is $O(|V||E|)$.

Proof of correctness follows from the proof of correctness of the Bellman Ford algorithm with the additional bookkeeping step of storing the parent of a node. By finding a node for which $d_{|V|}(v) < d_{|V|-1}(v)$, a negative cycle is detected. Also, the negative weight cycle must lie on the $|V|$-length path having shortest weight from this vertex to the source. Therefore, we need to retrace the parents in order to find the $|V|$-length path from this vertex to the source having the smallest weight in order to find the negative weight cycle along this path.

Question 3 (Divide and Conquer Recursions, 25 points). *Give the asymptotic runtimes of the following divide and conquer algorithms.*

(a) *An algorithm that splits the input into two inputs of a two-thirds the size and then does $\Theta(n)$ extra work. [2 points]*

(b) *An algorithm that splits the input into five inputs of half the size and then does $\Theta(n^{5/2})$ extra work. [2 points]*

(c) *An algorithm that splits the input into four inputs of half the size and then does $\Theta(n^2)$ extra work. [2 points]*

(d) *An algorithm that splits the input into six inputs of a third the size and then does $\Theta(n^{3/2})$ extra work. [2 points]*

(e) *An algorithm that splits the input into two inputs of a third the size and then does $\Theta(n)$ extra work. [2 points]*

(f) *An algorithm that splits the input into two inputs of half the size and then does $\Theta(n\log(n))$ extra work. Note: you cannot use the Master Theorem in this case. You may have to do some work to derive the answer. [15 points]*

Solution 3.

For (a) to (e) we can simply apply Master Theorem: $T(n) = aT(\frac{n}{b}) + O(n^d)$;

(a) $a = 2, b = \frac{3}{2}, d = 1, T(n) = \Theta(n^{\log_{1.5} 2})$;

(b) $a = 5, b = 2, d = \frac{5}{2}, T(n) = \Theta(n^{2.5})$;

(c) $a = 4, b = 2, d = 2, T(n) = \Theta(n^2 \log(n))$;

(d) $a = 6, b = 3, d = \frac{3}{2}, T(n) = \Theta(n^{\log_3 6})$;

(e) $a = 2, b = 3, d = 1, T(n) = \Theta(n)$;

(f) Since we cannot express $n \cdot \log(n)$ as n^d, we cannot use the Master Theorem. The recurrence relation is now $T(n) = 2T(\frac{n}{2}) + \Theta(n \cdot \log(n))$. Assume that $n = 2^k$, then the height of the recurrence tree is $k+1$. On level i there will be 2^i recursive calls where each call will have size 2^{k-i}. So the time complexity of each recursive call on level i will be $\Theta(2^{k-i} \cdot \log(2^{k-i}))$. Given that there are 2^i calls on level i so the total time complexity on level i is $2^k \cdot \log(2^{k-i})$. As a result, $T(n) = \sum_{i=0}^{k} \Theta(2^k \cdot \log(2^{k-i}))$. Since $\log(x) = c \cdot \log_2(x)$ where c is a constant, $T(n) = \sum_{i=0}^{k} \Theta(n \cdot \log_2(2^{k-i})) = \Theta(n \cdot \sum_{i=0}^{k} (k-i)) = \Theta(n \cdot (\frac{k^2-k}{2})) = \Theta(n \cdot k^2) = \Theta(n \cdot \log^2(n))$. So for $2^k \leq n < 2^{k+1}$, we will have $\Theta(2^k \cdot \log^2(2^k)) \leq T(n) < \Theta(2^{k+1} \cdot \log^2(2^{k+1}))$. Since constant can be ignored in Θ notation, $T(n) = \Theta(n \cdot \log^2(n))$.

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