CSOR 4231: HW 6

1 Problem 1

We need to show that f_p satisfies capacity constraints (capacities here are the residual capacities $c_f(u,v)$) and flow conservation on the residual graph G_f.

$f_p(u,v) = 0$ if (u,v) is not on path p. If (u,v) is on path p,

$$\begin{aligned} f_p(u,v) &= c_f(p) & (1)\\ f_p(u,v) &= \min\{c_f(a,b) : (a,b) \text{ is on } p\} & (2)\\ f_p(u,v) &\leq c_f(u,v) & (3)\end{aligned}$$

Thus f_p satisfies the capacity constraint.

If u does not lie on path p, then there is no flow entering or exiting u, so flow conservation is maintained. If u is on p then either u is the source or sink, or $\exists\ a,b$ such that (a,u) and (u,b) lie on p. Since p is simple, a and b are unique. Then $f_p(a,u) = f_p(u,b) = c_f(p)$ so flow conservation is maintained. Since p is simple, there is a unique node b such that (s,b) is on p, and there is no node a with (a,s) on p. Thus, $|f_p| = c_f(p) > 0$.

2 Problem 2

Let f be a maximum flow on G. Either $f(e) = 0\ \forall e \in E$ or there exists some edge e such that $f(e) = \min\{f(d) : f(d) > 0\}$. Now find some path p that contains e (otherwise f is not a flow). We subtract $f(e)$ amount of $flow$ along p. The result is a new graph G' with maximum flow f'. Now, the observation is that f' is 0 on one more edge than is f. After each iteration, there exists at least one more edge with 0 flow, so after at most $|E|$ iterations the resulting flow would be 0 and the algorithm terminates. The paths found in each iteration is the series of augmenting paths that we want.

3 Problem 3

Since the problem is in NP, there is some polynomial-time verifier A for it such that for any input x of length n, there is a certificate y of length polynomial in n such that $A(x,y)$ = true. In particular, since y must be polynomial in

n, we know that there exists some constant k such that we only have to check certificates of size $O(n^k)$.

The certificate string is a binary string. Since it has length $O(n^k)$ there are at most $2^{O(n^k)}$ possible certificates. Thus, our algorithm is as follows: given any input x, run $A(x, y)$ for all possible certificates y. If $A(x, y)$ = true for any certificate y, then input x has a solution; else, if $A(x, y)$ = false for *all* certificates y, input x has no solution.

Given input x we run $A(x, y)$ for $2^{O(n^k)}$ certificates y, and each run takes polynomial time (since A must be a poly-time verification algorithm), so the total run-time is $2^{O(n^k)}\cdot$ poly(n). It is not hard to see that the poly(n) factor gets swallowed up in the $O(n^k)$ in the exponent, leading to an overall run-time of $2^{O(n^k)}$.

(You only had to mention that the poly(n) term gets swalloed up the $O(n^k)$ in the exponent; you didn't have to go into details. But I'll do it our here for those of you who want to see the formal proof. poly(n) means $d \cdot n^c$ for some constants d, c. But $d \cdot n^c < 2^{cd\log(n)}$ and $cd\log(n) = O(\log(n)) = O(n^k)$, so

$$2^{O(n^k)} \cdot d \cdot n^c < 2^{O(n^k)} 2^{cd\log(n)} = 2^{O(n^k)+cd\log(n)} = 2^{O(n^k)}$$

4 Problem 4

Proof that the 0-1 integer programming question is NP-complete. First of all we must demonstrate that the problem is in NP. This is clear because given a solution, x to an 0-1 ILP, we can verify that $Ax \leq b$ in n^2 time (the time to multiply A by x).

After demonstrating that 0-1 ILP is in NP, we must demonstrate that it is NP-hard in order to prove that it is NP-complete. We will do this by reducing 3-CNF-SAT to it in polynomial time. We will now convert a given instance of 3-CNF-SAT, ϕ, 0-1 ILP.

