CS 124 Data Structures and Algorithms — Spring 2015
Problem Set 8 Solutions

1 Problem 1

You are managing HBO GO's communications network N, which is comprised of a set of network towers T that can send information back and forth. Some towers can communicate with each other, and the maximum bandwidth (flow of data) allowed from tower u to tower v is described by $c(u, v) \geq 0$. N has a set of (not necessarily disjoint) tower pairs $P = (s_1, t_1)...(s_k, t_k)$, each of which demands at least d_i of bandwidth - s_i must be able to transmit a stream of d_i bits that t_i eventually receives. If the bandwidth needs are not satisfied your network will crash, Harvard students will unintentionally read spoilers on Buzzfeed before seeing the latest episode, and your company will tank!

Part 1 An enterprising CS 124 student, you initially think you can just maximize the total data flow across your network. Explain briefly (in 3 sentences or less) why this will not necessarily result in a satisfactory solution.

Part 2 Reformulate this problem as a linear program that will ensure Game of Thrones is streamed successfully. Make sure that each pair in P is allocated the appropriate bandwidth, and note that data transmitted from tower s_i is not identical to data transmitted from tower s_j. Define $f^i(\cdot, \cdot)$ as the data flow from s_i to t_i, and $f^i(u, v)$ as the amount of "i" data flowing from node u to node v. Find a solution such that maximizes y given that $f^i(\cdot, \cdot) \geq y \cdot d_i$ for all i.

Solution

Part 1 Solution

Just maximizing the total data flow across your network will result in uneven bandwidth and scrambled data. The optimal solution could have 500Mb/s between towers (s_i, t_i) and 0Mb/s between towers (s_j, t_j). Plus, data sent from s_i would be indistinguishable from data sent from s_j.

Part 2 Solution

This is known as the Fractional Multicommodity Flow Problem.

We need to make sure that our normal flow constraints are satisfied (positive flow between nodes, respecting capacity constraints). We'll be careful about distinguishing between i and j data when necessary. Let E denote the edge set in the graph. Note that (s_i, t_i) are considered source and sink nodes, but only for i data! For all other data types inflow must equal outflow. The linear program that describes the maximization problem is as follows:

$$\max y, \text{ s.t.} \tag{1}$$
$$\sum_{(s_i,v)\in E} f^i(u,v) \geq y \cdot d(s_i,t_i) \qquad \forall i = 1...k \tag{2}$$
$$\sum_{(u,v)\in E} f^i(u,v) = \sum_{(v,w)\in E} f^i(v,w) \qquad \forall i = 1...k,\ \forall v \in T\backslash\{s_i,t_i\} \tag{3}$$
$$\sum_{i=1}^{k} f^i(u,v) \leq c(u,v) \qquad \forall (u,v) \in E \tag{4}$$
$$f^i(u,v) \geq 0 \qquad \forall (u,v) \in E,\ \forall i = 1...k \tag{5}$$

Line (1) ensures that y is maximized. (2) ensures that we have enough bandwidth from s_i to t_i to satisfy the demand d (multiplied by y), as per the description in the prompt. (3) ensures that the flow of "i" data into a node v is equivalent to the flow of the same data type out of the node. Note that we do not sum over all k, since "i" data as distinct from "j" data - i data cannot turn into j data before flowing out of v. Additionally, this constraint need only be true for nodes who are not s_i and t_i - source and sink nodes for that data type. (4) ensures that the capacity of an edge is not exceeded by summing over all possible types of data that could flow through it. (5) ensures that flow of data over every edge is positive.

