CS180 Midterm

TOTAL POINTS

85 / 100

QUESTION 1

1 Problem 1 **10 / 10**

- **- 0 Correct**

QUESTION 2

2 Problem 2 **20 / 20**

- **- 0 Correct**

QUESTION 3

3 Problem 3 **15 / 20**

- **- 5 Doing BFS without looking every parent nodes**

QUESTION 4

4 Problem 4 **10 / 10**

- **- 0 Correct**

QUESTION 5

5 Problem 5 **15 / 20**

- **- 5 pseudo-poly calls of TSP**

QUESTION 6

6 Problem 6 **15 / 20**

- **- 5 should argue there are 2^p(n) many certificates**

CS180 Spring 2017 - Midterm

Monday, May 1, 2017

You will have 110 minutes to take this exam. This exam is closed-book and closed-notes. There are 6 questions for a total of 100 points. **Please write your name and student ID on every page of your solutions. Please use separate pages for each question.**

Question	Points
1	/10
2	/20
3	/20
4	/10
5	/20
6	/20
Total	/100

1. **[10 points]**

   (a) Prove formally that $2^n = O(n!)$.

   (b) You work for a company and one of your colleagues claims that he (or she) invented a new sorting algorithm whose running time is $f(n) = 16f(\frac{n}{16}) + \frac{1}{2}n$ where n is the size of input to the algorithm and that this algorithm is way better than all existing comparison based sorting algorithm (which is $n \log n$ at best) in terms of asymptotic running time. Is he or she right or wrong? State your answer and prove it formally.

a) $2^n = O(n!)$ → Let's do by induction →

Base case $n=0$ → $2^0 \leq c \cdot (0!)$ ⇒ $1 \leq c \cdot 1$ ✓

Induction step: Let's assume it do hold true for the first n numbers (that is $2^n = O(n!)$) for the n+1st term, we have

$2^{n+1} = 2^n \cdot 2$ compared to $O((n+1)!) = n! \cdot (n+1)$

for any term larger than $n \geq 1$, we see that $n+1 \geq 2$ and we already know by induction hypothesis that $2^n \leq c \cdot n!$

Therefore we have shown by induction that $2^n = O(n!)$.

b) $f(n) = 16 \cdot f(\frac{n}{16}) + \frac{1}{2}n$ → using the master theorem, we can see / looks like $a \cdot f(\frac{n}{b}) + \alpha \cdot n^c$ $k=0$ for Logarithm order

∴ Using master's theorem, $\log_b a = \log_{16} 16 = 1 = C$ so

we have a $f(n) = \Theta(n^{\log_b a})$ ⇒ So $\Theta(n \cdot \log n)$ $k+1 = 0+1 = 1$

Thus, our friend has not really created any better algorithm as the $f(n) = 16f(\frac{n}{16}) + \frac{1}{2}n$ is asymptotically bounded by $n \log n$ just like the best sorting algorithms based on comparisons.

2. **[20 points]** Your millionaire-friend decide to open pizza-parlors on a particular stretch of highway from Los Angeles to Las Vegas, since he knows that there are no pizza restaurants on this deserted highway stretch, and many drivers who love pizza go through it. He wants to open at least one pizza-parlor within 10 mile distance of each gas-station on the highway in order to dominate the inferior food quality offerings at gas stations, and he wants to dominate that particular highway stretch with high quality pizza offerings. Since he heard that you are taking an algorithms class at UCLA, he asks you for an algorithm to places as few pizza-parlors as possible to cover each gas station.

   Explain the algorithm that you would use to decide where on the highway to place pizza restaurants for your rich friend. He says that he will accept your solution only if you could prove to him that it is an absolute minimum number of restaurants to cover all gas stations, so you should prove that as well.

Let's assume we have a highway

where ⊓ is gas & R = restaurant.

A greedy algorithm to minimize costs would be to place a pizza restaurant R_i from west to east starting at 10 miles to the east of the west most gas station that is not covered. For example: suppose gas stations A, B, C, D (west → east) s.t. A & B in 10 mile range and C & D are in their own range. We place 10 miles to east of A to cover both A and B, then 10 miles east of C, then 10 miles east of D, so only 3 restaurants for 4 gas stations.

