

Feel free to work in groups of at most 4 for these - if you have a group of more than 4, please run it by me first. If you do work in a group, please include the names of those that you worked with. **However: each student should submit a separate copy where the solutions have been written by yourself.**

1. (10 points) A certain table tennis league is organized so that each season the n players play against each other in a round-robin format, so that each player plays against each other player. The player who wins the most games is declared the winner: ties are handled by tossing a coin. A certain player has come to us mid season, worrying that they may have been technically eliminated. Our goal is to determine in a polynomial amount of time whether they can still win or not. Show that you can solve this problem with a runtime that is polynomial in n . Specifically, I want a polynomial time reduction to the flow decision problem. With even more clarity:

ELIMINATION GAME (EG)

Input: Players $P_1, \dots, P_n$, where player P_i has won w_i games so far, and remaining games $G_1, \dots, G_m$.

Output: TRUE if P_n can possibly finish with the most wins out of $P_1, \dots, P_n$.

FLOW

Input: A graph G with special vertices s and t , capacities c_e for every edge e , and a number k .

Output: TRUE if there is a valid flow on G with value k and FALSE otherwise.

For full credit for this problem, I'm asking you to show: $\text{EG} \leq_p \text{FLOW}$.

As an example, suppose this league has 8 players $P_1, \dots, P_8$, where we are trying to determine if P_8 has been eliminated. We are given that the only game left to be played is between P_6 and P_7 . We are also given that P_6 , P_7 , and P_8 each have 5 wins in total. Then no matter the results of the remaining games (note that every game individually cannot end in a tie), P_8 cannot be the winner of the league.

Solution: Let g be the number of games remaining that P_n is involved in and let $W = w_i + g$. Then W is the maximal number of wins that P_n can finish the season with. Given an instance of EG, we create an equivalent instance of FLOW as follows: create special vertices s and t , and create vertices $u_1, \dots, u_m$ for every game remaining and vertices $v_1, \dots, v_{n-1}$ for every player other than P_n . We create edges between s and each game vertex with capacity 1, between each player vertex v_j and t with capacity $W - w_j$. Lastly, we create a pair of edges from the game vertex u_i to the two player vertices playing in G_i . Lastly, we set $k = m$. Clearly this reduction can be computed in $O(n \times m)$, so it is a polytime reduction.

If there is a way for P_n to win the league, then there must be a way to assign victors to all the remaining games so that P_n is either tied with or has more victories than all the other players. Note that we can assume in this solution that P_n wins in each game they play, since this only increases the number of wins P_n would have. Hence, every

player P_i has won at most W games in this scenario, which means they have won at most $W - w_i$ games out of the remaining games they are involved in. Hence we can create a flow of value m by pushing a unit of flow from s to u_i to the winner of G_i to t for each i : the resulting flow will push exactly one unit of flow per game, and no player vertex v_j will have more units of flow than $W - w_j$. Hence we have created a valid flow of value m .

If there is a valid flow of value m , then there is a way to assign flow to each remaining game so that no player P_i has more flow than $W - w_i$, so by assigning victors to each game according to which player it sends flow to, we have that no player wins more than W games in total, so there is a way to assign victors over all the remaining games so that P_n is either tied with or has more wins than every other player. Thus P_n can still win their league.

2. (10 points) Google has a list $A_1, \dots, A_n$ of advertisers that want to show ads to a group of users $u_1, \dots, u_m$. Specifically, each advertiser A_i has a list of users G_i (a subcollection of $u_1, \dots, u_m$) that it would like to show ads to (these groups may overlap); however, different advertisers have purchased different plans from Google, so Google will only show r_i users the advertisement from A_i . Additionally, Google has run some analytics over their userbase and knows for each u_j the number c_j of ads u_j can see before u_j will get fed up and install an adblock. Your goal is to determine whether Google can display ads so that r_i of advertiser A_i 's ads get shown to users in G_i , no user u_j gets more than c_j ads, and no user gets shown the same ad multiple times.

ADVERTISEMENT PROBLEM (ADS)

Input: Advertisers $A_1, \dots, A_n$ and users $u_1, \dots, u_m$, groups of users $G_1, \dots, G_n$, numbers $r_1, \dots, r_n$ and $c_1, \dots, c_m$

Output: TRUE if there is a way to show users ads so that each advertiser has its ads shown to r_i users in G_i and no user u_j gets shown more than 1 ad from any particular advertiser and no more than c_j ads in total and FALSE otherwise.

Specifically, I'm asking you to show that $\text{ADS} \leq_p \text{FLOW}$.

Solution: Given an instance to ADS, create an instance of FLOW as follows: create special vertices s and t , vertices $x_1, \dots, x_n$ for each advertiser, and vertices $y_1, \dots, y_m$ for each user. Then we create edges from s to each x_i with capacity r_i , from each y_j to t with capacity c_j , and from each x_i to all y_j such that user u_j is in group G_i . We set $k = r_1 + \dots + r_m$. As in 1., this reduction takes $O(n \times m)$ time to create and is thus polytime.

If there is a way to show users ads so that each advertiser has its ads shown to r_i users in G_i and no user u_j gets shown more than 1 ad from any particular advertiser and no more than c_j ads in total, then push a unit of flow along the path $s \rightarrow x_i \rightarrow y_j \rightarrow t$ for each advertiser A_i and user u_j that they show an ad to. Then no capacity constraints are broken since we've chosen edge capacities corresponding to the rules of the ADS problem, so we have a flow of size $r_1 + r_2 + \dots + r_m = k$.

