CSOR 4231, Fall 2015
Problem Set 6 Solutions

Problem 1. Exercises 25.2-6. 25.1-10. All pairs shortest paths.

25.2-6

If any shortest-path distances from node i to itself, i.e., the main diagonal, becomes negative then there is a path from i to itself, i.e. a cycle, with negative weight. If a negative cycle exists, it will be discovered by the time the algorithm finishes because it is a path with cheaper weight than the original weight, namely 0. Also, the algorithm will only reveal this cycle when one exists because there is no other way for such a path to be discovered.

Alternatively, one could just run the normal FLOYD-WARSHALL algorithm one extra iteration to see if any of the d values change. If there are negative cycles, then some shortest-path cost will be cheaper, if there are no such cycles, then no d-values will change because the algorithm gives the correct shortest paths.

25.1-10 Run FAST-ALL-PAIRS-SHORTEST-PATHS on the graph and return the k value when $l_{ii}^{(k)}$ first becomes negative.

Problem 2. Exercises 26.1-2, 26.2-10. Maximum Flows. For 26.2-10, the problem is asking you to show that there always exsists a series of of at most E augmenting paths that leads to a maximum flow. You can assume that you already know the maximum flow, you have to explain how to construct the E paths.

Exercise 26.1-2

For a single-source single-sink network, there are two flow properties which need to be satisfied:

1. Capacity constraint: $\forall u, v \in V, 0 \leq f(u,v) \leq c(u,v)$
2. Flow conservation: $\forall u \in V - \{s,t\}, \sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$

Moreover, the the value of the flow is as follows

$$|f|_{single} = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s) = \sum_{v \in V} f(s,v)$$

Now, to convert a multiple-source, multiple-sink to a single-source, single-sink network, construct a new flow network G' as follows:

Sum up the flow for all sources, and get the value of the flow for multiple sources as: ($S = \{s_1, s_2, \ldots, s_n\}$)

$$|f|_{multiple} = \sum_{v \in V} \sum_{s \in S} f(s,v)$$

Next, add the super source s_0 and super sink t_0 with the following conditions

$$c(u, v) = \infty \text{ for } u = s_0, v = \{s_1, s_2, \ldots, s_n\}$$

$$c(u, v) = \infty \text{ for } v = t_0, u = \{t_1, t_2, \ldots, t_n\}$$

By following this procedure, we ensure that the following conditions are still satisfied for any arbitrary flow:

1. Capacity constraints are followed, since the new edges added have infinite capacities.
2. All internal vertices satisfy flow conservation properties
3. For $S = \{s_1, s_2, \ldots, s_n\}$ and $T = \{t_1, t_2, \ldots, t_n\}$, since ∞ flow is permitted along each (u, v) where $u = \{s_0\}$ and $v = \{s_1, s_2, \ldots, s_n\}$, we get $f(s_0, s_i) = f(s_i, \text{any internal vertex})$ where $i \in \{1, 2, \ldots, n\}$. This ensures that we provide equivalent flow from the source to the vertices as needed.
4. In the same way for the super sink, we assign $f(v, t_i) = f(t_i, t_0)$ where $v = t_0$. Again, using flow conservation, we release the flow from each vertex into the super sink.

Finally,

$$|f|_{multiple} = \sum_{v \in V} \sum_{s \in S} f(s, v) = \sum_{s \in S} f(s_0, s) = |f|_{single}$$

26.2-10

From the max flow network find the max flow graph, in which the weight on each edge is equal to the flow on same edge in the max flow network, if there is no flow on a edge, then delete it from the max flow graph. Then run DFS on the max flow graph from the source until we have reached the sink. Store this path as a augmenting path, then find the smallest edges on the path, deleting these edges from the graph and decrease the weight of other edges on the path by the same value that these smallest edges have. Continue this process until the sink is not reachable and all the augmenting path from the source to the sink are discovered. Each time we at least eliminate one edge from the graph and at the same time find one augmenting path.So the total number of augmenting paths won't be more than $|E|$ and these less or equal than $|E|$ Augmenting paths results a max flow.

Problem 3. Problem 26-2. Minimum Path Cover.

1. Consider a max-flow on the suggested graph G', with source x_0 and sink y_0. Let the capacities of the arcs be 1. The claim is that every maximal set of vertices $S_{i_1,\ldots,i_k} = \{x_{i_1}, yi_2, x_{i_2}, y_{i_2}, \ldots, x_{i_k}, y_{i_k}\}$ in G', where $(x_{i_1}, y_{i_2}), \ldots, (x_{i_k}, y_{i_k})$ carry flow, corresponds to a path $< v_{i_1}, v_{i_2}, \ldots, v_{i_k} >$ in the minimum cardinality set of disjoint paths in G, that cover the entire V.