2 Problem 2

The class NP was defined as a class of decision problems. However, typically we have an optimization problem we would like to solve and not just a decision problem. For example, in the HamiltonianCycle problem we must decide whether a directed graph G has a simple cycle of length n. This is in contrast with the non-decision FindHamCycle problem of actually *finding* a cycle. Similarly in the VertexCover$_k$ problem we must decide whether a vertex cover exists of size at most k, as opposed to the MinVertexCover problem of finding a vertex cover of minimum size. Show that a polynomial time algorithm for HamiltonianCycle implies a polynomial time algorithm for FindHamCycle, and polynomial time algorithms for VertexCover$_k$ for all k imply a polynomial time algorithm for MinVertexCover.

Solution

Part 1 - Hamiltonian Cycle: If we have a polynomial time algorithm D to decide if $G \in$ HamiltonianCycle, we can use D as a black box to find a Hamiltonian path in G. To prove this, we will describe a polynomial time algorithm to do so.

Take a graph $G \in$ HamiltonianCycle with n nodes. Order all edges and initialize them as unmarked, then follow this algorithm until all edges are marked:

1. Delete the first unmarked edge e, giving the graph G'.

2. If $G' \in$ HAMILTONIANCYCLE, e wasn't necessary for a cycle, so we continue with G'.

3. If $G' \notin$ HAMILTONIANCYCLE, e is necessarily in G's Hamiltonian cycle, so restore the edge and mark it. We continue with G.

If we follow this algorithm until all edges are marked, we will be left with a graph G^* with the same set of nodes as G and n edges forming a Hamiltonian cycle. How do we know this?

- Each iteration of the algorithm leaves a graph with a Hamiltonian cycle, so $G^* \in$ HAMILTONIANCYCLE.
- G^* must have the same set of nodes as G, since our algorithm never delete nodes.
- Since each edge in G^* is marked, removing any of them results in a graph not in HAMILTONIANCYCLE. Thus, each is necessary for a Hamiltonian cycle. Since a Hamiltonian cycle has length n, there are exactly n edges in G^*.

This algorithm runs in polynomial time. A graph of size n has $O(n^2)$ edges. Our algorithm thus iterates $O(n^2)$ times. Each iteration requires checking our decider D, which runs in $O(n^c)$ time, once, for some constant c. Thus, the algorithm to find a Hamiltonian cycle runs in $O(n^{c+2})$ time.

Part 2: Vertex Cover Again, we will use a decider $D(G, k)$ that checks if G has a vertex cover of size k in polynomial time as a black box to find the smallest vertex cover of G.

First, check if G has a vertex cover of size $k = 1$ with the decider D. Increment k until we have found the size of the minimum vertex cover. Since we know that every graph has a vertex cover of size n, this process will terminate.

Once we know that G has a minimum vertex cover of size k, we can find the vertex cover using the following algorithm:

1. Choose an unmarked vertex v from the graph G with minimum vertex cover k. Delete v and all of its edges, leaving G'.

2. If G' has a vertex cover of size $k-1$, then v is in a minimum vertex cover of G. Continue the algorithm with G'.

3. If G' doesn't have a vertex cover of size $k-1$, v isn't in a minimum vertex cover of G. Continue the algorithm with G after marking v.

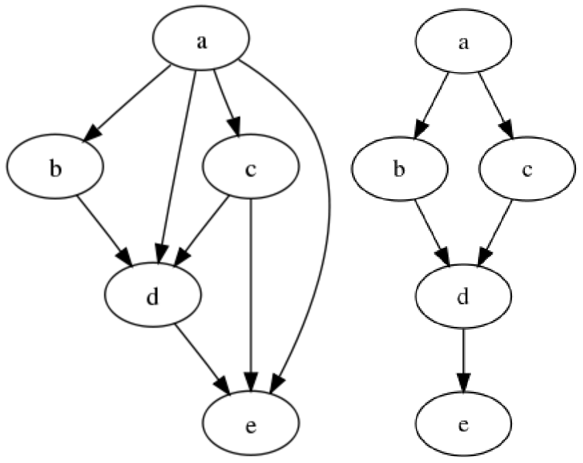
If we follow this algorithm, we will end up with a graph G^* with no more edges. All deleted nodes make up a vertex cover. If we have a graph G with a minimum vertex cover of size k, then deleting any node in the vertex cover (as well as its edges) must give us a graph with a minimum vertex cover of size $k-1$. However, deleting any other node and its edges does not change the size of our minimum vertex cover. Therefore, we delete all nodes in the vertex cover and their edges. By the definition of a vertex cover, we have no edges left.

