CS 255: Homework 3

Due Friday May 3 at 11:59 pm

Instructions:

1. You may work in groups of up to 3 students.
2. Each group makes one submission on Canvas.
3. Include the name and SJSU ids of all students in the group.
4. You may discuss high level concepts with other groups, but do not copy other's work.

Problem 1

The Autocratic Party is gearing up their fund raising campaign for the 2024 election. Party leaders have already chosen their candidates for president and vice-president, as well as various governors, senators, representatives, judges, city council members, etc. For each candidate, the party leaders have determined how much money they must spend on that candidate's campaign to guarantee their election.

The party is soliciting donations from each of its members. Each voter has declared the total amount of money they are willing to give each candidate between now and the election. (Each voter pledges different amounts to different candidates. For example, everyone is happy to donate to the presidential candidate, but most voters in San Jose will not donate anything to the candidate for Trash Commissioner of Chicago). Federal election law limits each person's total political contributions to $100 per day.

Describe and analyze an algorithm to compute the donation schedule, describing how much money each voter should send to each candidate on each day, that guarantees that every candidate gets enough money to win their election. The schedule must obey federal laws and individual voters' budget constraints. If no such schedule exists, your algorithm should report that fact.

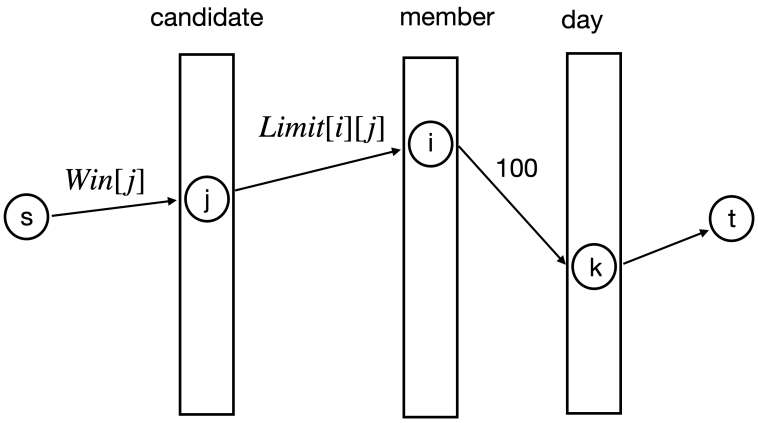
Assume that there are n candidates, p party members, and d days until the election. The input to your algorithm is a pair of arrays $Win[1 \ldots n]$ and $Limit[1 \ldots p][1 \ldots n]$, where $Win[i]$ is the amount of money candidate i needs to win, and $Limit[i][j]$ is the total amount of party member i is willing to donate to candidate j.

Your algorithm should return an array $Donate[1 \ldots p][1 \ldots n][1 \ldots d]$, where $Donate[i][j][k]$ is the amount of money party member i should donate to candidate j on day k.

Solution: There are three discrete sets that interact with each other: candidates, party members, and days. Specifically, there are two types of connections:

- For each member-candidate pair (i, j), $Limit[i][j]$ gives the total amount member i is willing to give to candidate j. We add an edge connecting i and j with capacity $Limit[i][j]$.
- Each member can donate no more than $100 dollars per day. We make edge between member i and day k with capacity 100.

We can organize these sets in the order: candidate, member, day. Since candidate j needs $Win[j]$ total donations to win, we create a edge from the source s to candidate j with capacity $Win[j]$. We also create edges connecting each day k to the sink t with capacity ∞. Refer to the figure below for an illustration of the construction.



Any feasible flow on this graph can be decomposed in to s-t paths where each path defines a member, candidate, day tuple (i, j, k). The minimum flow over any edge in a given path can be interpreted as the amount member i donates to candidate j on day k. The capacity constraints between candidates, members, and days ensure that donations obey members' daily contribution limits, and member limits on contributions to each candidate. If the flow saturates the edges leaving the source, then there is a feasible donation schedule which gives each candidate the total amount of money needed to win their election.

We find a donation schedule by computing a max-flow and decomposing the flow into paths as described above. The total number of vertices in this network is $n+p+d+2$. There are $n + np + pd + d$ edges. We can find a max-flow in the network in $O(VE) = O\big((n + p + d)(n + np + pd + d)\big) = O((n + p + d)(np + pd))$ time. Flow decomposition also takes $O(VE)$ time.

Problem 2

Suppose you are given a magic black box that can determine in polynomial-time, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a polynomial-time algorithm that either computes a proper 3-coloring of a G or correctly reports that no such coloring exists, using the magic black box as a subroutine. [Hint: The input to the magic black box is a graph. Only a graph. Vertices and edges. Nothing else.]

