CS 180: Introduction to Algorithms and Complexity
Midterm Exam

Mar 19, 2019

Name	Cody Swain
UID	205189694
Section	

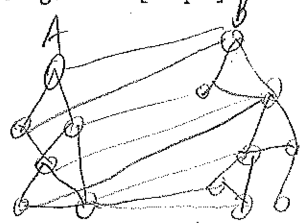
1	2	3	4	Total

★ **Print your name, UID and section number in the boxes above, and print your name at the top of every page.**

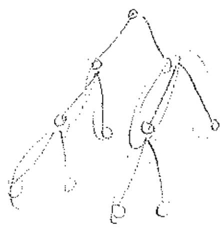
★ **Exams will be scanned and graded in Gradescope. Use Dark pen or pencil. Handwriting should be clear and legible.**

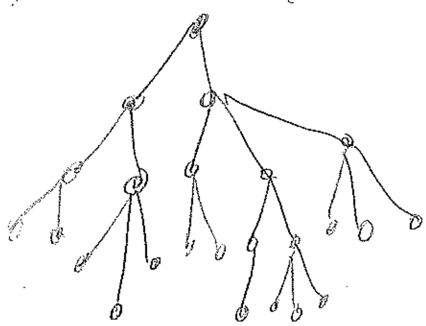
- The exam is a closed book exam. You can bring one page cheat sheet.
- There are 4 problems. Each problem is worth 25 points.
- Do not write code using C or some programming language. Use English or clear and simple pseudo-code. Explain the idea of your algorithm and why it works.
- Your answer are 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]

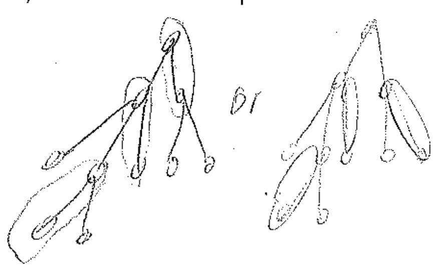


Since I am not completely clear on the professors definition of a "maximum matching", since we didn't have hw problems on matching (and we skipped the stable matching chapter), I will assume this is the maximum number of edges connecting A, B where no node for these edges $e_i \in E$ $e_i = \{s_i, t_i\}$ is repeated in set of maximum matching.

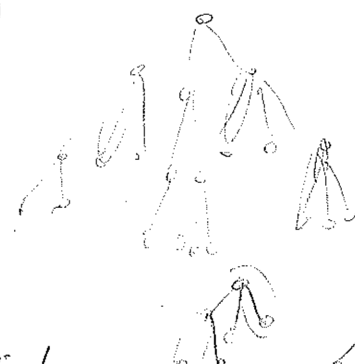


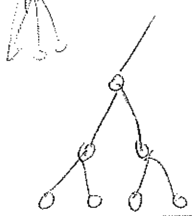


Simple Case → Simple Tree



Maximum matching = 3





I believe that the number of matchings in a tree may be acquired in polynomial time by using a divide and conquer algorithmic paradigm. By looking at each child node and recursively counting matchings in smaller trees where the child is the root, the maximum number of matchings may be found. The base case of the divide and conquer algorithm would have one matching, and it should color its parent node so that no other node may match with it. Upon reaching the leaf nodes, a bottom up counting of every other node with multiple children should give the ~~total number~~ ~~max number of~~ max matchings

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]

from how this question is worded, I am assuming each hour interval is one hour.

List of professors $P = \{P_1, P_2, P_3 \dots P_n\}$

Classroom list $H_{R_k} = \{R_1, R_2, R_3\}$

$C_1 = \{C_{11}, C_{12}, C_{13}, \dots C_{1m}\}$ $C_2 = \{\dots\}$ $C_3 = \{\dots\}$ $C_n = \{\dots\}$

$\{S_1, e_1\}$ $\{S_2, e_2\}$ $\{\dots\}$ availability

Hour intervals $H = \{(S_1, e_1), (S_2, e_2), \dots\}$

To begin to check if all professors' requests may be satisfied, a brute force solution involves iterating through every hour interval for each professor, and checking that it falls into a range of an open classroom. If it does,

P_1 → 3:00-4:00, 12:00-1:00

P_2

P_3

a.) I think this problem is not NP-complete

b.) For each hour interval, iterate over classroom intervals and find match if H_{R_k} start < C_{P_i} start and H_{R_k} end > C_{P_i} end. If this is the case, create 1 or two new sets from H_{R_k}. Either H_{R_k} start to C_P start and C_P end to H_{R_k} end or the first half or last half. A set here is really just a pair of start and end time. If an hour interval (also a pair of start and end time) doesn't fall into remaining set, then the algorithm returns false. To obtain a feasible assignment, create a hash map for each class C_i and the pair of start and finished times, as they are removed from the total set of classroom times.

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

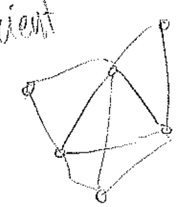
   (a) The Euclidean TSP (call in class mistakenly "planar") is a TSP problem where edge weights in the graph satisfy the triangle inequality ($\forall\ v_1, v_2, v_3,\ w\{v_1, v_2\} \leq w\{v_1, v_3\} + w\{v_3, v_2\}$). Prove that the Euclidean TSP is *NP*-complete. [10 pts]

   (b) 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]

   (c) 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 $w_1, w_2, \ldots$ 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.) Euclidean TSP is still undirected and may be reduced to another NP-complete problem. Since it is given that TSP is NP-complete, showing that Euclidean TSP has same properties should be sufficient

Euclidean TSP $\leq_p$ TSP
TSP $\leq_p$ Euclidean TSP
Euclidean $\equiv$ TSP



This Euclidean constraint doesn't reduce the problem space to polynomial time solvable. All it does is remove extra edges / define shortest path to always be optimal but the constraint that TSalesman must visit each city once is what creates the NP-Complete solution space ∴ is NP-complete

b.) This can be reduced to Hamiltonian cycle by splitting nodes which are visited multiple times

c.) This relaxed skewed problem is no longer NP-complete

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-Warshal 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]

a.) Bellman-ford essentially iterates n-1 times, and reconnects the shortest path length for each node during each iteration. A dynamic programming algorithm can be built which operates in a similar way because during Bellman-ford, the shortest path to a vertex $v \in V$, always consists of the shortest path to an adjacent vertex to v. It follows logically that this problem is amenable to DP because in DP you store the shortest paths to all nodes in a ground up manner.

Bellman Ford is 1→all

edge weight ↓ $e = \{u, v\}$

$$OPT[V_i] = \begin{cases} \min\limits_{n_i \in \text{Adjacent Nodes}} (OPT[n_i] + \text{Weight}(V_i, n_i)) \end{cases}$$

* To avoid endless loop you do however need to have some array which tracks adjacent nodes which have been analyzed
*Essentially coloring

Bellman-ford is essentially an iterative version because you essentially go through the permutations of how adjacent edge updates will eventually lead to shortest path to any node

b.) You won't have a cycle because nodes that would create a cycle each have shared min weight edge. Instead of running a sort for each node, you could initially run n log(n) sort then use the union find datastructure to track and store adjacent edges which may then be used in the "contraction". This should achieve desired complexity.