Luca Sannino 1542194 Homework 2 for Algorithm Design 2020/2021

Exercise 1.1

An ILP that describes the problem can be:

$$\min z$$
$$s.t.\ \ z \geq \sum_{h:\, h \in H_i} c(h) * x_i(h) \qquad \forall \text{ i} \in \text{E}$$
$$\sum_{i:\, h \in H_i} x_i(h) = 1 \quad \forall\, h \in H, \qquad with\ x_i(h) \in \{0,1\}$$

Where:

- H_i is the list of houses that are doable for an elf ***i***, while ***c***(***h***) is the cost of a delivery to a house, which Santa expects to be around 60 minutes.
- $x_i(h) = 1$ means that house ***h*** has been assigned to elf ***i***, otherwise it's 0. Each house must be necessarily assigned to an elf who is able to deliver to it, hence the second constraint above.
- The total delivery time for elf ***i*** is obtained by the sum of all deliveries' costs to the houses in H_i that have been assigned to elf ***i***.
- ***z*** is the time took by the last elf to finish their delivery, so it must be bigger or equal than all the deliveries.
- We want to minimize ***z***.

A LP version of this problem can be simply found by relaxing $x_i(h) \in \{0,1\}$ to $x_i(h) \in [0,1]$.

Exercises 1.2 and 1.3

Given an optimal solution to the LP formulation, an algorithm that outputs a feasible solution to Santa's problem can be this: for each house ***h*** in ***H***, we assign ***h*** to elf ***i*** with probability $p_i(h) = x_j(h)$. With this approach we expect:

$$E[z] = \max_{i \in E} \sum_{h:\, h \in H_i} c(h) * E[h\ is\ given\ to\ i] = \max_{i \in E} \sum_{h:\, h \in H_i} c(h) * p_i(h) = \max_{i \in E} \sum_{h:\, h \in H_i} c(h) * x_i(h) = OPT^{LP}$$

$$With\ OPT^{LP} \leq OPT^{ILP}$$

This randomization alone, however, does not ensure a feasible solution since there's always the possibility that a house is not covered by any elf for which is doable:

$$p(h\ is\ not\ covered) = \big(1 - x_1(h)\big) * \ldots * \big(1 - x_k(h)\big) \leq \left(1 - \frac{1}{k}\right)^k \leq \frac{1}{e}$$

So we can repeat the randomized rounding ***t*** times and finally assign to each elf a set of houses that is the union of all the houses assigned to them across the various iterations. If ***t*** = 2*log***n**, then we have:

$$p(h\ is\ not\ covered\ after\ t\ iterations) \leq \left(\frac{1}{e}\right)^t = \left(\frac{1}{e}\right)^{2logn} \cong \left(\frac{1}{n^2}\right)$$

$$p(\exists\ a\ house\ not\ covered\ after\ t\ iterations) \leq \sum_{h \in H} \left(\frac{1}{n^2}\right) = \frac{1}{n} \leq \frac{1}{m} \quad (because\ of\ Boole's\ inequality)$$

$$p(all\ houses\ are\ covered\ after\ t\ iterations) \geq 1 - \frac{1}{m}$$

However we can't return the biggest delivery time yet because there's also the possibility that a house is assigned more than once which, again, makes the solution not feasible: we can fix this part by adding a greedy portion of the algorithm that iteratively removes the assignment of ***h*** to the elf that has the biggest delivery time between all the elves in conflict over ***h***, until only one elf remains. As a bonus, this means we may also decrease the value of ***z*** if the elf with delivery time ***z*** happens to have ***h*** assigned.

To speed up the algorithm, we can keep track of the deliveries' times for each elf, which houses have been assigned to whom and and how many times a house is assigned by keeping lists and iterators that are refreshed every time we assign (or remove) a house.

Exercise 2

In order to prove that this isn't an easy problem, I need to show that it is NP-Hard.

For each piece ***p*** ∈ ***P***, and for each sleigh ***s*** ∈ ***S***, with $|S| = 187$, I can introduce a variable x_i^{s-c}, where ***c*** indicates the category that ***p*** belongs to (categories can be front lights, seats, reins etc.) and where ***i*** is its index inside the category, that if set to 1 means that ***s*** contains such piece, 0 otherwise: every constraint from Santa can be then expressed by using these variables.