Solution. Intuitively, we want to process the vertices one by one: check if any of the three colors allows for a valid 3 coloring, fix a color for the current vertex, and move on to the remaining vertices. Unfortunately, the black box doesn't allow us to directly specify the color of any given vertex. However, we can achieve this effect using a coloring gadget similar to the truth gadget in the reduction from 3SAT to 3Color. The color gadget consists a complete subgraph of three vertices (each vertex is connected to the other two.) This ensures that in

any valid 3 coloring, the three vertices of the color gadget are assigned distinct colors. We will call these vertices Red, Green, and Blue.

Observe that we can 'color' a vertex v by adding two edges from v to the vertices of the color gadget. For example, to color vertex v red, we add edges from Green to v and from Blue to v. Since v connects the Green and Blue vertices, it must be colored red in any valid 3 coloring. This leads us to the following algorithm to compute a 3 coloring.

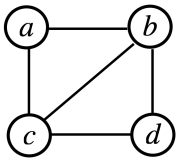
- Check if G is 3 colorable. If false, return 'not 3 colorable'.
- For each vertex $v \in V$, check which of the three colors allows for a valid 3 coloring. Fix the color of v by keeping the edges from v to the color gadget.
- Repeat until all vertices are colored.

In each call to the black box, the graph contains $|V| + 3$ vertices. For each vertex $v \in V$, we color v by adding two edges to G. Thus, at most $|E| + 2|V|$ edges in any call to the black box. Since the inputs are polynomial in the size of G, each call to the black box runs in polynomial time. Further, we make three calls to the black box for each vertex (really we only need 2). Therefore, the algorithm runs in polynomial time.

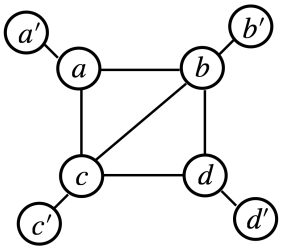
Problem 3

(a). A subset S of vertices in an undirected graph G is *half-independent* if each vertex in S is adjacent to at most *one* other vertex in S. Prove that finding the size of the largest half-independent set of vertices in a given undirected graph is NP-hard.

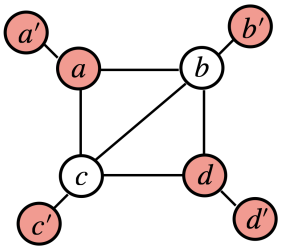
Solution: We give a reduction from IndependentSet. Given an input graph $G = (V, E)$ for an arbitrary instance of IndependentSet, we create a new graph $H = (V', E')$ by adding new vertices and edges to G. For each vertex $v \in V$, we create a new vertex v' and add an edge (v, v'). The new vertex does not connect to anything else. This step clearly takes polynomial-time. We call the vertices of G real, and the newly created ones copy vertices. Let C denote the set of copy vertices.



(a) Input graph G



(b) The graph H.



(c) A max half-independent set in H

It is easy verify that for any independent set I in G, the set $I \cup C$ is half-independent in H. The following claim shows the correctness of the reduction.

Claim: If S is a maximum half-independent set of H, then $S = I^* \cup C$, where I^* is a maximum independent set of G.

Proof. Let S be a maximum half-independent set of H. First, we show that the set of real vertices of S form an independent set. For contradiction, suppose S contains two adjacent real vertices u and v (that is, $(u, v) \in E$). Notice that S does not contain either of the copy vertices u' or v', otherwise u and v are adjacent to two vertices in S. If we remove *one* of the vertices u or v, then we can add *both* copies u' and v' and obtain a strictly larger half-independent set. This contradicts that S has maximum size. Therefore, the set of real vertices of S form an independent set. Since a copy vertex v' is only adjacent to is corresponding real vertex v, then S must contain all copy vertices. Thus, $S = I \cup C$, for some independent set I of G. Then, the claim flows easily.

(b) A subset S of vertices in an undirected graph G is *sort-of-independent* if each vertex in S is adjacent to *at most 255* other vertices in S. Prove that finding the size of the largest sort-of-independent set of vertices in a given undirected graph is NP-hard.

Solution: (direct proof) We can generalize the construction and proof from part a. For each vertex $v \in V$, we create 255 new copy vertices $v_1, \ldots, v_{255}$ and edges (v, v_i) for each $i \in \{1 \ldots 255\}$. Again, let C denote the set of copy vertices. It is easy to verify that for any independent set I of G, $I \cup C$ is sort-of-independent in H.

Claim: The maximum sort-of-independent set in H has the form $I^* \cup C$, where I^* is a maximum independent set of G.

