

CS180 Final

Jason Less

TOTAL POINTS

45 / 100

QUESTION 1

1 Problem 1 20 / 25

- **0 pts** Correct. Used greedy algorithms and scan/remove from leafs.
- **0 pts** Correct DP solution
- **0 pts** Correct. Modeled it as bipartite matching or network flow
 - **5 pts** Used Dynamic Programming and gave a recursion. However, the recursion cannot reflect how you make sure that the edges do not share nodes.
- ✓ - **5 pts** Did not indicate how do you construct two disjoint node sets and converted this problem to a bipartite graph matching problem. A simple approach is taking any node as the root and coloring nodes level by level.
 - **10 pts** Pointed out that we first solve the subproblem of the child node and merged the result. However, no correct DP recursion is given or you do not consider the two cases separately that the current root can be taken/not taken.
 - **5 pts** The way that you divide nodes into two sets is not right. A simple approach is taking any node as the root and coloring nodes level by level.
 - **15 pts** Misunderstood of the problem / gave not quite related solution. The correct solution is modeling the tree as a bipartite graph or using dynamic programming.
- **25 pts** No answer
- **20 pts** No clear algorithm
- **10 pts** Used greedy algorithm but not started from leafs. The algorithms cannot always yield correct solution if you just start from a random node.

QUESTION 2

2 Problem 2 10 / 25

- **0 pts** Correct
- **25 pts** It's not NP-complete
- **10 pts** Only answered the feasibility question but did not give an algorithm that returns the assignment
 - **8 pts** Didn't consider the availability of classrooms
- ✓ - **15 pts** Modest attempt. Your algorithm may output no feasible assignment when there is indeed one
 - **3 pts** Did not describe how to assign edge capacity
 - **3 pts** Did not answer when there is or is not a feasible solution. (When max flow equals to what?)
 - **3 pts** Wrong edge capacity design
 - **12 pts** Correct direction but wrong algorithm. You need "perfect matching" but not a "matching"
 - **20 pts** No solution is given for (b) / No related is solution given for (b)
 - + **5 pts** Mentioned that we can use max flow/matching though no algorithm is given
 - **5 pts** It's not NP-complete. The algorithm is fine, though
 - **0 pts** Click here to replace this description.

QUESTION 3

3 Problem 3 3 / 25

- **0 pts** Correct
- **0 pts** (a) full credit
- **10 pts** (a) wrong reduction
- ✓ - **7 pts** (a) Reducing from Hamiltonian cycle or tsp. but no discussion about the length of the cycle. no or wrong discussion about how to set the edge weights and no or wrong discussion about why the constructed edges weights satisfy the triangle inequality.

- **5 pts** (a) the edge weights are set improperly. The triangle inequality is broken.

- **0 pts** (b) full credit
- **3 pts** (b) reduction from euclidean TSP, but fail to explain the edge weight setting correctly.
- ✓ - **5 pts** (b) wrong reduction or fail to explain how to set the edge weights
 - **3 pts** (b) the reduction details are missing or wrong
 - **0 pts** (c) full credit
 - **5 pts** (c) Apply the MST idea with wrong MST details.
 - **8 pts** (c) The polynomial solution is missing or wrong. No explanation why it works.
- ✓ - **10 pts** (c) empty answer or wrong answer
 - **5 pts** (c) confirm it as a NPC and give a MST-like solution
 - **25 pts** Empty answer for three sub-questions
 - **5 pts** (c) The polynomial solution is partially correct

QUESTION 4

4 Problem 4 12 / 25

- **0 pts** a. Correct
- **6 pts** a. wrong recursion (formula)
- ✓ - **8 pts** a. no/irrelevant recursion (formula)
- **10 pts** a. No answer / irrelevant answer
- ✓ - **0 pts** b1. Correct
 - **0 pts** b2. Correct
 - **3 pts** b1. Incorrect argument to avoid $O(v^2 \log v)$ total cost
 - **5 pts** b1. no answer to how to avoid $O(v^2 \log v)$ total cost
 - ✓ - **5 pts** b2. no(wrong) explanation for log N rounds in total
 - **6 pts** b2. algorithm lacks details
 - **7 pts** b2. The algorithm lacks many details and no explanation for log N rounds in total
 - **10 pts** b2. no answer

CS 180: Introduction to Algorithms and Complexity

Midterm Exam

Mar 19, 2019

Name	Jason Less
UID	404-640-158
Section	Shirley Chen

1	2	3	4	Total

- ★ Print your name, UID and section number in the boxes above, and print your name at the top of every page.
- ★ Exams will be scanned and graded in Gradescope. Use Dark pen or pencil. Handwriting should be clear and legible.
- The exam is a closed book exam. You can bring one page cheat sheet.
- There are 4 problems. Each problem is worth 25 points.
- Do not write code using C or some programming language. Use English or clear and simple pseudo-code. Explain the idea of your algorithm and why it works.
- Your answer are supposed to be in a simple and understandable manner. Sloppy answers are expected to receiver fewer points.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.

