Homework #7
Introduction to Algorithms
601.433/633
Spring 2020

Due on: Wednesday, April 29th, 12pm

Problem 1 (15 points)

For a directed graph $G = (V, E)$ with capacities $c : E \to \mathbb{R}^+$ and a flow $f : E \to \mathbb{R}^+$, the *support* of the flow f on G is the set of edges $E_{\mathsf{supp}} := \{e \in E \mid f(e) > 0\}$, i.e. the edges on which the flow function is positive.

Show that for any directed graph $G = (V, E)$ with non-negative capacities $c : E \to \mathbb{R}^+$ there always exists a maximum flow $f^* : E \to \mathbb{R}^+$ whose *support* has no directed cycle.

Hint: Proof by contradiction?

Proof. I want to prove this with contradiction. Let's make a claim that for directed graph $G = (V, E)$ with non-negative capacities $c : E \to \mathbb{R}^+$, there doesn't exist a maximum flow f whose support has no directed cycles.
From our claim we know that the support of flow f has directed cycles. For each cycle in the original, we can do the following 2 steps to eliminate the cycle until at least the flow of one edge in the cycle becomes zero:
1. Find the edge with minimum value of flow in this cycle. we assume that the minimum flow value is s.
2. for each edge of this cycle, reduce the flow of each edge by s.
In this way, since the flow in the cycle will not got out of the cycle, the new flow is still the max flow of the graph. More specifically, diminishing a cycle of flow does not change the max flow because the total output of this cycle does not change.

And after doing these steps for all cycles are diminished and we get a maximum flow f with its support has no directed cycle. This contradicts with our claim.
So the claim is wrong. For any directed graph $G = (V, E)$ with non-negative capacities $c : E \to \mathbb{R}^+$ there always exists a maximum flow $f^* : E \to \mathbb{R}^+$ whose *support* has no directed cycle. $\square$

Problem 2 (20 points)

In a graph $G = (V, E)$, a matching is a subset of the edges $M \subseteq E$ such that no two edges in M share an end-point (i.e. incident on the same vertex).

Write a linear program that, given a bipartite graph $G = (V_1, V_2, E)$, solves the maximum-bipartite-matching problem. I.e. The LP, when solved, should find the largest possible matching on the graph G.

Clearly mention the variables, constraints and the objective function. Prove why the solution to your LP solves the maximum-bipartite-matching problem. You may assume that all variables in the solution to your LP has integer values.

Solution:
1.The original maximum-bipartite-matching problem can be solved by maximum flow. This is a corollary mentioned in the book. The cardinality of a maximum matching M in a bipartite graph G equals the value 0 of a maximum flow f in its corresponding flow network G.(This is introduced at Chapter 26.3 and at Corollary 26.11 in CLRS) Therefore, I can transfer the original problem into the following max flow problems. This transformation is introduced on book at Chapter 26.3. So now Let's construct the corresponding flow network the same way as the book does.
Let's define the source of the max flow as s, the sink as t. Then we can define $V' = \{s, t\} \cup V_1 \cup V_2$. $E' = \{(s, u) : u \in V1\} \cup \{(u, v) : (u, v) \in E\} \cup \{(v, t) : v \in V2\}$. Then the original maximum-bipartite-matching problem becomes max flow problem on $G' = (V', E')$ with source s and sink k. To complete the construction, we assign unit capacity to each edge in E'. This means the capacity of each edge in E' is 1.

2.The maximum-flow problem can also be solved as a linear program. This is also mentioned in the Chapter 29.2 in CLRS. A maximum flow is a flow that satisfies these constraints and maximizes the flow value. So at first we must define the optimization function in the LP problem. This function corresponds to the requirement of maximizing the flow value. Since source s only has out flow, we can define the optimization as $\sum_{v \in V_1} f_{sv}$.
Now we must set several constrains to keep the flow as a max flow problem. The first constrain is to make each flow less or equal to the capacity of that link. In this case, all capacities are 1. The second is to make sure that the for every node in $V_1 \cup V_2$, inflow is equal to outflow. This is the basic concept of the max flow

problem. Only the outflow of the source s and the inflow of the sink t is positive. The third constrain is that each flow must be greater or equal to 0. So the LP is presented below. These three constrains correspond to the three requirements in the max flow problem one by one, and they are enough to convert the max flow problem to the linear programming. The format of these 3 constrain are shown below.

So the overall linear programming is as below.

maximize: $\sum_{v \in V_1} f_{sv}$

subject to:

$f_{uv} \leq 1$ for each $u, v \in V'$

$\sum_{v \in V} f_{vu} = \sum_{v \in V} f_{uv}$ for each $u \in V_1 \cup V_2$

$f_{uv} \geq 0$ for each $u, v \in V'$

Problem 3 (15 points)

In class you saw that VERTEX-COVER and INDEPENDENT-SET are related problems. Specifically, a graph $G = (V, E)$ has a vertex-cover of size k *if and only if* it has an independent-set of size $|V| - k$.

