Set 2

Platon
Karageorgis
p3180088

(1) Consider the 4 following variations of Max Flow in a network $G = (V, E)$.

(i) In G there are multiple sinks and multiple sources and we want to maximise the total flow from all the sources to every sink.

(ii) Each vertex $v \in V$ includes a value that corresponds to the capacity limiting the Max Flow that can pass through v.

(iii) Each edge $e \in E$ has also (besides the capacity) a lower bound of the flow that can pass through e.

(iv) The ~~flow~~ outward flow of each vertex $v \in V$, is not equal to the inward flow in v, but smaller by a factor $(1-\epsilon_v)$ where ϵ_v is a loss-factor that "accompanies" v.

Prove that each of the above variations of Max Flow can be solved optimally in polynomial time. Specifically, prove that (i), (ii) can be reducted to the standard version of Max Flow while (iii), (iv) can be reducted to the Linear Programming problem.

(i) In this variation, all ~~we~~ we have to do is add an extra source and an extra sink. The source will be connected with every source of the initial network and the sink with every sink of the initial network. The resulting network is obviously classified as a standard Maximum Flow network.

(ii) Since the vertices are limiting Max Flow by having an upper bound in their capacities, we need to make the following transformation. For each vertex in the network we will assign 2 vertices. The first will "control" the outward edges and the other one will control the inward edges. This way, every edge will be assigned to one of the 2 vertices, and the decision will be a matter of the direction. With this transformation, Max Flow is no longer being limited because the network won't have a vertex handling both inward and outward edges. Finally, we can add an extra edge connecting the 2 vertices, to complet the transformation.

(iii) To begin with, let's formulate the standard version of Max Flow as a linear programming problem. We have:

Objective function: $\max_f \left(\sum_{u: s \to u} f_{su} - \sum_{v: v \to s} f_{vs} \right)$

where s: source
t: sink

Constraints: s.t. $\sum_{v: v \to u} f_{vu} = \sum_{w: u \to w} f_{uw}$, $\forall u \neq s, t$ (1)

$0 \leq f_{vu} \leq c_{vu}$, $\forall (v, u) \in E$ (2)

The (1) constraint assures that the flow is conservated while the (2) is enforcing capacity constraints.
All we have to do now, is to add some extra constraint to ensure that each edge has a lower bound of the flow that can pass through it.
We add $f_{vu} \geq b_{vu}$ for each $(v, u) \in E$ where b is the corresponding bound of each edge. Therefore, we have reducted this variation to the Linear Programming problem.

(iv) In the last variation, the problem can be formulated as a Linear program similarly to the (iii) case. The mere difference is that the (1) constraint instead of:

$$\sum_{v: v \to u} f_{vu} = \sum_{w: u \to w} f_{uw}, \quad \forall u \neq s, t$$

will be,

$$\sum_{v: v \to u} f_{vu} = \sum_{w: u \to w} f_{uw} \cdot (1 - \epsilon_u), \quad \forall u \neq s, t$$

This way, the flows (inward/outward) don't have to be equal and therefore the reduction is completed.

② A Formulate the problem "Maximum Independent Set" as an Integer Linear Programming problem. Also, include the LP relaxation. Explain the purpose of each variable, constraint and the objective function's. Moreover answer the following:

- The integrality gap of a snapshot of a problem is defined as the optimal integral solution divided by the optimal solution of the linear relaxation in a LP problem. Find a snapshot of Maximum Independent Set where the integrality gap for the relaxation you formulated is $\leq \frac{2}{n}$.

B Given the following algorithm which returns the Maximum Independent Set S of a graph $G = (V, E)$:

```
RANDOM(G):
Get a permutation π of V uniformly at random;
Find a subset S(π) ⊆ V as follows:
        For each vertex u ∈ V:
                u ∈ S(π) if and only if no neighbour of u precedes u
                in the permutation π
Return S(π)
```

Prove that $S(\pi)$ is ⓐ an independent subset and ⓑ has expected value of cardinality $E[|S(\pi)|] = \sum_{i=1}^{n} \frac{1}{d_i+1}$, where d_i is the degree of vertex i.

A For a start, let's write down the original definition of the Maximum Independent Set. "An independent set of an undirected graph is a subset U of nodes such that no two nodes in U are adjacent. An independent set is maximum if it has a maximum cardinality". In other words, given a graph $G = (V, E)$, start picking vertices that don't share a common edge until there is no other vertex you can pick, with the best possible manner, since the target is a maximum independent set, not a maximal. The problem can be formulated in the following way:

First we create a variable x_i for each vertex $\equiv i$. This variable will have two different possible values, 0 and 1. If $x_i = 0$ it means that the i_{th} vertex will not be included in the solution, whereas if $x_i = 1$ it will be part of it. Since the maximum independent set is trying to maximize the number of the vertices in the final solution, the objective function will be trying to maximize the sum of the x_i's.