On the other hand, if there is a flow of size k in the graph, then by sending an ad of A_i 's to u_j if there is a unit of flow along the edge between x_i and y_j in the graph. Again, since we've set up capacities in the graph to correspond to the constraints of the ADS problem, we know that this ads assignment guarantees that we show users ads so that each advertiser has its ads shown to r_i users in G_i and no user u_j gets shown more than 1 ad from any particular advertiser and no more than c_j ads in total.

3. I've mentioned a few times how decision problems are easier to conceptualize for reduction type problems rather than optimization, but it's good to check along the way that these problems really are the "same", at least in terms of difficulty. Let's check for the Clique problem:

CLIQUE DECISION (CD)

Input: A graph G and number k

Output: TRUE if there are k vertices in G that are all adjacent to each other and FALSE otherwise.

CLIQUE OPTIMIZATION (CO)

Input: A graph G

Output: The largest collection of vertices in G that are all adjacent to each other.

Show that

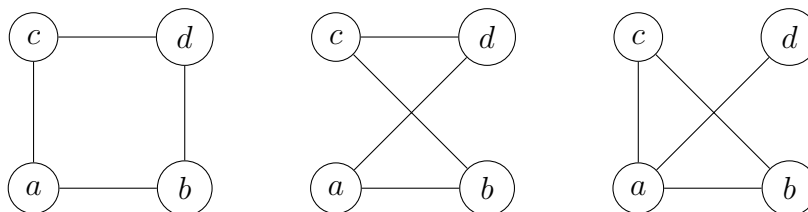
- (a) (5 points) if you have a polynomial time algorithm for CO then you can design a polynomial time algorithm for CD

Solution: To solve CD, we run the algorithm CO and look at the vertices returned. If there are at least k vertices, then there are K vertices in the graph that are all pairwise adjacent. Otherwise, there cannot be k vertices in the graph that are all pairwise adjacent (or they would have been returned instead).

- (b) (10 points) if you have a polynomial time algorithm for CD then you can design a polynomial time algorithm for CO

Solution: Let the runtime of the algorithm A for CD be $O(n^c)$. Then to solve CO, we first determine the correct size of the clique by running A on the graph with $k = 1, 2, \dots$ until we reach a value k where there is a clique of size k but there is no clique of size $k + 1$. Then the maximal clique size must be size k . This will take $O(n \times n^c) = O(n^{c+1})$ time, which is still polynomial. Next, to determine the clique itself, pick an arbitrary ordering $v_1, v_2, \dots, v_n$ of the vertices and initialize the clique set to be returned as $C = \emptyset$. Iterate through $i = 1, 2, \dots, n$ with the following rule: let $G \setminus v_i$ denote the graph G without vertex v_i . If there is still a clique of size k in $G \setminus v_i$ (which we can determine by running A on $G \setminus v_i$), reset $G = G \setminus v_i$. Otherwise, we know that v_i must be in the clique, so we add it back into G . After processing v_n , we will have that the only vertices remaining in G are a clique of size k .

4. Often times we look at problems over graphs which we assume are either directed or undirected, but maybe we start wondering about which of these types of graphs are harder to deal with. Consider the following graph isomorphism problem: we are given two graphs, and we want to see if one is simply a relabeling of the other. For example, consider the following graphs:



The first two are just relabellings of the vertices, but there is no way to relabel the vertices of the last one to become the same as the first two.

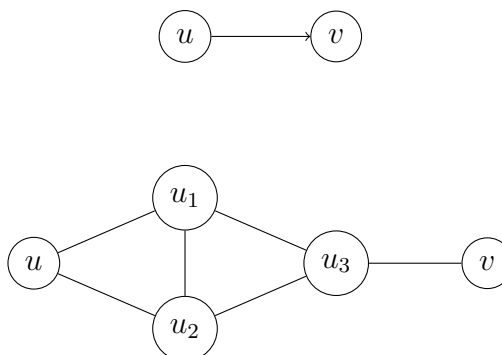
Let n be the number of vertices and m be the number of edges.

- (a) (5 points) Show that if you have a $O(n + m)$ time algorithm for the graph isomorphism problem for directed graphs, then you can design a $O(n + m)$ time algorithm for the graph isomorphism problem on undirected graphs.

Solution: Given an undirected graph, we can replace every edge with a pair of directed edges (one for each direction) to obtain precisely the same graph, just written with directed edges. Hence the pair of directed graphs are isomorphic exactly when the original graphs were, so we can just run the algorithm for the directed graphs. The number of edges m has doubled, but this will still be $O(n + m)$.

- (b) (10 points) Show that if you have a $O(n + m)$ time algorithm for the graph isomorphism problem for undirected graphs, then you can design a $O(n + m)$ time algorithm for the graph isomorphism problem on directed graphs.

Solution: For each directed edge, replace it with the following subgraph:



The idea here was to encode the direction of edges using a small number of new vertices and edges: the total number of new vertices in the new graph will be $n + 3m$, and the total number of new edges in the new graph will be $6m$. Hence, by running the undirected graph algorithm on the new inputs and returning the output, we obtain a $O(n + 3m + 6m) = O(n + 9m) = O(n + m)$.

To see that this works, note that clearly if the original graphs are isomorphic then the new ones also are. Let A and B be the original two graphs and let A' and B' be the transformed graphs. It's a little more tricky to see that if the new graphs are isomorphic, then the old ones are too: if the new ones are isomorphic, then there is some relabelling of the vertices in the new graphs so that under the relabelling the graphs are the same. To get the relabelling of the original graph, we just take the relabelling of the vertices that were copied over (ignoring the new ones). To see that this relabeling works, note that if there is an edge between two vertices in A , then there is a subgraph between those same two vertices in A' , and so there must be the same subgraph between the two vertices in the relabelling of B' , so there is an edge between the two end points of the subgraph in B .