To prove that this is the best algorithm, we can use an "always stays ahead" argument. Any other algorithm will not put a restaurant 10 miles to east of west most station already covered. Thus, this algorithm will cover for example A in our example from before but will fall short of B. It will need another restaurant for B. Meanwhile our greedy algorithm always schedules so that there is most overlap of gas stations. We have already covered all gas stations to the left of the west most station that is free and can thus keep on greedily including other gas stations with fewer restaurants than other algorithms.

To rephrase the previous argument, let's say that we have some other algorithm that includes gas stations g_1 through g_i with its placement of the resturants, R_1 through R_i. If we assume it to be the best algorithm then it will need another resturant R_{i+1} to cover g_{i+1} which is located beyond the reach of the last resturant (that is R_i only covers till g_i).

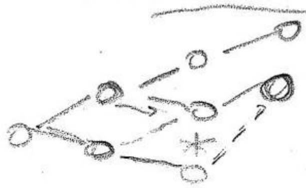
Now say we use our greedy algorithm and in R'_i, we can cover g_{i+1}. This would mean that based on the other algorithm(s), we placed R_i so that g_i was less than the 10 mile distance that each resturant covered ($d_i < 10$) so g_{i+1} was $d_{i+1} \geq 10 + (10 - d_i)$. However in our algorithm, we would place g_i @ $d_i' = 10$ so that the quantity $d'_{i+1} \geq 10 + 0$ and so d'_{i+1} (for g_{i+1}) is also covered by our algorithm. Thus our algorithm "stays ahead" of any other algorithm at all steps of its creation! ✓

3. **[20 points]** Often, there are multiple shortest paths between two nodes of a graph. These shortest paths may share edges between them. That is s → a → b → t and s → a → d → t are two distinct shortest paths, even though they both use the edge s → a. Give a linear-time algorithm for the following problem:

Input: An undirected graph G = (V,E) with unit edge length, nodes s and t.

Output: The number of distinct shortest paths from s to t.

Give a outline of algorithm, its proof of correctness, and the running time analysis.



a) Since we have an undirected graph of unit length edges, we know that BFS can be used to also find shortest path, much like Dijkstra.

There fore, my algorithm is as follows: 1) Maintain array of # of "extra" edges for each node 2) Perform a BFS on the graph G, 3) every time there is an edge that is attached to a previously included node, 4) check that if this edge leads to a vertex in the same layer or the layer above. 5) If same layer, we can ignore it, if one layer above, increment count of our "extra edges" for our array by one. 6) In the end, use the path found by BFS that is shortest, and also add up the counts of the "extra edges" for each node in the path to get the total number of shortest paths in the graph → 1 + count # of edges of all the nodes of shortest path

c) Run time = $O(|V|+|E|)$ for BFS, then $O(|V|)$ to update array, then $O(\log|V|)$ to check which layer the extra edges are attached (using union find as in book). Total = $O(|V|+|E|+|V|+\log V)$ which is $O(|V|+|E|)$ which is linear

$O(|v|) > O(\log|V|)$ so $\log V$ term goes away

b) Proof of correctness: We already know that BFS will give us a shortest path if there are unit length edges. By induction we can show that at each step, if we simply take the subgraph that only includes the layer for the n^{th} node in the path, that is a→b→c→d, n=1, so until layer 3, we are counting the number of ways a node in the previous layer can reach this node. By counting the # of edges that are in two different layers that lead to this node. We assume to be true for n^{th} layer, then for layer (n+1) we do a similar strategy to count the # of ways layer n nodes can reach our node, V_{n+1} in layer n+1, But we also have the different ways to reach nodes in layer n that go to V_{n+1} in layer n+1. Thus we must sum all those different ways and all the ways for layer n-1, n-2, ... till layer 1. Layer 1 can only have one way because that is the start node, s, and we must choose it.

Another version of above proof: Say we know that for a 2 layer graph, there is only one shortest path (since it is the only one from start node to finish node in 2nd layer ... any other path has length > 1). We have counted the number of ways to reach node s (start) = 0 (we must choose it) + the # of ways to choose node t = 1 (the output of BFS). This is our "base step".

Assume it works for n layers where we count the # of ways to reach node t in layer n as the sum of all the paths to reach the nodes in layers above it (i.e. edges between node $V_{n-1} \in$ Layer n-1 to $V_{n-2} \in$ Layer n-2, etc...). (induction hypothesis) Then for the (n+1)st layer, we count the number of ways to reach t_{n+1} from nodes in layer n and add that many # of more of shortest paths. This is exactly what my algorithm is doing by maintaining an array with a count of repeated edges that reach a node from a previous layer!