This algorithm runs in polynomial time. We know that our decider D runs in time $O(n^c)$ for a constant c. We first check it at most n times, giving us $O(n^{c+1})$ operations. Then we run our main algorithm. Since we delete or mark every vertex, and we run D once for every vertex, we run D at most n times. Therefore, our overall algorithm runs in $O(n^{c+1}+n^{c+1}) = O(n^{c+1})$. Given D, we have a polynomial time algorithm for MinVertexCover.

3 Problem 3

(The origin of this problem was Homework 8, Exercise 3 of Dexter Kozen's book "The Design and Analysis of Algorithms".)

A transitive reduction of a directed graph is a graph with as few edges as possible that has the same reachability relation as the given graph. Said another way, a a graph G' is a transitive reduction of G, there is a path from vertex u' to v' if and only if there is a path from vertex u to v. Additionally, G' has the smallest number of edges possible. Here is an example of a graph and its transitive reduction:



Show that it is NP-complete to determine whether a given graph G has a transitive reduction with at most k edges. (The input G need not be acyclic.) You may use that the directed HamiltonianCycle problem is NP-hard when when the input graph is promised to be strongly connected.

Solution TransReduction is clearly in NP - given a graph G and a possible transitive reduction G', we can check in polynomial time that for any (u, v) pair, there is a path from u to v in G', if and only if there is also a path in G. Checking if a path exists between two nodes can be performed in linear time with a breadth-first search. If this check is successful, as long as G' has at most k edges, we'll know that G' is in fact a transitive reduction of G.

To prove that it's NP-complete, we'll reduce from the NP-hard HamiltonianCycle problem on directed strongly connected graphs.

A strongly connected graph has a Hamiltonian cycle if and only if it has a transitive reduction with at most n edges (proof below). Therefore, we can transform any HAMILTONIANCYCLE problem into a TRANSREDUCTION problem by asking if it has a transitive reduction with at most n edges. Thus, HAMILTONIANCYCLE reduces to TRANSREDUCTION. Since HAMILTONIANCYCLE is NP-hard and TRANSREDUCTION $\in NP$, TRANSREDUCTION must be NP-complete.

Proving that $\langle G, n\rangle \in$ TRANSREDUCTION $\rightarrow G \in$ HAMILTONIANCYCLE: In a strongly connected graph (and correspondingly, in its transitive reduction), there must be a path from each node to every other node. The transitive reduction must have at least n edges, since every vertex must have at least one incoming edge. If the transitive reduction has exactly n edges, they must be in the form of a perfect cycle - the only way n edges could create a strongly connected graph. If such a reduction exists, then it forms a Hamiltonian cycle of the original graph! Therefore, the original graph has a Hamiltonian cycle.

Proving that $G \in$ HAMILTONIANCYCLE $\rightarrow \langle G, n\rangle \in$ TRANSREDUCTION: As we said above, any transitive reduction of a strongly connected graph must have at least n edges. If a Hamiltonian cycle exists, it is a transitive reduction of our graph: it has the minimal number of edges (n) and maintains the connectivity of the original graph by ensuring every node is reachable from every other node. Thus, by finding a Hamiltonian cycle we've found a transitive reduction with at most n edges.