We also know that there is a polynomial-time 2-approximation algorithm for VERTEX-COVER. Does this relationship imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for INDEPENDENT-SET? Justify your answer.

Solution:
No.
We can prove this conclusion by contradiction.
As we know, the approximation ratio of $\rho(n)$ is defined as for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$
Since the statement in the question is true. A graph $G = (V, E)$ has a vertex-cover of size k *if and only if* it has an independent-set of size $|V| - k$. Then we can find out that having a maximum vertex-cover with size k means that the size of minimum independent set is $|V| - k$.(Because k is maximized, $|V| - k$ is minimized.)
Let's assume that we there exists a polynomial-time P-approximation algorithm for Independent Set, which P is a constant. Since the optimal(smallest) value of vertex-cover is C^*, using the algorithm APPROX-VERTEX-COVER in the book, we can find an a vertex-cover of size C. And we have $\frac{C}{C^*} \leq 2$, which is $C \leq 2C^*$. Here for the simplicity of our proof, Let's assume that $C = 2C^*$.
Assuming that the optimal(largest) value of independent set is K^*, then with this algorithm, we can find an independent set of size K. Since this is a P-approximation algorithm, we have $\frac{K^*}{K} \leq P$, which is $PK \geq K^*$. Since a graph $G = (V, E)$ has a vertex-cover of size k *if and only if* it has an independent-set of size $|V| - k$, we have $K^* = |V| - C^*$, $K = |V| - C$
So we have

$$PK \geq K^* \Longrightarrow P \geq \frac{K^*}{K} = \frac{|V| - C^*}{|V| - C} = \frac{|V| - C^*}{|V| - 2C^*}$$

Since $\frac{|V|-C^*}{|V|-2C^*}$ is different on different graph and it has no upper bound. Then P is not a constant, which contradicts with our assumption.
Therefore, this relationship doesn't imply that there is a polynomial-time approximation algorithm with a constant approximation ratio for INDEPENDENT-SET

Problem 4 (10 points)

You are given a biased coin c but you don't know what the bias is. I.e. the coin c outputs *H* with probability $p \in [0, 1]$ and *T* with probability $1 - p$ every time you toss it but you don't know what p is.

Can you simulate a fair coin using by tossing c? I.e. Come up with a "rule" for tossing c (possible several times) and outputting *H* or *T* based on the output of the coin tosses of c such that you will output *H* with probability $1/2$ and *T* with probability $1/2$? Please prove the correctness of your solution.

Hint: i) What if you had two coins c_1, c_2 with the same bias instead of one? Can you toss them together and output *H* or *T* based on the output? ii) You shouldn't be trying to "estimate" p.

Solution:
To solve this problem, we must find the 2 cases with the same probability. Here's my solution: 1. Toss the coin twice. 2. If the result in order is "head"-"tail", then return "head". Otherwise if the result in order is "tail"-"head", then return "tail". 3. If the result is "head"-"head" or "tail"-"tail", return to step 1 until it ends with "head"-"tail" or "tail"-"head".

Proof of correctness:

Proof. For the four cases, the probability is shown as below:
1. $P[\text{"}head\text{"} - \text{"}head\text{"}] = p * p = p^2$
2. $P[\text{"}head\text{"} - \text{"}tail\text{"}] = p * (1 - p) = p(1 - p)$
3. $P[\text{"}tail\text{"} - \text{"}head\text{"}] = (1 - p) * p = (1 - p)p$
4. $P[\text{"}tail\text{"} - \text{"}tail\text{"}] = (1 - p) * (1 - p) = (1 - p)^2$
For the solution above, let's consider the probability of the return "head".

$$\begin{aligned} P[return\text{"}head\text{"}] &= P[return\text{"}head\text{"} | return\text{"}head\text{"} or\text{"}tail\text{"}] \\ &= \frac{P[\text{"}head\text{"} - \text{"}tail\text{"}]}{P[\text{"}head\text{"} - \text{"}tail\text{"}] + P[\text{"}tail\text{"} - \text{"}head\text{"}]} \\ &= \frac{p(1-p)}{p(1-p) + (1-p)p} = \frac{1}{2} \end{aligned}$$

Similarly,

$$\begin{aligned} P[return"tail"] &= P[return"tail"|return"head"or"tail"] \\ &= \frac{P["tail" - "head"]}{P["head" - "tail"] + P["tail" - "head"]} \\ &= \frac{(1-p)p}{p(1-p)+(1-p)p} = \frac{1}{2} \end{aligned}$$

So we will output *H* with probability $1/2$ and *T* with probability $1/2$.

□