CSE 101 Homework 2 Solutions

Fall 2019

Question 1 (Return to Graphania, 40 points). Sylvester is still trying to plan his trip to Graphania and their newly uni-directional road system. He is given a list of cities and a list of one-way roads connecting pairs of them (i.e. a directed graph \( G \) on the cities). For each city he is given the price of the cheapest flight from his home to that city, and the cheapest flight from that city back home (there are no plane flights between cities in Graphania). He is trying to find the cheapest trip consisting of flying to some city from home, driving along some sequence of roads (free), and then flying home from the resulting city. For each of the following, provide a linear time algorithm for computing the best possible price.

(a) Give an algorithm that works if \( G \) consists of only a single strongly connected component. [10 points]

(b) Give an algorithm that works if \( G \) is a DAG. [10 points]

(c) Give an algorithm that works for an arbitrary graph \( G \). Hint: compute the metagraph and find a way to combine the above ideas. [20 points]

Solution.

(a) **Algorithm.** Let \( G = (V, E) \) be a directed graph consisting of only a single strongly connected component. Our algorithm proceeds as follows:

- Iterate through the list of inbound flights (from home to each city), and find the city with the cheapest inbound flight.
- Iterate through the list of outbound flights (from each city to home), and find the city with the cheapest outbound flight.
- Once finished, output the sum of (cheapest inbound flight) + (cheapest outbound flight).

**Correctness.** Let \( A_{in} \) and \( A_{out} \) be the inbound and outbound flight costs calculated by the algorithm. Since \( G \) is a single strongly connected component, every vertex of \( G \) is reachable from every other vertex, so we can travel by road between any two cities. Therefore, flying into the city with inbound flight cost \( A_{in} \) and driving to the city with outbound flight cost \( A_{out} \) is a valid trip. So the trip our algorithm computes the cost of is achievable.

Given a graph \( G \), let \( C = C_{in} + C_{out} \) be the cost of an achievable trip – we will show that the output of our algorithm is a cost \( \leq C \). Since our algorithm loops through all inbound flight costs and finds the cheapest one, \( A_{in} \) must be \( \leq C_{in} \). Similarly, since our algorithm loops through all outbound flight costs, \( A_{out} \) must be \( \leq C_{out} \). Therefore, the sum of \( A_{in} + A_{out} \) must be \( \leq C_{in} + C_{out} \leq C \).

**Runtime analysis.** Our algorithm iterates through each city and calculates the cheapest inbound and outbound flights. There are \( |V| \) cities total, and updating the cheapest flights takes constant time, so the runtime of our algorithm is \( O(|V|) \).

(b) **Algorithm.** Let \( G = (V, E) \) be a DAG. Our algorithm proceeds as follows:

- Topologically sort the vertices of \( G \), and reverse the order. Let \( L \) be the resulting list of vertices in reverse topological order (sort by smallest postorder first).
- For each vertex \( v \), we want to compute and store the cheapest outbound flight reachable from \( v \). Define the function \( R \) as \( R(v) = (\text{cheapest outbound flight reachable from vertex } v) \). We will compute \( R \) for all vertices:
- Iterate through the vertices of $G$ in the order given by $L$ (starting from the sinks of $G$).
- For each vertex $v$, take all of the neighboring vertices $u$ of $v$ such that $v$ has an outgoing edge $(v, u)$ to $u$.
- $R(v)$ is then $\min\{\min_u(R(u)), \text{cost of outbound flight from } v\}$, where $\min_u(R(u))$ is the minimum of the cheapest reachable outbound flights over all vertices $u$ with edges from $v$ to $u$.
- Loop through each vertex $v$, and store the sum of $(\text{cost of flight into } v) + R(v)$.
- Loop through all the vertices and return the smallest of the sums calculated in the above step. This will be the cost of the cheapest trip.

**Correctness.** Since $G$ is a DAG, it will have a topological ordering. So our list $L$ of vertices in reverse topological order is well-defined. We now show that our algorithm outputs the cheapest trip in $G$.

**Claim:** Our algorithm will correctly compute $R(v)$ for every vertex $v$ of $G$.

**Proof of claim:** We will prove this by induction on the position of $v$ in the list $L$.

**Base case:** $v$ is a sink vertex. Because of how we sorted our list $L$ of vertices, the sink vertices will be considered first by the algorithm. If $v$ is a sink, it has no outgoing edges to any other vertex. This means that no other vertex is reachable from $v$, so the cheapest flight out from $v$ must be equal to the cost of the outbound flight from $v$. There are no outgoing edges $(v, u)$ from $v$ to some other vertex $u$, so $R(v)$ contains the correct cheapest reachable outbound flight.

