CS 170 Efficient Algorithms and Intractable Problems
February 29, 2016 Liam Leahy

HW 4

Due February 15, 11:59pm

Partner: Stephen Qu

1. (15 pts.) Short Questions

For these questions, either provide a short justification for your answers (one or two sentences should be enough) or provide a counterexample (please make your counterexample as small as possible, for the readers' sake).

(a) We want to use the FFT to multiply two polynomials of degree 62 and 83, respectively. What would be the size n of the FFT needed? What is the corresponding ω? What is ω^{64}? (It is a very simple complex number.)

(b) Let G be a dag and suppose the vertices $v_1, \ldots, v_n$ are in topologically sorted order. If there is a path from v_i to v_j in G, are we guaranteed that $i < j$?

(c) Let $G = (V, E)$ be any strongly connected directed graph. Is it always possible to remove one vertex (as well as edges to and from it) from the graph so that the remaining directed graph is strongly connected?

(d) Consider the array `prev` computed by Dijkstra's algorithm ran on a connected undirected graph, starting at any vertex s. Is the graph formed by the edges $\{u, \texttt{prev}(u)\}$ a tree?

(e) Give an example where the greedy set cover algorithm does not produce an optimal solution.

Solution:

(a) n would be 256 as we should use the first power of two greater than or equal to the sum of the powers of the functions. ω would be $e^{i\pi/128}$. ω^{64} is $e^{i\pi/32}$ as we have the general form the the nth root of unity as $e^{2\pi i/n}$.

(b) Yes, as the graph is a DAG and it is topologically sorted, we know that there are no "backedges". Therefore, it is impossible to "go back". This means that it is necessarily the case that $i < j$, or else we would have a path "back up" the graph and would violate this property.

(c) Yes, this only removes the edges relevant to this vertex. Therefore, there is no case in which this would remove edges that are required to maintain the strongly connected property of the graph that remains.

(d) Yes. In order for it to not be a tree, there would necessarily be two edges leading to the same vertex at some point in the graph of $\{u, \texttt{prev(u)}\}$. But this would imply that there is more than one shortest path stored in `prev`, which contradicts its definition.

(e) Not required.

2. (15 pts.) All roads lead to Rome

You are the chief trade minister under Emperor Caesar Augustus with the job of directing trade in the ancient world. The Emperor has proclaimed that *all roads lead to (and from) Rome*; that is, all trade must go through Rome. In particular, you are given a strongly connected directed graph $G = (V, E)$ with positive edge weights, and there is a particular node $v_0 \in V$ (Rome). Give an efficient algorithm for finding shortest path between *all pairs of nodes*, with the one restriction that these paths must all pass through v_0 (Rome). Make your algorithm as efficient as you can, perhaps as fast as Dijkstra's algorithm.

(a) Give the efficient algorithm.

(b) Occasionally, Augustus will ask you for the (smallest) distance between two vertices. You want to do this as quickly as possible, so that Augustus does not have your head.

This is called a *distance query*: Given a pair of vertices (u, v), give the the distance of the shortest path that passes through v_0. Describe how you might store the results such that you require $O(|V|)$ storage and from your data structure you can compute the result in $O(1)$ time. For your answer, a clear description of the data structure and its usage is sufficient.

(c) On the other hand, the traders need to know the paths themselves.

This is called a *path query*: Given a part of vertices (u, v), give the shortest path itself, that passes through v_0. Describe how you might store the results such that you require $O(|V|)$ storage and from your data structure you can compute the result in $O(|V|)$ time. Again, a clear description of the data structure and its usage is sufficient.

Solution:

(a) **Main Idea:** Run Dijkstra's on Rome to find the shortest path from Rome to any other node. To find the shortest path from one node, through Rome, to any other node, we simply take the distance calculated using Dijkstra's from Rome to one city and add it to the distance from Rome to the other city.

Pseudocode:

Algorithm 1 Returns a generator function that returns the shortest distance from vertex u to vertex v or vice versa (whichever is smaller), through Rome.

```
1: function THROUGH_ROME_DISTANCES(G, v_0)
2:     G' ← REVERSE_EDGES(G)
3:     s ← DIJKSTRA(G, v_0)
4:     s' ← DIJKSTRA(G', v_0)
5:     function SHORTEST_PATH_THROUGH_ROME(u, v)
6:         return s'[u] + s[v]
7:     return SHORTEST_PATH_THROUGH_ROME
```

Proof: Since the graph is strongly connected, we know that there is a path from one vertex to any other vertex. The shortest path through Rome to another vertex must be the shortest path to Rome concatenated with the shortest path from Rome to the other vertex.

