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.

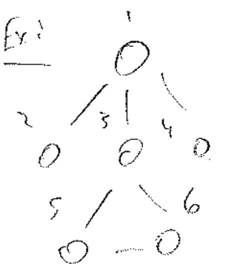
Name: UID: CS180 Final Exam

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]

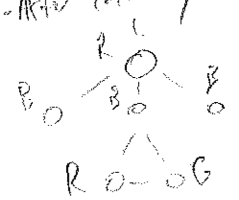
- for finding maximum matching of a bipartite graph: w/ nodes in sets A, B w/ edges E
  - Application of max flow by creating a modified graph G'
  - Create src node S and direct edges from S to all nodes in A
  - Create dst node T and direct edges from all nodes in B to T
  - Also direct all edges in 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 k 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 = the # of colors to color G)
  - Divide the 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 n dst nodes w/ edges from each disjoint set on the right going to the n 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 n 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)

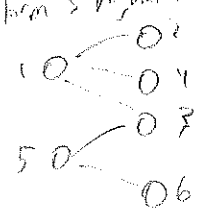
Ex:



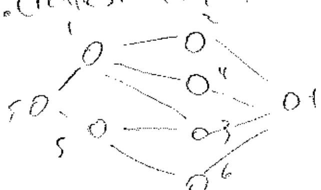
After coloring:



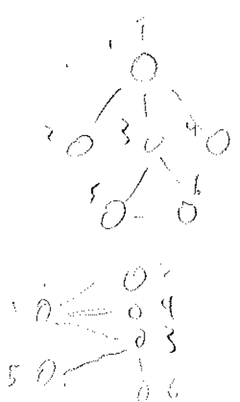
From 3 disjoint sets:

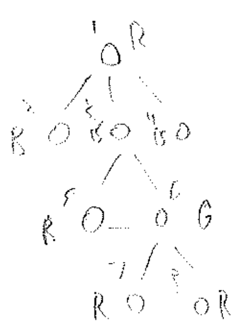


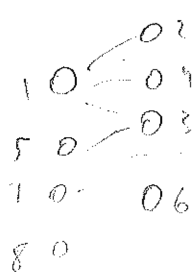
Create src node, dst node:

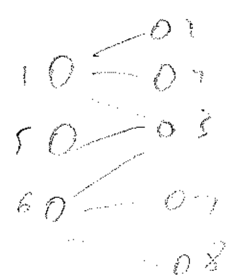


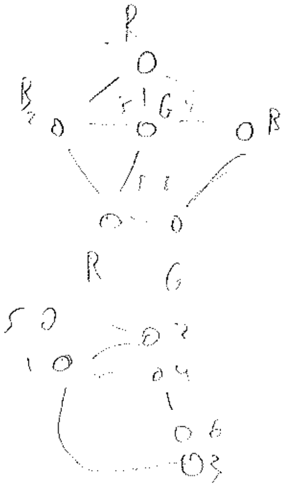
Run Ford-Fulkerson to find max flow











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 noting that:

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

• Using this claim, then the problem amounts to: given a certain number of time intervals and R classrooms (w/ 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 end times
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 rs
Push the 1st interval (i.e. intervals[0] to PQ)

for i=1 to n interval in intervals:
    curr = interval[i]
    interval = PQ.top (which is interval finishing earliest)
    if curr starts at or after interval then
        for each j=1 to n classroom in R:
            for each k=1 to n interval in TR:
                if curr can be scheduled at TR for R then
                    remove TR from R (so we don't reprocess it)
                    assign val to rs => rs[R[j]] = tuple (corresponding prof P, corresponding class C, time TR[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 PQ
    push interval (whether updated or not) back to PQ
ret true and rs (containing the feasible assignment)

P = [A, B, C]
Aclasses = [1, 2]
Atime = [0-1, 3-4]

Bclasses = [3]
Btime = [2-3]

Cclasses = [4, C]
Ctime = [2-3, 0-1]

=>

- Num prof. is irrelevant
- 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]]

R = [σ, α]

σ_{time} : [0-4]

α_{time} : [0-3]

	1	3	2
σ =	0-1	2-3	3-4
	4	5	
α =	0-1	2-3	

* (b) and (c) on backside

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) Hamiltonian Cycle (HC) $\leq_p$ 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 k', the TSP is to find whether G' contains a simple cycle of length $\leq k$ 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
  - edges between all pairs of vertices that were not connected in G $E' = \{(u,v)\}$ for any u,v elem of V'
    - the new graph $G'=(V',E')$ where $V'=V$ and $E'=\{(u,v)\}$ for any u,v elem of 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 and has length ≤ 0 (i.e. solution for instance of undirected-TSP with $k=0$)
  - If so, then the cycle contains only edges originally present in G (as the 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 an 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 exists 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_1,v_2) 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 an HC in G, then it forms a cycle in G' w/ length $=0$, so there exists a solution to 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

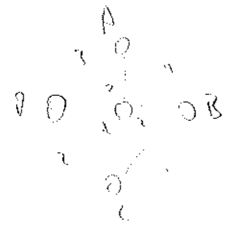
(c) • The Relaxed-Skewed-TSP can be shown to be NP-complete by using the same proof in part (b), which is a reduction from Hamiltonian Cycle

• The fact that the weights are skewed 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 k to our advantage (and solving an instance where k = 0)

- In this way, the proof previously shown assign weights only of 0 or 1 depending on if it is originally present in G
- Therefore, the weights being skewed doesn't affect the overall length of the simple cycle in G' w/ val = 0 iff there exists a HC in original graph G

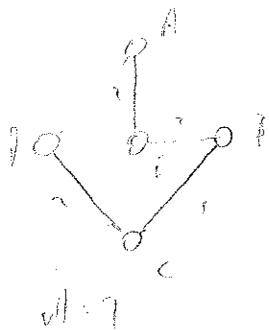
* (a) is on backside of page

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-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]



(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 incr. 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

```
minWst(edges E):
    sort edges of E by incr. wt
    mstWt = 0
    create Disjoint Set
    for each (u,v) in E:
        if u and v are in diff trees then
            mstWt += wt(u,v)
            union(u,v)
```

• This is $O(|E|\log|V|)$ as sort is outside of any for-loop and the loop runs for all edges $|E|$, where each iter does $\log|V|$ work via the union operation

(a) Algorithm:

init dist of all v to inf
create empty min heap pq where each item = (wt, vertex)
insert (0, src) 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. dist[u] + wt(u,v) < dist[v]) then
            update dist of v (i.e. dist[v] = dist[u] + wt(u,v))
            insert v to pq