**Inductive step:** Assume that the algorithm calculates the cheapest reachable outbound flight for all vertices in the list $L$ up to (and not including) vertex $v$. This means that $R(u)$ contains the cheapest reachable outbound flight for all vertices $u$ with a smaller postorder than $v$. So, by the induction hypothesis, all of the vertices $u$ with edges from $v$ to $u$ will have the correct value of $R(u)$ computed. If an outbound flight is reachable from $v$, it is either a flight out of $v$ itself, or it is a flight reachable from some city with an edge from $v$ to $u$. Therefore, to find the cheapest reachable outbound flight from $v$, we first find the minimum of the cheapest outbound flights reachable from all vertices $u$ with edges from $v$ to $u$, then take the minimum of this value and $v$’s outbound flight cost. This is precisely $R(v) = \min\{\min_u(R(u)), \text{cost of outbound flight from } v\}$. We conclude that our algorithm computes $R(v)$ correctly.

**Correctness of algorithm:** Let $A_{\text{in}}$ and $A_{\text{out}}$ be the inbound and outbound flight costs calculated by the algorithm. Our algorithm iterates through every vertex and calculates the trip cost for flying into that city, driving to the city with the cheapest reachable outbound flight, and then flying out. By the above claim, the cheapest reachable outbound flights we compute for each vertex are in fact reachable. So, flying into the city with inbound flight cost $A_{\text{in}}$ and driving to the city with outbound flight cost $A_{\text{out}}$ is a valid trip, which means the trip our algorithm computes the cost of is achievable.

Given a graph $G$, let $C = C_{\text{in}} + C_{\text{out}}$ be the cost of an achievable trip. We will show that the output of our algorithm is a cost $\leq C$. Let $v$ be the vertex with $C_{\text{in}}$ as an inbound flight, and $w$ be the vertex with $C_{\text{out}}$ as an outbound flight. Since the trip is achievable, $w$ must be reachable from $v$. By the above claim, our algorithm computes the correct cheapest reachable outbound flights, which means $R(v) \leq C_{\text{out}}$. After computing $R$ for every vertex, our algorithm loops through all vertices and returns the minimum overall trip cost, using the computed values for cheapest reachable outbound flights. So, $A_{\text{in}} + A_{\text{out}} \leq C_{\text{in}} + R(v) \leq C$.

**Runtime analysis.** Topological sort of a graph takes $O(|V| + |E|)$ time. We then iterate through the reverse topological order, visiting each vertex once. The update rule for each vertex visits the outgoing edges from that vertex to compute the minima. Using the handshaking lemma, computing $R(u)$ for all the vertices costs $O(|V| + |E|)$. Finally, in the last two steps we iterate through the list of vertices in $O(|V|)$ time. So, the total runtime is $O(|V| + |E|)$

(c) **Algorithm.** Let $G = (V, E)$ be an arbitrary directed graph. Our algorithm proceeds as follows:

- Compute the metagraph of $G$ (a DAG whose vertices are the SCCs of $G$).
- Label each vertex of the metagraph with inbound flight cost = the minimum inbound flight cost of all vertices of the SCC in the original graph, and outbound flight cost = the minimum outbound flight cost of all vertices of the SCC in the original graph.
• Use the algorithm described in part 1b to find the cheapest trip of the metagraph. Return the result.

**Correctness.** Since the metagraph of $G$ is a DAG, we can apply the algorithm from part 1b. We now show that our algorithm outputs the cheapest trip in $G$.

Let $A_{in}$ and $A_{out}$ be the inbound and outbound flight costs calculated by the algorithm. Because part 1b is correct, the vertex of the metagraph with cost $A_{out}$ is reachable from the vertex with cost $A_{in}$. The in/outbound flight costs used for each vertex in our metagraph are the cheapest flights for each corresponding SCC, and since every vertex is reachable from every other vertex in a SCC, flying into the city with inbound flight cost $A_{in}$ and driving to the city with outbound flight cost $A_{out}$ must be a valid trip. So the trip our algorithm computes the cost of is achievable.

Given a graph $G$, let $C = C_{in} + C_{out}$ be the cost of an achievable trip – we will show that the output of our algorithm is a cost $\leq C$. Let $v$ be the vertex with $C_{in}$ as an inbound flight, and $w$ be the vertex with $C_{out}$ as an outbound flight. Since the trip is achievable, $w$ must be reachable from $v$, so the SCC that $w$ is in must be reachable from the SCC that $v$ is in. Let $S_{in}$ be the minimum inbound flight cost in $v$’s SCC, and let $S_{out}$ be the minimum outbound flight cost in $w$’s SCC. We must then have $S_{in} \leq C_{in}$ and $S_{out} \leq C_{out}$. By correctness of part 1b, we can then conclude that $A_{in} + A_{out} \leq S_{in} + S_{out} \leq C$.