Proof by contradiction.

Consider the contrary, that there is a shorter path through Rome to the other city. This must take a path that is shorter either to Rome or to the city, as it must pass through both in that order. This contradicts our initial assumption that the path is the concatenation of those two shortest paths. Therefore, this path must be the shortest path connecting there three vertices.

Reversing the edges is necessary because the shortest path between two vertices could go in either direction. This is because the graph is directed. Therefore, we need the distances both from Rome to

every other vertex and vice versa. Reversing can be accomplished linearly in $O(|E|)$. Once we have the shortest paths in both directions the rest is a matter of adding and finding the best direction to go. □

Running Time: $O((|V|+|E|)\log|V|)$.

Justification: This performs Dijkstra's twice and reverses the edges once. We know the running time of Dijkstra's and the reversal is linear with respect to the number of edges. It performs only constant time operations in creating the generator function. Any query to the generator function will take constant time to compute as we already have all of the distances stored.

(b) This follows from above. We store two shortest-path trees constructed via Dijkstra's, with size $O(|V|)$. Querying is simple and is already explained above.

(c) This would require taking advantage of `prev` in Dijkstra's. We would compute the path by following `prev` through the graph and storing it in a running list. We would do the same for the inverted graph. For each vertex, we only have two values stored (the previous vertex in each direction) and there are $|V|$ total vertices. Thus, the additional storage required is $O(|V|)$. The running time for computing the path is $O(|V|)$ as we would be going through the entire list of vertices in the worst case. In order to obtain the path, we follow the `prev` values from u to v_0 in the inverted graph and then follow from v_0 to v in the first graph.

3. (15 pts.) Unique Shortest Path

Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V|+|E|)\log|V|)$ time.

Input: An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.

Output: A Boolean array `usp[·]`: for each node u, the entry `usp[u]` should be `true` if and only if there is a *unique* shortest path from s to u. (Note: `usp[s] = true`.)

Solution:

Main Idea: We perform Dijkstra's algorithm with an additional check to determine whether or not this is a unique shortest path. This will be a constant time operation and will therefore not affect the running time.

Pseudocode:

Algorithm 2 Evaluates the minimum distance to every node in the graph and also keeps track of whether or not this minimum distance is a unique minimum distance.

```
1:  function UNIQUE_SHORTEST_PATH(G, s)
2:      for all u ∈ V do
3:          DIST(u) = ∞
4:          PREV(u) = NULL
5:          USP(u) = TRUE                          ▷ by default, the path to a node should be unique
6:      DIST(s) = 0
7:      H ← MAKE_QUEUE(V)
8:      while H is not empty do
9:          u ← DELETE_MIN(H)
10:         for all (u, v) ∈ E do
11:             if DIST(v) > DIST(u) + L(u, v) then
12:                 DIST(v) = DIST(u) + L(u, v)
13:                 PREV(v) = u
14:                 USP(v) = TRUE
15:                 DECREASE_KEY(H, v)
16:             else if DIST(v) == DIST(u) + L(u, v) then
17:                 USP(v) = FALSE                 ▷ found another path with the same cost
18:     return USP
```

Proof: This is exactly Dijkstra's except that we are storing additional information. This information is only updated when we find that there is another *different* path that reaches this vertex but uses the same cost.

Consider any tentative shortest path found by Dijkstra's at any point during its processing. If it is the case that this is the shortest path to this vertex *and* there is another path that has the same cost that also reaches this vertex that we have not explored yet, then it is the case that we will explore this path at some point later. When we follow this alternate shortest path later, we find that the cost that we could update is the same and save this information.

If we find a path that is faster than the previously computed distance to any vertex in G, then it is the case that is the *first* path that yields this distance at this point. Therefore, it is necessarily true that this path is the unique shortest path at this point; thus, we update our array to save this information. If it the case that there is another path that has the same cost, it is definitely true that we have yet to explore it but will in the future. This is because, if we had already explored it, then it would already be saved as the new shortest distance to this vertex. □

Running Time: $O((|V|+|E|)\log|V|)$.

Justification: There is only a contant time operation appended to each iteration. Therefore, we do not change the running time of what is otherwise a vanilla Dijkstra's.

4. (20 pts.) Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \ldots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair i, j of currencies, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ for all i, j.