   The paths are disjoint, because no x_i or y_i, $i \neq 0$, can participate in more than one paths, since in a max-flow solution the flows are integers and every arc has capacity 1.

The set of paths has minimum cardinality : for every maximal set

$$S_{i_1,\ldots,i_k} = \{x_{i_1}, yi_2, x_{i_2}, y_{i_2}, \ldots, x_{i_k}, y_{i_k}\}\ , k = 1, \ldots, n$$

of vertices in G', and a flow f we have :

$$S_{i_1,\ldots,i_k} = \sum_{(x_i,y_j)\in E',\ x_i,y_j\in S_{i_1,\ldots,i_k}} f(x_i, y_j) + 1. \tag{1}$$

If we add the equalities for all the sets we see that V, the number of vertices in G, equals the value v of the flow in G' plus the number of paths in the set. Thus, a max-flow f will yield a minimum cardinality set with $V - v(f)$ elements.

For this problem, the running time of the generic max-flow algorithm presented in class is $O(VE)$, since the value of the flow is bounded above by V.

2. If the graph has cycles the algorithm does *not* work. The reason is that equation (1) does not hold any more. For example, consider the following situation with 5 nodes and edges: $< 1, 3 > < 3, 4 > < 4, 5 > < 5, 1 >$ Even though the value of the maximum flow is 4, we get two disjoint paths : $< 1, 3, 4, 5 >$ and $< 2 >$. This is because of the cycle $< 1, 3, 4, 5, 1 >$.

Problem 4. Graph Isomorphism and Hamiltonian Path

Graph Isomorphism

Let the input be a node map of from graph $G_1(V_1, E_1)$ to $G_2(V_2, E_2)$, the map shows as $i_1, i_2, ..., i_n$, where i_k means node k in G_2 maps node i_k in G_1. Now we prove that verifying the map is a correct solution is in polynomial time.

(a) Verify all i_k is unique and $1 \leq i_k \leq n$.

(b) For each pair $i_k i_l$, verify that $(i_k, i_l) \in E_1$ if and only if $(k, l) \in E_2$.

(c) If any of the above step fails, return false. Else return True;

Step (a) takes at most $O(V)$ time and step (b) runs in $O(V + E)$ time, therefore the verification runs in $O(V + E)$ time. Thus the problem of determining whether two graphs are isomporphic is in NP.

Hamiltonian Path

Let the input be a sequence of vertices $v_1, v_2, ..., v_n$ is a path in graph $G(V, E)$ from x to y. Now we prove that verifying the sequence is a hamiltonian path is in polynomial time.

(a) Verify all v_k is unique, $1 \leq v_k \leq n$, and $n = |V|$.

(b) Verify if $v_1 = x$ and $v_n = y$.

(c) For all pair v_k, v_{k+1}, verify that $(v_k, v_{k+1}) \in E$.

(d) If any of the above step fails, return false. Else return True;

Step (a) and (b) takes at most $O(V)$ time and step (c) takes at most $O(V)$, therefor the verification runs in $O(V)$ time. Thus the problem of determining whether there is a hamiltonian path from x to y is in NP.

Problem 5. Exercise 34.5-2. Integer linear programming.

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.

Problem 6. Problem 34-2. Bonnie and Clyde.

0.1 a

There is a polynomial time solution to this problem. If there are n total coins, then w.l.o.g, assume that there are a coins worth x dollars, and $n - a$ coins worth y dollars. This means that their are $a * (n - a)$ unique sums that one of the two subsets of coins can take. An algorithm for solving this problem is as follows: enumerate all sums, e.g. start with 0 coin of worth x and 0 coin worth y on the left, and a coin worth x and $n - a$ coin worth y on the right. Check if the sums are equal. If they are, then we are done. If they are not then move one of the a coins to the left and repeat.

Once all a 'x' coins are on the right, move all of them back to the right and move one of the $n - a$ 'y' coins to the left and repeat. If at any point the sum on the left is equal to the sum on the right, return that partitioning of the coins, or if we end up with a 'x' coins and $n - a$ 'y' coins on the left then return 'no feasible solution'. It is clear that this simple algorithm runs in time $O(n^3)$. This is a naive, but straightforward, way to approach this problem.