**Runtime analysis.** Constructing the metagraph of $G$ takes linear time. We then iterate through each SCC and calculate the cheapest in/outbound flights which takes $O(|V|)$ time in total. We then run the algorithm from part 1b, which takes $O(|V| + |E|)$ time. Therefore, the overall runtime of the algorithm is $O(|V| + |E|)$.

**Question 2 (Modified Priority Queue, 10 points).** A Van Emde Boas priority queue is an implementation of the priority queue data structure so that if the entries in the queue are all integers between 1 and $M$, the basic operations can each be performed in $O(\log \log (M))$ time. Suppose that Dijkstra is run on a graph $G$ whose edge weights are all positive integers of size at most $n$. What runtime would you expect if the priority queue is implemented by a Van Emde Boas priority queue? When is this favorable to using a binary heap? What about a Fibonacci heap?

**Solution.** Let V and E denote the vertices and edges of graph G respectively. Suppose we want to find the shortest path from vertex $S_0$ to all other vertices. Since the edge weights are positive, the shortest path to any vertex (say $D_0$) will not have any repeated vertex. If the path contains a repeated vertex (eg. $S_0$, $v_0$, $v_1$, ..., $v_1$, ..., $D_0$), then we can simply remove all edges on the path between the repeated vertices and still get a path from $S_0$ to $D_0$. This path will be shorter than the earlier path because the removed edge weights are positive. Hence, the number of edges in the shortest path from $S_0$ to any other vertex in $V$ will be $\leq |V| - 1$. Since the edge weights are at most $n$, the length of the shortest path to any vertex will be at most $n(|V| - 1)$.

During Dijkstra’s, the priority queue stores the distances to the undiscovered vertices. Since these distances can be at most $n(|V| - 1)$, we can construct a Van Emde Boas (VEB) priority queue with $M = n(|V| - 1)$. Hence, Dijkstra’s will run in $\Theta((|V| + |E|) \log \log (n|V| - 1))$ which is equal to $\Theta((|V| + |E|) \log \log (n|V|))$

The runtime for Dijkstra’s with binary heap is $\Theta((|V| + |E|) \log |V|)$. The runtime for Dijkstra’s with Fibonacci heap is $\Theta(|V| \log |V| + |E|)$. The runtime for Dijkstra’s with Fibonacci heap is $\Theta(|V| \log |V| + |E|)$.

**Comparing VEB PQ to Binary Heap PQ:**

The implementation using VEB PQ will be better when:

$$(|V| + |E|) \log \log (n|V|) \leq (|V| + |E|) \log |V|$$

$$\Rightarrow \log \log (n|V|) \leq \log |V|$$

$$\Rightarrow \log (n|V|) \leq |V|$$

$$\Rightarrow n \leq \frac{2^{|V|}}{|V|}$$

**Comparing VEB PQ to Fibonacci Heap PQ:**
Consider two cases:

Case 1: $|E| \geq |V| \log |V|$

For Fibonacci PQ: $\Theta((|V|\log |V| + |E|)) = \Theta(|E|)$
For VEB PQ: $\Theta((|V| + |E|)\log |V|) = \Theta(|E|\log |V|)$ (because $|E| \geq |V|\log |V| \geq |V|$)
Since $|E| \leq |E|\log |V|$, Fibonacci heap PQ will be better than the VEB PQ asymptotically.

Case 2: $|E| < |V| \log |V|$

For Fibonacci PQ: $\Theta((|V|\log |V| + |E|)) = \Theta(|V|\log |V|)$
For VEB PQ: $\Theta((|V| + |E|)\log |V|) = \Theta(|E|\log |V|)$ (assuming the graph is connected $|V| \leq |E|$)
Hence, VEB PQ will be preferable when $\log |V| \leq \frac{|V|\log |V|}{|E|}$ or equivalently $n \leq \frac{2^{|E|\log |V|}}{|V|}.$

**Question 3** (Currency Trading, 50 points). Gordon works in currency trading. He has a portfolio of various currencies. He also knows a number of vendors each willing to trade one specific currency for another specific one at a given rate. For example, he might know someone willing to trade him 1.1 US Dollars for a Euro.

(a) Gordon wants to rebalance his portfolio, trading some of his currency $A$ for currency $B$. However, rather than finding a single vendor willing to perform the transaction, he is willing to make a series of trades in order to accomplish this. For example, he might find a vendor to exchange some of his currency $A$ for currency $X$, and another to trade that for currency $Y$ before finally trading for currency $B$. He would like to find a sequence of such trades that gives him the best possible exchange rate from $A$ to $B$. Give an algorithm that finds this best rate. You can assume that you are given a list of currencies and a list of vendors and the trades they are willing to make at what rates. For full credit your algorithm should run in polynomial time. [25 points]

(b) Gordon wonders if there is a way to make a profit just by trading. For example, there might be some series of trades he could make starting with currency $A$ that would end up with him ending up with more of currency $A$ than he started with. Give an algorithm to determine whether or not this is possible. For full credit your solution should be polynomial time. [25 points]

**Part A:**

**Algorithm:**
Construct a graph $G$ where the vertices are currencies in the given list. For each exchange a vendor is willing to do, add a directed edge between the corresponding currencies with weight of the edges as $R'' = \log(1/R)$. For any particular exchange between currency 1 and currency 2, the rate $R$ is defined as the units of currency 2 we can get by trading 1 unit of currency 1. Minimize $\sum R''$ in any path from $A$ to $B$. This is same as finding shortest path from $A$ to $B$. Observe that the weights can be negative if $R > 1$.