(a) The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*.) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ and two specific currencies s, t, find the most advantageous sequence of currency exchanges for converting currency s into currency t. We recommend that you represent the currencies and rates by a graph whose edge lengths are real numbers.

(b) In the economic downturn of 2016, the FEMO had to downsize and let Oski go, and the currencies are changing rapidly, unfettered and unregulated. As a responsible citizen and in light of what you saw in lecture this week, this makes you very concerned: it may now be possible to find currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \times r_{i_2,i_3} \times \ldots \times r_{i_{k-1},i_k} \times r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

You decide to step up to help out the World Bank. Given an efficient algorithm for detecting the presence of such an anomaly. You may use the same graph representation as for part (a).

Solution:

(a) **Main Idea:** This problem is exactly a restatement of finding a shortest path through a graph with negative edges. Therefore, we apply Bellman-Ford to the graph to find the minimum distance from s to t.

Pseudocode:

Algorithm 3 Finds the best combination of exchanges for some $r_{i,j}$ such that the cost of converting currency s to currency t is minimized. We take the negative log of every $r_{i,j}$ in order to be able to have each edge be some number that we can subtract instead of multiply. We want the negative because we want the minimum path, not the maximum.

```
1: function MINIMUM_CURRENCY_EXCHANGE(R[..., r_{i,j}, ...], s, t)
2:     R' ← []
3:     for all r_{i,j} ∈ R do
4:         APPEND(R', −log(r_{i,j}))
5:     G ← CONSTRUCT_GRAPH(R')
6:     return BELLMAN-FORD(G, s, t)
```

Proof: We want the best combination of $r_{s,1}, \ldots, r_{k,l}, r_{l,t}$ such that the product of all $r_{i,j}$ in this combination is minimized. Initially, this is not solvable directly as each edge would require multiplying by its value (if we were to just use the conversion rate directly). However, if we were to take the negative log of the entire product (for some shortest path), then we could express the *negative log* of the answer in terms of a minimization of sums. Taking the log of every edge would yield a graph who's edges could be added and the result could be derived simply with vanilla Dijkstra's. □

Running Time: $O(|V| \cdot |E|)$.

Justification: Only two operations are performed. First, we construct a graph in linear time with respect to the number of edges (conversion rates). Then we perform Bellman-Ford on the graph. Bellman-Ford dominates the graph construction running time.

(b) **Main Idea:** This is the exact same problem as before except now there can be negative cycles. In order to detect this, we have to simple perform one additional iteration in the Bellman-Ford algorithm.

Pseudocode:

Algorithm 4 Determines if there exists a combination of exchanges in the currency market that yields an increase of money throughout it and ends at the currency that it started at. Assumes G is the graph constructed in (a).

```
1: function FIND_CYCLE(G)
2:     return BELLMAN-FORD_CYCLE_DETECTION(G)
```

Proof: First, note that the fact that the possibility that some cycle of edges yields a number greater than one implies that the negative log of this cycle yields a negative number. This means that this sequence yields a negative cycle in our graph. From this, we know that negative cycle detection is possible from Bellman-Ford, so we are done. □

Running Time: $O(|V| \cdot |E|)$.

Justification: This is mereley Bellman-Ford with the cost of constructing the graph, which is dominated in this case.

5. (20 pts.) Hyperloop construction

Elon Musk has perfected Hyperloop technology, and would like to shorten travel time between Palo Alto and Hawthorne. There is already a network of (two-way) roads $G = (V, E)$ connecting a set of cities V. Each road $e \in E$ has a (nonnegative) travel time $\ell(e)$ needed to cross it.

(a) Mr. Musk has enough cash in his checking account to finance the construction of *one* Hyperloop route, out of a set H of possible routes. Like normal roads, each Hyperloop route h would connect two cities (in the same set V) and have a travel time $\ell(h)$. As Mr. Musk's assistant, you are asked which of these Hyperloop routes, if added to the existing network G, would result in the maximum decrease in the driving time between two fixed Palo Alto (s) and Hawthorne (t). Design an efficient algorithm for this problem.

In particular, the problem is:

Input: an undirected graph $G = (V, E)$, a set of candidate edges H (between the same set of vertices V), with both $e \in E$ and $h \in H$ have nonnegative edge lengths $\ell(\cdot)$, and two vertices s, t

Output: the edge $h_{best} \in H$ such that adding it to G reduces the distance from s to t as much as possible.