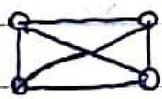
$$\text{Max}\left(\sum_{i=1}^{n} x_i\right)$$

$$\text{s.t.} \quad x_i + x_j \leq 1, \text{ for all } (v_i, v_j) \in E,\ i<j \quad (1)$$
$$x_i \in \{0,1\} \text{ for } i = 1, 2, \dots, n \quad (2)$$

The last thing that we need to mention is that (1) is ensuring that if a vertex i is part of the solution, then every adjacent j will not be picked, since that would violate the core definition of the independent set.

The LP relaxation of the above problem, would allow each x_i to pick a value $\in [0,1]$ instead of $\{0,1\}$. This means that a vertex could be part of the solution, but also a segment of that vertex could be part of the solution. A case where we split a vertex into 2 sub-vertices and one of them is eligible to be picked, is a solid example, since $x_i = \frac{1}{2}$.

Finally, for the integrality question, consider the following graph:



This graph is complete so the IP version would be able to pick only 1 vertex. On the other hand, let's say that the LP version picks $\frac{1}{2}$ of each vertex. Then, $\frac{IP}{LP} \leq \frac{1}{n \cdot \frac{1}{2}} \Rightarrow \frac{IP}{LP} \leq \frac{2}{n}$ where $n = 4$ so if we substitute:

$\frac{2}{4} \leq \frac{2}{4}$, and the upper bound is proved

$\boxed{D}$ (a) To prove that $S(\pi)$ is an independent subset, first we need to show that it is actually a subset, which is obvious and directly infered. The other thing that we need to show is that each vertex in $S(\pi)$ is not adjacent with any other vertex in $S(\pi)$. The algorithm processes the vertices in the order specified by the permutation p. For any vertex added to $S(\pi)$, none of its neighbours will be added because the algorithm doesn't allow it. Therefore, it is not possible to have 2 or more vertices that share a vertex and we have proved that $S(\pi)$ is an independent subset.

(b) Let's first consider the expected probability of a vertex to be placed in $S(\pi)$. The vertex has to follow the algorithm's logic so it can't be preceded by any of its neighbours. We will explain with a simple example. If i has 3 neighbours then it has $d_i = 3$ where $d_i \equiv \text{degree}(i)$. Considering the possible permutations p, i can be placed e.g. in position 1 but then none of its neighbours can be placed in position 1, 2, 3 because in that case it would be preceded. The only case that would be valid is if none of its neighbours precede it and that would happen only in 1 out of 4 permutations. We can generalize it as $\frac{1}{d_i+1}$ which is again, the probability of i being selected through d_i+1 possible positions in the permutation π. This holds true for any vertex i, regardless of its degree. So, the expected cardinality of $S(\pi)$ is:

$$E[|S(\pi)|] = \sum_{i=1}^{n} \frac{1}{d_i+1}.$$

This is deduced simply by adding $\frac{1}{d_i+1}$ for each vertex and in total we have n vertices.

③ Given the weighted Vertex Cover problem:

(i) Formulate the problem as an integer linear programming problem

(ii) Assume that the weights of the vertices in G are equal to 1 and that G is a triangular graph containing three vertices $V=\{a,b,c\}$ and three edges $E=\{(a,b),(b,c),(c,a)\}$. Formulate an integer linear program that describes this snapshot of the problem and also append its linear relaxation. Explain the execution of the LP-Rounding algorithm in this snapshot and find the solution it returns. What is the value of the proximity factor reached in the above snapshot? Is the above snapshot a tight example of the LP-Rounding algorithm? If not, find an example for which the LP-Rounding algorithm has a 2-proximity factor.

(i) The weighted vertex cover can be formulated as an integer linear programming problem as follows:

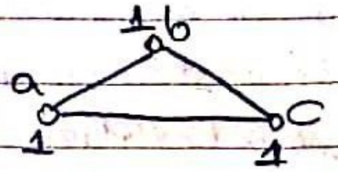
$$\min \sum_{v \in V} w_v \cdot x_v$$

$$\text{s.t. } x_u + x_v \geq 1, \ \forall (u,v) \in E \quad (1)$$

$$x_v \in \{0,1\}, \ \forall v \in V \quad (2)$$

The variable X represents the vertices and since we are formulated the weighted version of vertex cover, we assign a weight W to each vertex. The product of W·X which obviously has to be minimized, comprise the objective function. The (1) constraint is making sure that each edge will never have more than 1 adjacent vertex in the vertex cover, which is a basic VC constraint and finally (2) is just showing that X can have only integer values, 0 or 1. If $X_i=0$ for a vertex i then it will not be part of the solution and in case of $X_i=1$ we will include it. Therefore, we have completed the formulation.