1. We have seen in class a polynomial time algorithm for maximum matching in bipartite undirected graphs. In general undirected graph, the problem is not NP-complete but the algorithm is quite involved. Suppose we take a tree and ask for maximum matching. Can you give a polynomial time algorithm? If you can, outline the algorithm. [25 pts]

- for finding maximum matching of a bipartite graph: w/ nodes in sets A, B w/ edges E
 - Application of max flow by creating a modified graph G'
 - Create src node S and direct edges from S to all nodes in A
 - Create dst node T and direct edges from all nodes in B to T
 - Also direct all edges in E from A to B
 - Then assign capacity of 1 to all the edges, which gives a valid network-flow prob
 - finding the max matching is equivalent to finding the max flow of G'
which can be done via Ford-Fulkerson Alg

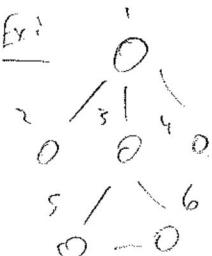
- for finding max matching of a tree: rooted at node T, called graph G

- Can be done by first checking if G is a bipartite graph
 - If so, then just do the above (i.e. transform G into a network-flow problem and find max flow via Ford-Fulkerson, which is a polynomial-time alg)
 - Checking if a graph is bipartite can also be done in polynomial time

- If not a bipartite graph, then max matching can be done by performing a coloring of the tree starting at root T using DFS and backtracking
 - Start by coloring the root a certain color and then DFS on its children to try to color the whole tree

- If you get to a node where coloring a given color b would result in two nodes connected by an edge with the same color, then use an entirely new color to color the given node, and then recursively color the entire tree

- After the coloring, you will be left w/ the nodes of graph G colored a certain way representing k disjoint sets of nodes (where $k = \#$ of colors in G)
 - Divide $\#$ to k disjoint sets s.t. they don't share any nodes
 - Then create n src nodes with edges going from each src node to the sets on the left and m dst nodes w/ edges from each disjoint set on the right going to the m dst nodes
 - Then, create a super src node w/ edges going to all n src nodes and a super dst node w/ edges from each m dst node to the super src node
 - Finally, transform the graph into a network-flow problem and run Ford-Fulkerson to find max flow (i.e. max matching)

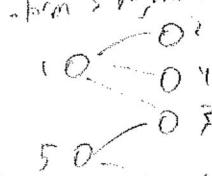


After coloring:

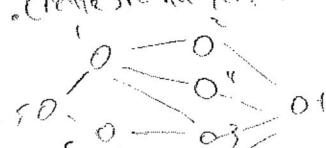


R O - O B

From 3 disjoint sets:



(Create src node, dst node):



• Run Ford-Fulkerson to find max flow

P w/ list of classes, list of intervals
R w/ list of intervals

2. You are given a list of professors. Each professor P_i teaches C_{P_i} different classes, each of an hour, and submits C_{P_i} different hour intervals in which she wants to teach. She is indifferent to what class she teaches in each hour interval which she submitted. On the other hand we have a list of classrooms. H_{R_k} is the list of time intervals when each classroom R_k is available. We want to answer whether it is feasible for all professors' requests to be satisfied, and if it is, output the assignment. This problem is obviously in NP (why?).

- (a) Is the problem NP -complete? [5 pts]
(b) If yes, prove it is NP -complete. If not, give a polynomial time algorithm to answer the feasibility question and output a feasible assignment if there is one. [20 pts]

* The problem is obviously NP because given the input, we can guess whether it is feasible for all professors' requests to be satisfied in polynomial time.

(a) • The problem is not NP -complete

This is because the problem is much easier and there is a known polynomial-time algorithm to solve the problem.