Create a constraint for each clause in ϕ. For a given clause, replace the $\vee$'s with $+$, and negations of variables with $1 - x_i$. e.g.:

$$(x_1 \vee \bar{x_2} \vee x_3) \rightarrow x_1 + (1 - x_2) + x_3 \geq 1$$

Set the constants relating to the given clause variables to 1 for normal variables, and -1 for negated variables. Put zero's in the rest of the entries of A for the given clause (row), and finally set $b_i = 1-$ the number of negated variables. This way one of x_1, $\bar{x_2}$ or x_3 will need to be true in order to satisfy the above constraint.

If there is a solution to this 0-1 ILP then there is a solution to the satisfiability problem. This is clear because each constraint is satisfied iff a clause in ϕ is satisfied, therefore if all constraints are satisfied, then all clauses are satisfied. By similar logic, if there is not a solution to the 0-1 ILP then there is no solution to ϕ.

5 Problem 5

1. Suppose there are m_1 coins worth x dollars and m_2 coins worth y dollars. You need to find some a and b such that $ax + by = \frac{m_1x+m_2y}{2}$. Doing a brute force approach where we check each a and b possible is $O(n^2)$ since a and b are both upper bounded by n. Thus, there is a polynomial time algorithm.

2. Let d_i be the number of coins of denomination 2^i. Assume 2^k is the largest denomination. We do the following step:

   (a) If d_k is even, then we split d_k between Bonnie and Clyde.

   (b) If d_k is odd, we give one more to Bonnie than Clyde. Then we greedily give coins from the largest denomination to the smallest to Clyde until they have the same amount.

   If there is a way to split the amount in half, then I claim the previous procedure finds it. We prove correctness by induction. The base case means there is an even number of 1 coins and we just split it between them.

   If d_k is even, we split the 2^k coins between them and then apply induction. If d_k is odd, Bonnie gets one more coin than Clyde. When we greedily give coins to Clyde, there is no point in the procedure where Clyde has less than Bonnie, and then adding another coin pushes Clyde to have a strictly greater amount than Bonnie. Thus, if the amount can be split, at some point in the procedure, Clyde will have the same amount as Bonnie. Afterwards, apply induction.

   The whole procedure takes $O(n)$ time because each time we examine a coin, we allocate it to Bonnie or Clyde.

3. This is simply the Partition problem, thus is NP-Complete.

4. We reduce Partition to this problem. Suppose we take any arbitrary instance of Partition: We're given some set of natural numbers A, let $S := \sum_{x \in A} x$ and we want to find some set B such that $\sum_{x \in B} x = S/2$. If the difference between any two numbers in A is greater than 100, then solving the Bonnie and Clyde problem will also solve Partition.

   Suppose the minimum difference is less than or equal to 100. Now multiply each number by 101. Then S becomes $101S$ and the sum of any subset will be 101 times the sum before. Thus, solving the Bonnie and Clyde problem here will give an exact solution to the original problem. Therefore, this Bonnie and Clyde problem is NP-Hard.

6 Problem 6

The algorithm does not give a 2-approximation. Construct a bipartite graph $G = (A, B, E)$ as follows: It has n vertices in A. In B, we keep adding vertices as follows: We do this in phases, starting with $i = 2$.

In the i^{th} phase we add n/i vertices into B. Each vertex is adjacent to i vertices in A. Furthermore, the neighborhoods of any two vertices added in the i^{th} phase are disjoint. We stop after the n^{th} phase.

By now we have added essentially $n \ln n$ vertices into B. (Strictly speaking we added $\lfloor (n/i) \rfloor$ vertices into B in phase i - but the sum is still at least $n \ln n - n$). The proposed greedy vertex cover algorithm always removes vertices from B in the reverse of the order in which they were inserted, since at the time a vertex is introduced (in the i^{th} phase) it has degree i and all vertices in A have degree (at most) $i - 1$. Thus, this algorithm would place all vertices of B into our vertex cover, which is roughly $n \log n$ vertices. However, notice that the vertices in A form a vertex cover and only has n vertices. Thus, we get a $O(\log n)$ approximation rather than a 2-approximation.