Clarifications: There might not be an $s-t$ path in G, but there is definitely an $s-t$ path in $G \cup h$ for at least one $h \in H$. Edges in H may connect the same two vertices as edges in E, and their lengths could be greater or smaller than the parallel already-existing edge.

(b) Mr. Musk is feeling more generous, and is willing to bestow even more Hyperloop routes upon California! Rather than paying for one Hyperloop route, he is now willing to pay to build n of them. The rest of the problem is the same as before: design an efficient algorithm to find the set of n Hyperloop routes that will decrease the driving time between Palo Alto and Hawthorne the most.

Note: It's likely that extending your algorithm from part (a) to this problem will result in an exponentially slow algorithm. When we encounter a problem that has extra features that a standard algorithm (Dijkstra's in this case) cannot handle, our choices include using the standard algorithm as a subroutine in a more complex algorithm; modifying the standard algorithm to take into account the extra features; or modifying the problem instance to incorporate additional information so that the standard algorithm will return the correct result. In this case, we recommend you try the last strategy: construct a graph such that every node also includes information about how many Hyperloop routes have been taken to reach it.

Solution:

(a) **Main Idea:** We perform Dijkstra's twice. First from s and then from t, keeping track of the distances for each independently. Subsequently, we consider each Hyperloop independently, adding one at a time, finding the distance that we decrease the path by, and then removing it before the next one is inserted. We keep track of a running minimum and pick the one Hyperloop with the greatest impact.

Pseudocode:

Algorithm 5 Returns the best Hyperloop to be added such that it reduces the distance from s to t the most.

```
1:  function BEST_HYPERLOOP(G, H, s, t)
2:      dists ← DIJKSTRA(G, s)
3:      dists' ← DIJKSTRA(G, t)
4:      min ← ∞
5:      h_best ← H[0]                          ▷ some hyperloop for starters
6:      for all h = (u, v) ∈ H do
7:          if dists[u] + L(u, v) + dists'[v] < min then
8:              min ← dists[u] + L(u, v) + dists'[v]
9:              h_best = h
10:         if dists[v] + L(u, v) + dists'[u] < min then
11:             min ← dists[v] + L(u, v) + dists'[u]
12:             h_best = h
13:     return h_best
```

Proof: First, it is the case that we have the distance from s to any other vertex as we performed Dijkstra's from s. The same goes for t. Therefore, we know the distance from s to t and the amount that this would decrease byif we were to add some Hyperloop to the graph. We can do this by simply looking to see which vertices the Hyperloop connects. Say the Hyperloop connects two vertices u and v. We can find the distance from s to u, add to that the distance from u to v by finding the weight of the Hyperloop, then add to that the distance from v to t to see the new weight of this path. This value can be compared to the shortest path from before in order to determine the difference in distances (if there is one).

For any graph given, it is not necessarily the case that s and t are connected. Thus, their distance could be any number in $[0,\infty)$. We also know that adding at least one of the Hyperloops will connect them if they are not connected. Therefore, in the case that they are not connected, there will be a solution as the distance decrease will be infinity for at least one Hyperloop.

Alternatively, if they were already connected, it may or may not be the case that adding any one Hyperloop decreases the distance between s and t. Iterating through each Hyperloop and keeping track of the minimum will guarantee that we find a solution. □

Running Time: $O((|V|+|E|)\log|V|)$.

Justification: We perform Dijkstra's twice and then add each Hyperloop once (performing only constant time operations during each insertion). This runtime is linear with respect to H. However, since H is not greater than $|V|+|E|$, this is dominated by the lienar times logarithm time from Dijkstra's.

(b) **Main Idea:** We can say that we have two sets of vertices. For any edge $h \in H$ connecting one set (A) to the other subset (B), h must only go from A to B and not the other way. This implies that for every vertex in A connected to a node in B, this must go through exactly on h. This allows for us to find a shortest path through the graph using only one h. To use n edges, we have to make $k+1$ "layers" of G with directed edges that correspond to the edges in H going from some layer i to some next layer $i+1$. If we only find some l less than n, we add the remaining edges arbitrarily.

Pseudocode:

Algorithm 6 Returns a list of the n best Hyperloops to add to G in order to best reduce the minimum distance from s to t.