Proof. Let S be a largest sort-of-independent set in H. For contradiction, suppose S contains a real vertex v and $j \in \{1 \ldots 255\}$ other real vertices that are adjacent to v. This means that j of v's copy vertices are not in S, and that for each adjacent real vertex $u \in S$ at least of it's copy vertices is not selected. If we remove v, then we can add all of these missing copy vertices yielding a net gain of $2j - 1 > 0$ additional vertices. This contradicts that S was the largest sort-of-independent set in H. The claim now follows from the same reasoning as part a.

Alternate Solution: (induction) For any $k \geq 0$, define a k-independent set as a subset of vertices $S \subseteq V$ so that any vertex $v \in S$ is adjacent to at most k other vertices in S. For example, a 0-independent set is the standard problem, a 1-independent set is half-independent (part a), and part (b) asks for a 255-independent set.

We show that finding the size of the largest k-independent set is NP-hard for all $k \geq 0$ by induction. We proved the base case, $k = 0$ (the standard IndependentSet problem), in lecture using a reduction from 3SAT. Now assume that for some $k \geq 0$, the k-IndependentSet problem is NP-hard. We show that the $(k + 1)$-IndependentSet problem is hard using a similar approach to (a). Let $G = (V, E)$ be the k-IndependentSet instance. We create the $(k + 1)$ instance (H) by adding a single copy vertex v' for each $v \in V$, and adding an edge (v, v'). Observe that if $S \subseteq V$ is k-independent in G, then $S \cup C$ is $(k + 1)$-independent in H.

Claim: Let I^* be a maximum k-independent set in G. Then, $I^* \cup C$ is a maximum $(k+1)$-independent set in H.

Proof. We use an exchange argument to show that we can swap certain real vertices with their copy counterparts. Let S be a largest $(k+1)$-independent set in H. Suppose that some vertex $v \in S$ is adjacent to $(k+1)$ real vertices in S. This means the copy vertex $v' \notin S$, otherwise v is adjacent to $k+2$ vertices in S. We can swap v and v' without decreasing the size of S. Repeating this replacement for all real vertices adjacent to $(k+1)$ real vertices in S proves the claim.

Problem 4

The Partition problem asks if a set of positive integers X can be partitioned into disjoint subsets A and B, so that the sum of the numbers in each subset is the same. This problem is NP-hard. Consider the following optimization version, partition the set of positive integers X into disjoint subsets A and B so that $\max\{\sum_{a\in A} a, \sum_{b\in B} b\}$ is as small as possible.

(a). Prove that the following algorithm is 3/2-approximation.

```
GreedyPartition(X[1...n]):
    a ← 0, b ← 0
    for i ← 1 to n do
        if a < b then
            a ← a + X[i]
        else
            b ← b + X[i]
    return max(a, b)
```

[Hint: In the best case, the maximum would be $M = \frac{1}{2}\sum_{i=1}^{n} X[i]$. Use a similar analysis to online job scheduling (Lecture 13) and consider two cases: $\max_j X[j] \geq M$ and $\max_j X[j] < M$.]

Solution: Without loss of generality, we can assume that $a \geq b$. Let $X[j]$ be the last number added to a, and let a' and b' be the values of a and b just before this happens. Now, $X[j]$ is added to a since $a' \leq b'$. Therefore, the largest possible value for a' is:

$$a' \leq \frac{1}{2}\sum_{i\neq j} X[i] = \frac{1}{2}\big(\sum_{i=1}^{n} X[i] - X[j]\big) = M - X[j]/2,$$

where $M = \frac{1}{2}\sum_{i=1}^{n} X[i]$. Since $a = a' + X[j]$, this yields the upper bound:

$$a \leq M + X[j]/2.$$

Next we lower bound OPT. For any partition (A, B) of X, we have $\max(A, B) \geq (A + B)/2 = \frac{1}{2}\sum_i X[i] = M$. Therefore, $OPT \geq M$. This bound is loose when $X[j]$

is large. Specifically, if $X[j] > M$, then $\sum_{i \neq j} X[i] < M$ so that the optimal partition is $\{X[j],\ \cup_{i \neq j} X[i]\}$ and $OPT = X[j]$. In summary, we have:

$$OPT \geq \max(M, X[j]).$$

Combining the upper bound on ALG with the lower bound on OPT yields:

$$ALG \leq M + X[j]/2 \leq \frac{3}{2} \max(M, X[j]) \leq \frac{3}{2} OPT.$$

(b). Prove that if the array X is sorted in non-decreasing order, then approximation ratio remains 3/2. That is, give an example of a sorted array X where the output of GreedyPartition is 50% larger than the cost of the optimal partition.

Solution: Suppose $X = \{1, 1, 2\}$. GreedyPartition creates the sets $\{1, 2\}$ and $\{1\}$. The optimal is $\{1, 1\}$ and $\{2\}$.