0.2 b

There is a greedy polynomial time solution to this problem. The solution stems from the observation that larger powers of two are made up of smaller powers of two, e.g. $2^a = \sum_{i \in S} 2^i$ for some set S, where all $i \leq a$. Another way to think about this is that, given $2^a > \sum_{i \in S} 2^i$ where $i < a$, if we add 2^j, where $j < a$, it is impossible to transition to $2^a < \sum_{i \in S} 2^i + 2^j$, but rather 2^a must be greater or equal to $\sum_{i \in S} 2^i + 2^j$.

Here is the algorithm: Sort the coins in decreasing order. Give Bonnie the largest coin, then give Clyde the next largest, and then the next largest, and so on until his coins total to Bonnie's. This is guaranteed to occur because of our original observation about powers of two.

When they have the same amount, give Bonnie the largest remaining coin and repeat the above process until all coins are gone. If after giving Bonnie a coin, the sum of the remaining coins is less than her most recent coin, then their is no solution to the problem. If we end this process with no coins remaining, then Bonnie and Clyde have partitioned the coins appropriately.

This algorithm clearly runs in polynomial time as it requires a single pass through the coins (Clyde can keep a running sum at any given time, so as to avoid duplicate summations).

0.3 c

We show that this problem is in NP. We are given an equal split, where s checks go to Bonnie and t checks go to Clyde such that s + t = n. We then must verify that this split is equal. We do this by adding up all the checks for Bonnie and for Clyde respectively and checking if the monetary value is equal. If they are, then the split is equal, if not, then the split is not equal. This takes polynomial time, namely O(n), since we have to look at n checks once to add it to the sum of either Bonnie or Clyde. Thus we are in NP.

The function to convert BONNIE-CLYDE-CHECKS to SUBSET-SUM: to find t, we sum up all the money from the n checks, where c_i is the value of each check:

$T = \sum_{i=1}^{n} c_i$

Our set S is the n checks. We then add one more check c_{n+1} by the following: If $t \geq T/2$, then choose c_{n+1} such that T + c_{n+1} = 2t, so c_{n+1} = 2t - T.

Otherwise, if $t < T/2$, then choose c_{n+1} such that t + $c_{n+1} = \frac{T+c_{n+1}}{2}$, so c_{n+1} = T - 2t.

This ensures an equal partition because we fix the sums to be equal to 2t.

This conversion function runs in polynomial time with respect to n, namely O(n) to run the sum of all the checks to find T.

We now must prove that SUBSET-SUM is true if and only if BONNIE-CLYDE-CHECKS is true.

$\Rightarrow$: Assume that we have a subset S' of S that sums to t: If $t \geq T/2$, then all the integers that sum to t are given to Bonnie and c_{n+1} is given to Clyde with all the other integers. Our total is T + c_{n+1} and we subtract out the t we give to Bonnie to get T +c_{n+1} -t = 2t-t = t. Thus we have an equal partition, with the remaining integers of the set with c_{n+1} summing to t as well.

Otherwise if $t < T/2$, then all integers sum to t with c_{n+1} will be given to Bonnie and the rest to Clyde. This is an equal partition because t + c_{n+1} = t + T - 2t = T - t.

$\Leftarrow$: Assume that BONNIE-CLYDE-CHECKS is true. Then there exists a partition of the n checks and we know that c_{n+1} will either be with Bonnie or Clyde:

If c_{n+1} = 2t - T, then T + c_{n+1} = 2t and both Bonnie and Clyde's money must be equal to t. Thus Bonnie will have the integers that sum to t including c_{n+1} and Clyde will have just integers in S' that sum to t.

Otherwise, if c_{n+1} = T - 2t, then T + c_{n+1} = 2T - 2t = 2(T - t), and both Bonnie and Clyde will have checks that sum up to T - t. Thus c_{n+1} + t = T - 2t + t = T - t, and so Bonnie will have checks all from set S' and Clyde will have the rest of the integers and c_{n+1}.

0.4 d

This problem is NP-hard because part c reduces to it in polynomial time. As we are dealing with checks, they must have at least decimal units (e.g. we can assume that no check will be made out for a 1/3 of a penny). Multiply each check denomination by 10000 dollars, and feed these modified checks into the algorithm for d. If the algorithm for d returns a partitioning of the modified checks where the sums of either side is within 100 dollars, then return the equivalent partitioning for the original set of checks. If there was a partitioning of the modified checks, then the check sums in the original partitioning are within a cent of eachother, e.g. the check sums are equal.

We know that this problem is in NP because given a solution, a partitioning of the checks into two sets, we can easily check if the two sets sum to within 100 dollars in linear time.