(ii) To begin with, let's draw G



We already have the generic weighted vertex cover ILP formulation so in this part we will make it specific for the given snapshot.

$$\min \; X_a + X_b + X_c$$

Since the weights are equal and have value=1 we don't need to add them to the formulation

s.t.

$$X_a + X_b \geq 1$$
$$X_b + X_c \geq 1$$
$$X_c + X_a \geq 1$$
$$X_a, X_b, X_c \in \{0,1\}$$

To apply the LP-Rounding algorithm we need to convert this to a LP problem, so we have to allow X_a, X_b, X_c to have values in $[0,1]$. We can see that the best case for the LP method is to set each X_i to $1/2$, since all the constraints will be successfully met. The rounding method will pick at least one of $X_u, X_v = \frac{1}{2}$. This results to:

$$\left.\begin{array}{l} OPT_{LP} = 1/2 + 1/2 + 1/2 = \frac{3}{2} \\ OPT_{LPR} = 2 \end{array}\right\} \Rightarrow OPT_{LP} \leq \frac{3}{4} OPT_{LPR}$$

$$\Rightarrow \boxed{OPT_{LPR} \geq \frac{4}{3} OPT_{LP}}$$

Which means that the snapshot is not a tight example.

A tight example with a 2-approximation would be the n-clique where $n \to \infty$. We can see that complete graphs are the most efficient examples for the LP method, because the rounding method always - except $n=1$ case - needs $n-1$ cost to cover every edge. So, while $n \to \infty$ the $\frac{OPT_{LP}}{OPT_{LPR}}$ would approximate $1/2$, or to be more precise, $\boxed{OPT_{LPR} \geq OPT_{LP} \cdot 2}$

④ (i) Formulate the following problem as an integer linear programming problem. Assume a set of clients L, who work in a delivery service company, and a set of assistance/store points S belonging to the company. Assume that each client $i \in L$ waits for a notification from the company in order to go to a store $j \in S$ and pick-up the parcel. We know that the client i has a commute cost $c_{i,j}$ for each store $j \in S$, while every store $j \in S$ has a usage cost s_j. The target is to advise the company to choose which stores they should use in order to assist all the clients while achieving the lowest possible cost including commute and usage costs.

(ii) Assume that you have formulated an integer linear program for an optimization problem. You have used 5 integer variables x_1, x_2, x_3, x_4, x_5, $x_i \in \{0,1\}, i=1,\dots,5$. Explain how would you add a linear constraint to your integer program so that you can enforce each of the following:

1. At most one of x_1, x_2 equals to 1
2. Between x_3 and x_4 exactly one equals to 1
3. If in a valid solution $x_4 = 1$, then $x_5 = 1$
4. At most 3 variables are equal to 1.

(i) We will first see the formulation and there will be detailed explanation afterwards.

$$\min \sum_{i \in L} \sum_{j \in S} \left((c_{ij} + s_j) \cdot x_{ij} \right)$$

s.t. $x_{i,j} \in \{0,1\}, \forall i \in L, \forall j \in S$

$c_{i,j}, s_j > 0, \forall i \in L, \forall j \in S$, respectively

The ILP doesn't need any constraints besides the basic constraint that forces the core variable of the problem to have integer values. The 2 different costs are constant and obviously greater than 0, so the focus should be on the objective function. The idea is simple, given a random example with i clients and j spots, calculate the cost of each pair and find a sum for every case The case that will be selected will be the one with the lowest total cost. The program will eventually return the minimum cost and most importantly, through it we will be able to get "j" which will contain the number of the optimal number of spots.

(ii) 1. This one is straightforward $\Rightarrow x_1 + x_2 \leq 1$

2. The next one will be accompanied by a truth table in order to explain the idea.

$$x_3 \Leftrightarrow x_4$$

x_3	x_4	Values I want	$x_3 \Leftrightarrow x_4$
0	0	0	0 ✓
1	0	1	1 ✓
0	1	1	1 ✓
1	1	0	0 ✓

3. $(x_4 = 1) \Rightarrow (x_5 = 1)$

For $x \neq y \neq z$,

4. $[\exists x, \exists y, \exists z (x=1 \wedge y=1 \wedge z=1) \Rightarrow (\forall k\, (k=1) \Rightarrow (k=x) \vee (k=y) \vee (k=z))]$

The above type states that if there is a case where there are 3 different variables true -and therefore 1-, then if you find a 4th variable that is true, it has to be equal to one of x, y, z.