For example, some constraints from Santa may require that in each sleigh there's only one component for a particular category of items: this means that for each sleigh ***s*** ∈ ***S***, and for each category ***c*** ∈ ***C*** of items that has such a constraint, I can write a linear constraint of the type:

$$x_1^{s-c} + x_2^{s-c} + \cdots + x_{|c|}^{s-c} = 1 \; where \; x_i^{s-c} = \{0,1\}$$

Other constraints, instead, may require that the components are chosen in such a way to insure certain characteristics are present, such as a particular color or size. So for each sleigh ***s*** ∈ ***S***, I can write linear constraints with all pieces from the various categories that satisfy that requirement. So, for example, if I want at least one component of color red, I can list all ***x*** that indicate red pieces and write:

$$x_4^{s-seats} + x_2^{s-reins} + \cdots + x_{10}^{s-cockpits} \geq 1 \; where \; x_i^{s-c} = \{0,1\}$$

Regardless of the constraints that Santa may give to us, one of them is always fixed: that no two sleights are exactly the same. Even this last requisite can be written as a linear constraint: as an example, let's assume we are in this possible (and very specific) scenario where Santa requires that in each sleigh there's only one component for <u>each category</u> of items.

So for each distinct and valid combination of items for each sleigh ***s*** ∈ ***S***, I need to make sure that if that particular combination is the one that has been chosen by the algorithm, meaning that the sum of all ***x*** that represent it is $|C|$, no other sleigh ***j*** ∈ ***S*** has the same combination whose sum is $|C|$. Here's the linear constraint for the case where ***s*** may have the first piece for each category, where ***M*** is a very big integer and ***z*** is a new variable to encode the IF statement (note that each combination will have its distinct ***z***):

$$Mz \geq \left(x_1^{s-c1} + x_1^{s-c2} + \cdots + x_1^{s-c|C|}\right) - (|C| - 1)$$

$$M(1-z) \geq (|C| - 1) - \left(x_1^{s-c1} + x_1^{s-c2} + \cdots + x_1^{s-c|C|}\right) \; where \; z = \{0,1\}$$

If ***s*** has all the first pieces for each category, then ***z*** = 1, so to make sure that every sleigh ***j*** ∈ ***S***: ***j*** != ***s*** doesn't have the same combination I can write:

$$\left(x_1^{j-c1} + x_1^{j-c2} + \cdots + x_1^{j-c|C|}\right) * z < |C|$$

Since all requisites from Santa form a decision integer linear program, and since 3-SAT, which is NP-Hard, polinomially reduces to a decision integer linear program, I have proven that also this problem is NP-Hard.

Each clause in a 3-SAT can be in fact written as integer linear constraint. If for example I have:

$$C_1 = \{x_1, \overline{x_2}, x_3\}, \; C_2 = \{x_1, x_4, \overline{x_5}\}, \; C_3 = \{x_2, x_3, x_4\}$$

Then I can write the following decision ILP:

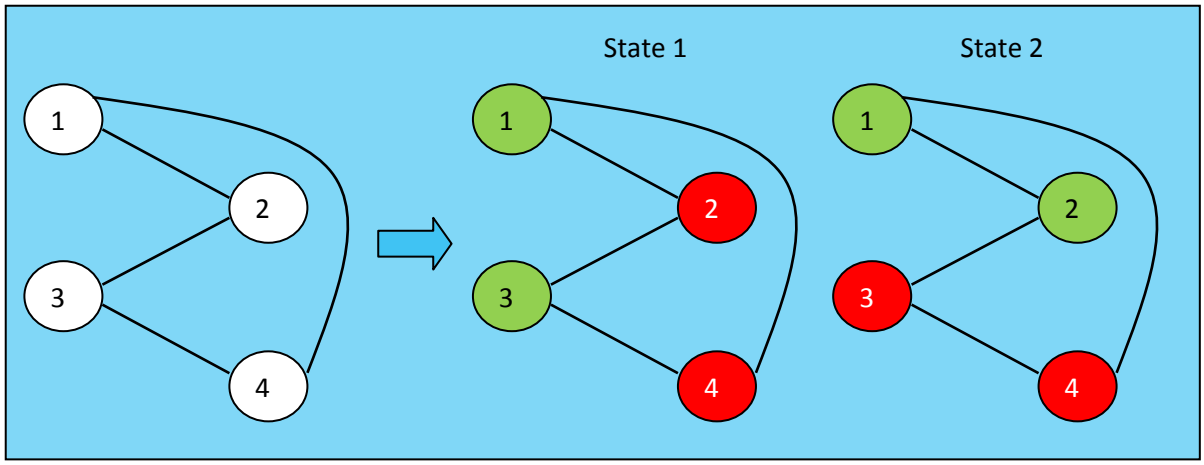
$$x_1 + (1 - x_2) + x_3 > 0$$
$$x_1 + x_4 + (1 - x_5) > 0$$
$$x_2 + x_3 + x_4 > 0 \; where \; x_i = \{0,1\}$$

So that if an assignment of values satisfy the 3-SAT, then also the ILP is satisfied, and vice versa.

Luca Sannino 1542194

Exercise 3.1

Let's consider this graph, and two possible states that we may get after the players choose their set:



If we consider ***GREEN***=***LEFT*** and ***RED***=***RIGHT***, we can easily see that S_1 is an optimal state, since each player gets a payoff of 2, which is the max value a player can get since each vertex has only two edges. This state is also a pure Nash equilibrium, since each player makes the best choice when they consider that the other players are not changing their decision: for example it doesn't make any sense for player 1 to pick ***RED***, since it would decrease their payoff to 0, and the same holds for the others if they choose another set.