4. [**10 points**] Your high-school buddy goes to lunch with you one day, and confidentially tells you that he made an amazing discovery: that he can reduce in polynomial time the Minimum Spanning Tree (MST) problem to Traveling Salesman Problem (TSP). That is, MST $\leq_p$ TSP. He plans to write up his solution carefully and send it to the most prestigious journal in computer science, but he does not want to share with you any details of his intricate solution.

Assume that he did not make any mistakes, and proved correctly that MST $\leq_p$ TSP. Do you feel that this result is important enough to be published in prestigious journal in computer science? Explain your answer in detail.

No, this discovery of his is in no way meaningful to our prestigious journal because MST $\leq_p$ TSP means if we have a black box solution to TSP, then we can solve MST in polynomial time using the Black Box. However, we already know that TSP is "harder" than MST and we already know how to solve MST in polynomial time using Kruskal's or Prim's algorithm. Thus the friends "discovery" is meaningless and not to be considered. However if he found TSP $\leq_p$ MST, then he would be on to something (probably a genius!)

To put it another way, while using the solution to a "harder" problem to solve an "easier" problem is the right way to go, in this case, we already know that the "easier" problem can be solved in poly time without the use of this harder problem's solution, so that black box is not useful and we cannot make the claim that the harder problem can be solved in an easier way (which is what the friend probably believes and thus is writing the proof).

5. [**20 points**] Recall the Traveling Salesman problem, TSP:

Input: A matrix of distances D (all distances are positive integers), a budget B.
Output: A tour which passes through all the cities and has length less than or equal to B, if such a tour exists.

The optimization version of this problem asks directly for the shortest tour TSP-OPT.

Input: A matrix of distances D(all distances are positive integers).
Output: The shortest path which passes through all the cities. → not necessarily the buget

Show that if TSP can be solved in polynomial time, then so can TSP-OPT.

If TSP can be solved in polynomial time, then so can TSP-OPT

→ Essentially TSP-OST $\leq_p$ TSP

by the following proof:

Assume we have a blackbox that can solve TSP in polynomial time given a matrix D and some budget B. It outputs yes if a path is there with that Budget, and no if no path with that Budget. We can repeatedly ask this box starting with the matrix and a budget B = sum of all distances. Then we continue to decrement B by one (we know that all distances are integers) and find the point where the output of B_i = yes, but the output of B_{i+1} is a no. Any smaller Budget will also output no (since TSP looks for ≤ B) so that means B_i is our shortest Budget for the ~~most~~ cities, and thus our shortest path.

It will take O(D) to sum up all the distances and then O(S · f(n)) to ask the Black Box. Since O(D) is polynomial, O(f(n)) is polynomial, and O(S · f(n)) is also polynomial, we know that our method to solve TSP-OST (given a way to solve TSP in poly time) is also polynomial.

6. [**20 points**] Show that for any problem Q in NP, there exists an algorithm which solves Q in time $O(2^{p(n)})$, where n is the size of the input and $p(n)$ is a polynomial dependent on input n.

Based on our definition of NP-complete, a solution to any of these NP complete problems can solve any other NP problem. Let's take a known NP complete problem like Vertex Cover and create an algorithm to solve it.

We have, n, nodes and m edges, so in total there are 2^n possibilities (2^n subsets) of vertices to choose from to cover all of the edges in our graph. So, by a brute force method, for each of the 2^n sets, we must see if all the m edges are incident in the vertices of that subset, which is merely $m \cdot 2^n$ → now m can be made into a function of 2^x by $2^x = m$ → $\lg_2 m = x$ → $2^{\log_2 m} = m$ ($e^{\ln t} = t$)

so $2^n \cdot 2^{\log_2 m}$ will give us $2^{n+\log_2 m}$ runtime to solve vertex cover. In this case, $p(n) = n + \log(m)$ which is polynomial dependency, and as we know that an algorithm to solve any NP Complete problem (like Vertex Cover) can be used to solve any NP problem. Thus we have an algorithm $O(2^{n+\log_2 m})$ which can solve $Q \in NP$.