(b) • The feasibility question can be answered using a greedy algorithm

• The problem is unnecessarily wordy and can be greatly simplified by asking that:

• The name of professors and the types of classes that they teach is irrelevant

• Using this claim, Recast the problem amounts to: given a certain number of time intervals and k classrooms (\cup associated availability), can we satisfy all professors' requests

* Ran out of room, so algorithm is rewritten on the backside of this page

Algorithm:

Flatten the time intervals of each professor into a single array intervals, where each elem is a pair of start and time

Sort the intervals based on start time in incr. order

Next, init a min heap sorted based on end time in incr. order

Next, init a hashmap mapping from key = classroom, val: tuple (professor, class, Time interval) called IS

Push the 1st interval (i.e. intervals[0]) to H

for each item in intervals:

- curr = intervals[i]
- interval = H.pop (which is interval finishing earliest)
- if curr starts at or after interval then
 - for each j in classroom in R:
 - for each k in interval in H:
 - if curr can be scheduled at H[k] for R[j] then
 - remove H[k] from R (so we don't replace it)
 - assign val to IS[i][R[j]] = tuple (corresponding prof, corresponding class, time H[k])
 - update interval end time (i.e. interval.end = curr.end)
 - if curr cannot be scheduled then
 - ret false
- else there is a time conflict we need a new room
 - scan rooms the same way as done in if case above to find if curr can be scheduled
 - if not then
 - ret false

push curr to H

push interval (whether updated or not) back to H

ret true and IS (containing the feasible assignment)

$$A = [A, B, C]$$

$$\text{Actions} = [1, 2]$$

$$\text{Atime} = [0-1, 3-4]$$

$$R = [0, \infty]$$

$$\text{Otime} = [0-4]$$

$$\text{Xtime} = [0-3]$$

$$\text{Btimes} = [3]$$

$$\text{Btime} = [2-3]$$

$$\text{Ctimes} = [2-3, 0-1]$$

$$\text{Dtimes} = [0-1]$$

$$\text{Xtime} = [0-1]$$

$$\text{Actions} = [4, 5]$$

$$\text{Atime} = [2-3]$$

$$\text{Ctimes} = [2-3, 3-4]$$

$$\text{Dtimes} = [3-4]$$

$$\text{Xtime} = [2-3]$$

• Num prof. is relevant

• Num classes is irrelevant

• Times: [0-1, 0-1, 2-3, 2-3, 3-4]

• Classes: [[0-1, 2-3, 3-4], [0-1, 2-3]]

at (b) and (c) on backside

3. We have seen in class, by reduction from Hamiltonian cycle, that undirected TSP is NP-complete.

- The Euclidean TSP (call in class mistakenly "planar") is a TSP problem where edge weights in the graph satisfy the triangle inequality ($\forall v_1, v_2, v_3, w\{v_1, v_2\} \leq w\{v_1, v_3\} + w\{v_3, v_2\}$). Prove that the Euclidean TSP is NP-complete. [10 pts]
- We relax the condition on the (non-Euclidean) TSP that each city is to be visited only once. If the saleswoman goes to Chicago through Huston, she can fly back to Huston on the way to Miami (nevertheless, at the end she back to her city). Show that the relaxed-TSP is NP-complete (hint: do not ignore context). [5 pts]
- We are now in the relaxed-TSP, and not only the weights do not satisfy the triangle inequality but also they are terribly skewed with respect to each other. Namely, the weights when ordered from low to high w_1, w_2, \dots satisfy that $w_i > \sum_{j=1}^{j=i-1} w_j$ for all i . Is the relaxed-skewed-TSP problem NP-complete? If not, give a polynomial time algorithm and argue the correctness of your algorithm. [10 pts]

(a) Hamiltonian Cycle (HC) \Leftrightarrow p undirected TSP

• The undirected TSP asks that given a graph $G = (V, E)$ and a weight W , does there exist a traveling salesman tour s.t. the total length of its edges $\leq W$.

• Can be shown by a reduction from HC:

• for a given weighted graph $G' = (V', E')$ w/ non-negative weights and W , the TSP is to find whether G' contains a simple cycle of length $\leq W$ passing through all the vertices

• to reduce the HC problem to the undirected-TSP for graph G , complete the graph G by adding edges between all pairs of vertices that were not connected in G

• the new graph $G = (V, E)$ where $V = V'$ and $E = \{(u, v) \mid \text{for any } u, v \in V'\}$

• for edges in G' also present in G , assign weight 0

• for other edges assign weight 1

• Construction of the complementary graph can be done in polynomial time

• The graph G has a HC iff there exists a cycle in G' passing through all vertices exactly once

• The graph G has a HC iff there exists a cycle in G' passing through all vertices exactly once and has length ≤ 0 (i.e. solution for instance of undirected-TSP with $W=0$)

• If so, then the cycle contains only edges originally present in G (as all new edges in G' have weight 1 and hence, can't be part of a cycle w/ length ≤ 0)

• Hence, there exists a HC in G

• If there exists no HC in G , then it forms a cycle in G' with length > 0 , since the weight of all edges is 0