We employ Bellman-Ford algorithm to compute the shortest path weight from $A$ to $B$ by starting from $A$. Let the shortest path weight be $SP^*$. The best rate is $R^* = \exp(-SP^*)$.

**Run-time Analysis:**
Graph construction takes $O(|V| + |E|)$ operations. Note that we are adding exactly one edge for each possible exchange offered by vendors and exactly one vertex for each currency in the list where $|V|$ is number of currencies in the input and $|E|$ number of possible exchanges in the input.
Bellman-Ford takes $O(|V||E|)$ run-time. So, total run-time is $O(|V||E|)$

**Justification:**
If we do a series of exchanges, $X_0 \rightarrow X_1 \rightarrow X_2 \rightarrow X_3, ..., X_n$ at rates $R_1, R_2, ..., R_n$, starting from 1 unit of $X_0$ we end up with $\prod_{i=1}^{n} R_i$ units of $X_n$. So, the best rate from currency $A$ to $B$ is obtained by doing a series of exchanges so that the product of rates is maximum i.e. $R^* = \max(\prod R_i)$ in any series of exchanges (any path) from from $A$ to $B$. Conversely, if there is a best rate from currency $A$ to $B$, it has to be following a series of exchanges from $A$ to $B$. So, the best rate has to be the product of rate of each exchange.
it is the best rate, it should correspond to maximum product among the series of exchanges from A to B i.e. $R^* = \max(\prod R_i)$ on a path from A to B. Let $R'_i = 1/R_i$ and $R''_i = \log(R'_i)$. By the construction of our graph with edge weights as $R''_i$, if Bellman-Ford returns the min-weight of a path from A to B, it takes the form $SP^* = \min(\sum R''_i)$ by definition i.e. Bellman-Ford returns the path with least sum of weights from A to B.

$$SP^* = \min(\sum R''_i) = \min(\sum (\log(R'_i)) = \min(\sum (\log(1/R_i))) = \min(\log(1/\prod R_i)) = \min(-\log(\prod R_i)) = -\max(-\log(\prod R_i)) = -\log(R^*)$$. This implies $R^* = \exp(-SP^*)$.

An assumption made in this part of problem (discussed on piazza) is that there is no way to make infinite money. So, there is no cyclic series of exchanges with $\prod R_i > 1$ or $\prod R'_i < 1$ or $\sum R''_i = \sum \log(R'_i) = \log(\prod R'_i) < 0$ reachable from A. So, there are no negative cycles reachable from A. This proves that the Bellman Ford indeed gives $SP^*$ in our graph and equivalently the best rate $R^* = \exp(-SP^*)$.

**Part B:**

**Algorithm:**

Construct the same graph as part (a). Add a node $v_0$ and connect it to all vertices in the graph with an edge of weight 0. Run Bellman Ford from $v_0$ for an extra iteration to check if a negative cycle exists. If yes, we have discovered a way to make profit just by trading and equivalently Gordon can keep trading to make infinite money.

**Run-time:**

The run-time of this algorithm is same as earlier as we are running text-book Bellman-Ford. We have one extra vertex, but that doesn’t affect the asymptotic runtime. Negative cycle detection does not add any additional burden on run-time apart from adding one loop over $O(|E|)$ edges. So, the total run-time is $O(|V||E|)$.

**Justification:**

If we find a negative cycle reachable from $v_0$, we can say two things. Firstly, the cycle does not include $v_0$, since there is no way reach back to $v_0$ once we move along any of its edges. Secondly, we found a path (that doesn’t include $v_0$) with $\sum R''_i < 0 \implies \sum \log(1/R_i) < 0 \implies \log(1/\prod R_i) < 0 \implies 1/\prod R_i < 1 \implies \prod R_i > 1$. Gordon can keep following this cycle of exchanges to always end up with more money than he started with, i.e. he can make profit just by trading.

Conversely, if there is a way to make infinite money then $\prod R_i > 1$ along a cycle and this is equivalent to a negative weight cycle in the graph (with the similar logic as above). This cycle might start from any of its vertices (excluding $v_0$) but we will be able to reach that cycle from $v_0$. Since we will be able to reach that cycle, Bellman Ford will detect it.