In S_2, instead, each player gets a payoff of only 1, since each vertex has an edge connected to a neighbor that belongs to the same set: this state, however, is once again a pure Nash equilibrium, since each player can't alter their payoff if they choose another set, so they have no incentive in changing the status quo: for example if player 1 picks ***RED***, then the payoff remains 1 since there's still an edge connected to a neighbor that belongs to the same set, in this case the newly chosen one.

Since S_1 is an optimal state, and since there are no states that give a worse payoff than S_2 while still being an equilibrium state, the Price of Anarchy is: (Payoff of S_1)/(Payoff of S_2) = 8/4 = 2.

Exercise 3.2

In a more general case, a state in order to be a pure Nash equilibrium needs to have a payoff for each vertex that is at least equal or greater than half of the total weight of its edges. Since there are only two possible sets to choose from, if the payoff for a vertex is smaller than that, then it's clear that the best choice for that vertex should have been the other set, meaning that the current state isn't an equilibrium. The worst payoff that such an equilibrium can give is when each vertex has a payoff of exactly half of the total weight of its edges: any payoff worse than this isn't an equilibrium, as stated above.

This means that in this type of cut games the denominator in the formula of the Price of Anarchy is equal or greater than the total payoff of the state where each edge has an individual payoff of half the total weight of its edges. Meanwhile the numerator is both equal or greater than the denominator, and equal or smaller than the total payoff of the state where each edge has an individual payoff that is equal to the sum of the weights of all its edges, which is the optimal scenario and gives a double payoff of the worst case in equilibrium.

The implication is that any quotient between numerator and denominator gives a Price of Anarchy that is at most 2.

Exercise 4.2

A 3-approximation can be found by using the 2-approximation greedy algorithm for center selection on the set ***S*** of cities: when that algorithm returns a set ***S'*** of size ***k***, we gain a good approximation of where we can set the ***k*** antennas, not because we can set them in the cities themselves (which is what happens in center selection, but it's forbidden in this problem), but because we can choose for each city ***s'*** ∈ ***S'*** its closest antenna.
It's a 3-approximation algorithm because:

- The center selection algorithm, being 2-approx, returns to us a situation where the distance between each city ***s*** ∉ ***S'*** and its closest ***s'*** chosen by the algorithm is at most 2****OPT***, since the distance needs to account for city ***s****, which is part of the optimal set ***S****, contained in the radius of ***s'***: dist(***s***, ***s'***) ≤ dist(***s***, ***s****) + dist(***s****, ***s'***) ≤ ***OPT*** + ***OPT***.
- For each s_i' ∈ ***S'*** and its closest antenna a_i, we have dist(s_i', a_i) ≤ ***OPT***.
- This means that the distance between a city ***s*** ∉ ***S'*** and its closest antenna can be found by this triangle inequality, where ***s'*** is the closest city to ***s*** that has been chosen by the center selection algorithm:
dist(***s***, a_i) ≤ dist(***s***, s_i') + dist(s_i', a_i) ≤ 2****OPT*** + ***OPT*** = 3****OPT.***

Exercise 4.3

Similarly to the proof for the center selection problem that states that there's no p-approximation with ***p*** < 2 unless P = NP, we can show that the Dominating Set also reduces to this problem.
Given an instance ***G***=(***V***,***E***) of the Dominating Set, we can create an instance of our problem:

- We create for each v_i ∈ ***V*** a city s_i and an antenna a_i. We then add an edge for each pair of nodes we obtained.
- For each edge that links a city s_i to its antenna a_i, we add weight = 1.
- For each edge that links a city s_i to an antenna a_j, we add weight = 1 if (***i***, ***j***) ∈ ***E***.
- For each edge that links a city s_i to another city s_j, we add weight = 2.
- For each edge that links an antenna a_i to another antenna a_j, we add weight = 2.
- For each edge that links a city s_i to an antenna a_j, we add weight = 3 if (***i***, ***j***) ∉ ***E***.

If ***G*** has a Dominating Set of size ***k***, then by selecting the ***k*** antennas that are associated to those ***k*** nodes we have a solution of 1, otherwise we have a solution of 3. The same for the other direction. This means that by executing the 3-approximation algorithm on an instance of our problem, we can't know if there's a Dominating Set of size ***k*** in ***G***, since it could return 3 even when the optimal answer was 1.
If, instead, we had a ***p*** < 3 approximation algorithm and there's a Dominating Set of size ***k*** in ***G***, then the result would be always < 3.
Since Dominating Set is an NP-Hard problem, any p-approximation algorithm with ***p*** < 3 would be NP-Hard as well.