1: **function** BEST_N(G, H, s, t, n)
2: $G' \leftarrow$ some directed graph from $n+1$ coping of G with the edges $(u_i, v_{i+1}), (v_i, u_{i+1}) \forall 0 \leq i < n, (u, v) \in H$
3: `dist, prev` $\leftarrow$ DISJKSTRA(G', s_0)
4: find k that minimizes shortest path between s_0 and t_k
5: **return** Hyperloop routes in the shortest path from s_0 and t_k

Proof: Dijkstra's always finds the best path in G'. And since there are $n+1$ "layers" in the constructed graph and the layer i is connected to the next one only in that direction, the shortest path from the zeroeth to the ith layer goes through exactly i edges from H. Therefore, we can go through every path using at most n Hyperloop paths. As always, if none of the edges work in our favor, we just choose some arbitrary Hyperloops to add to our graph. □

Running Time: $O(n \cdot (|V| + |E \cup H|) \log V)$.

Justification: This is similar to Dijkstra's but differs in two respects. The edges set is extended to include H, and there are at most n updates for every iteration.

6. (15 pts.) Proof of correctness for greedy algorithms

This question guides you through writing a proof of correctness for a greedy algorithm. A doctor's office has n customers, labeled $1, 2, \ldots, n$, waiting to be seen. They are all present right now and will wait until the doctor can see them. The doctor can see one customer at a time, and we can predict exactly how much time each customer will need with the doctor: customer i will take $t(i)$ minutes.

(a) We want to minimize the average waiting time (the average of the amount of time each customer waits before they are seen, not counting the time they spend with the doctor). What order should we use? You do not need to justify your answer for this part. (Hint: sort the customers by ____)

(b) Let $x_1, x_2, \ldots, x_n$ denote an ordering of the customers (so we see customer x_1 first, then customer x_2, and so on). Prove that the following modification, if applied to any order, will never increase the average waiting time:

- If $i < j$ and $t(x_i) \geq t(x_j)$, swap customer i with customer j.

(For example, if the order of customers is $3, 1, 4, 2$ and $t(3) \geq t(4)$, then applying this rule with $i = 1$ and $j = 3$ gives us the new order $4, 1, 3, 2$.)

(c) Let u be the ordering of customers you selected in part (a), and x be any other ordering. Prove that the average waiting time of u is no larger than the average waiting time of x—and therefore your answer in part (a) is optimal.

Hint: Let i be the smallest index such that $u_i \neq x_i$. Use what you learned in part (b). Then, use proof by induction (maybe backwards, in the order $i = n, n-1, n-2, \ldots, 1$, or in some other way).

Solution:

Note for the solution to this question. We refer to this summation (the average waiting time):

$$S_n = \frac{1}{n} \sum_{k=1}^{n} (n-k) \cdot t(k). \tag{1}$$

In this summation, k refers to the index of the person *in the order given* and not in any other ordering.

(a) We want to order them by increasing waiting time as this minimizes the summation.

(b) Note that the factor of $n-k$ decreases as you move through the summation. If it were the case that some larger $t(m)$ came earlier in the summation than some smaller $t(l)$, then the summation would have $a = (n-m) \cdot t(m)$ and $b = (n-l) \cdot t(l)$. We know that $(n-m) > (n-l)$ and that $t(m) \geq t(l)$. Thus, if we swap $t(m)$ and $t(l)$, we have $c = (n-m) \cdot t(l)$ and $d = (n-l) \cdot t(m)$. Note that we can directly derive the following expression: $a + b \geq c + d$. But these are the only two values that we change in the summation. Therefore, the value in the summation necessarily doesn't increase but could decrease it.

(c) This is the same argument for (b) but inverted. Simply, swapping any two values in the summation such that we bring the larger term earlier in the summation cannot decrease the time (it can only stay the same or increase).

Let the first element in x that is different than u be x_i. Since this is the first instance of a difference in x, we know that it must be the case that the element in u that is in x_i's place is further into x than x_i, call this element x_j. We know that swapping these in the ordering can never increase the average waiting time, but could potentially decrease it. This means that swapping these values could never increase the average wait time but could potentially decrease it. Let there be m of these swapped values in x. We could extend the argument to each of these pairs without any loss of generality. Therefore, it must necessarily be the case that the ordering found in (a) is an optimal solution.

7. (Bonus 0 pts.) Two-vertex Disjoint Paths

(Note: Only solve this problem if you want an extra challenge. We will not grade this problem.)

Find a polynomial-time algorithm to solve the following problem:

Input: A dag G, and vertices s_1, s_2, t_1, t_2.

Question: Does there exist a path from s_1 to t_1 and a path from s_2 to t_2, such that no vertex appears in both paths?

Your algorithm should have running time $O(|V|^c)$ for some constant c (e.g., $c = 4$ or something like that).