• Hence, there exists a solution for undirected-TSP in G' w/ length ≤ 0

• Euclidean TSP can be shown to be NP-complete by first showing it is NP and then reducing Hamiltonian Cycle to it

• It is NP as we can guess such a traveling salesman tour exist s.t. total length of edges $\leq W$

• Can reduce from Hamiltonian Cycle by constructing the complementary graph similar to the proof shown above, but instead the edges added to G' are between all pairs of vertices (v_i, v_j) not in connected in G s.t. the edges added are (v_1, v_3) and (v_3, v_2)

• Then, following the same process as shown above, if there exists no HC in G , then it forms a cycle in G' w/ length > 0 , so there exists a solution for Euclidean TSP in G' w/ length ≤ 0

- (b) • Relaxed-TSP can be shown to be NP-complete using the same proof shown in part 1 of (a), but relaxing the constraint that the simple cycle in G' can now visit vertices more than once (instead of just being able to visit them exactly once)
- This is a modified reduction from Hamiltonian Cycle
- (c) • The relaxed-Shuffled-TSP can be shown to be NP-complete by using the same proof in part (b), which is a reduction from Hamiltonian Cycle.
- The fact that the weights are shuffled has no bearing on the reduction from Hamiltonian Cycle as we have the freedom when doing an NP-complete reduction to use the value of b to our advantage (and solving an instance where $b = 0$)
 - In this way, the proof previously shown assign weights only of 0 or 1 depending on if A is originally present in G .
 - Therefore, the weights being shuffled doesn't effect the overall length of the simple cycle in G' w/ val = 0 iff there exists a TIC in original graph G .

* (a) is on backside of page

4. (a) In class we have seen the Bellman-Ford algorithm for one-to-all shortest paths with negative edges but no negative cycle. Write a recursion for shortest paths problem such that you can argue the recursion is amenable to dynamic-programming. And argue that the Bellman-Ford algorithm is in fact an iterative implementation of your recursion (Do not confuse Bellman-Ford with Floyd-Warshall which is all-to-all shortest paths algorithm). [10 pts]

- (b) We said in class that the "idea" of an algorithm is manifested in its recursion.

Here's a recursion to solve MST: For each node v find the min weight edge adjacent to it. These chosen edges create a forest (why no cycle?). We take these edges to be in the MST. Now we "contract" all nodes incident to the same tree into a single "new node", which is connected to other "nodes" by original edges that connect a node in one tree to a node in another tree. All the intra-tree edges (edges aside from the tree edges that connect nodes in the same tree) are "gone." Notice that this might create "parallel edges" but that is ok.

We want to implement this recursion into an $O(|E| \log |V|)$ time algorithm. The implementation that Prof. Gafni knows requires that for each node the edges around it are ordered by weights. Alas, this looks like resulting in a cost of $O(|V|^2 \log |V|)$ algorithm which is larger than $O(|E| \log |V|)$ for a sparse graph.

- Help get Prof. Gafni out of this conundrum. [5 pts]
- Outline an algorithm and argue it achieves the desired complexity (To find whether an edge is inter or intra tree its better be that all nodes in the same tree in the recursion are named the same. You want to argue that throughout the algorithm a node changes name at most $\log |V|$ time. Recall Union-find.) [10 pts]

- (b)
- Prof. Gafni is doing the sorting of edges in the wrong place, which is resulting in the $|V|^2 \log |V|$ time complexity.
 - Instead, sort all of the edges in increasing order before entering any for-loop, which will achieve the desired time complexity.
 - Also, checking whether two nodes already belong to the tree can be done using union and find of the union-find data structure, which allows the desired time complexity to be achieved.

- Algorithm to solve this is Kruskal's, which utilizes union-find

minMst(E):

sort edges of E by increasing weight

mstWt = 0

create disjoint set

for each (u, v) in E :

if u and v are in different trees then

mstWt += wt(u, v)

union(u, v)

This is $O(|E| \log |V|)$ - if sort is outside of any for-loop and

the loop runs for all edges $|E|$, where each iteration does $\log |V|$ work via the union operation

(a) Algorithm:

init dist of all v to ∞

create empty min heap pq where each item = (wt, vertex)

insert $(0, s)$ to pq

while pq is not empty:

extract min dist vertex u

for each edge (u,v) incident to u :

if there is a shorter path to v through u (i.e. $\text{dist}[u] + \text{wt}(u,v) < \text{dist}[v]$) then

update dist of v (i.e. $\text{dist}[v] = \text{dist}[u] + \text{wt}(u,v)$)

insert v to pq