4 Problem 4

Consider the two-player game given by the following matrix. (A positive payoff goes to the row player.)

$$\begin{bmatrix} 4 & 1 & 0 & -3 \\ 6 & -3 & -2 & 0 \\ -3 & -2 & 5 & -3 \\ -4 & 4 & -5 & 5 \end{bmatrix}$$

- Write down the linear program to determine the row player strategy that maximizes the value of the game to the row player. Do the same for the column player.
- Find an LP solver and use it to solve these linear programs, and give the proper strategies for both players. There are many online you can use, but we used `http://www.phpsimplex.com/simplex/simplex.htm?l=en`. Note that on many solvers, all decision variables are assumed to be positive - make sure your variables respect this assumption when plugging them into the solver!
- What is the value of the game? Should the column player pay the row player to play, or vice versa, and how much should one player pay the other to make the game fair?

Solution As per the lecture notes, we adopt the convention that if $P_{i,j}$ is positive in the payoff matrix P, then the row player is paid by the column player when the row player plays strategy i and the column player plays strategy j. A negative number means the row player pays the column player.

The row player's space of strategies includes picking row i with probability x_i for $i = 1, \ldots, 4$. Being probabilities, they should be nonnegative and sum to 1. Given any fixed strategy (x_i), the column player has an optimal strategy which is simply to always pick a certain column (the one that maximizes his own payoff). Similarly the column player needs to pick probabilities (y_i) to minimize the amount he pays out to the row player. The row player can guarantee himself a certain amount of profit by always sticking with the same row, and a pure row strategy is optimal for any choice of (y_i). Thus adopting the notation of the lecture notes on games, the linear program to compute the optimal strategy for the row player is

$$\begin{aligned} \max z & \\ s.t. & \\ z - 4x_1 - 6x_2 + 3x_3 + 4x_4 &\leq 0 \\ z - x_1 + 3x_2 + 2x_3 - 4x_4 &\leq 0 \\ z + 2x_2 - 5x_3 + 5x_4 &\leq 0 \\ z + 3x_1 + 3x_3 - 5x_4 &\leq 0 \\ x_1 + x_2 + x_3 + x_4 &= 1 \\ \forall i \in \{1, \ldots, 4\}, x_i &\geq 0 \end{aligned}$$

For the column player it is

$$\begin{aligned} \min z & \\ s.t. & \\ z - 4y_1 - y_2 + 3y_4 &\geq 0 \\ z - 6y_1 + 3y_2 + 2y_3 &\geq 0 \\ z + 3y_1 + 2y_2 - 5y_3 + 3y_4 &\geq 0 \\ z + 4y_1 - 4x_2 + 5x_3 - 5x_4 &\geq 0 \\ y_1 + y_2 + y_3 + y_4 &= 1 \\ \forall i \in \{1, \ldots, 4\}, y_i &\geq 0 \end{aligned}$$

We used `http://www.phpsimplex.com/simplex/simplex.htm?l=en` to solve the linear program. Since the LP implicitly has the constraint that all variables are nonnegative, we used a variable w to represent $-z$. The optimal strategy for the row player (with decimal values in parentheses) is:

$$z = -145/1198 \ (-.121)$$

$$x_1 = 93/599 \ (.155)$$
$$x_2 = 275/1198 \ (.230)$$
$$x_3 = 409/1198 \ (.341)$$
$$x_4 = 164/599 \ (.274)$$

That is, if the row player plays optimally then he has to pay the column player $145/1198 = .121035058\ldots$.

For the column player, the optimal strategy has payoff $145/1198 = .121$ and is achieved by the following probabilities:

$$y_1 = 104/599 \ (.174)$$
$$y_2 = 193/1198 \ (.161)$$
$$y_3 = 407/1198 \ (.340)$$
$$y_4 = 195/599 \ (.326)$$

Thus, if both players play optimally then the row player has to pay the column player $145/1198$ in expection. For the game to be fair, the column player should pay the row player this much beforehand.

5 Problem 5

Solve "Problem A - Airplanes" on the programming server; see the "Problem Sets" part of the course web page for the link.

Solution We create an algorithm to decide if all of the people can be moved out in t steps. Given this decision function, we can increment t to find the shortest time in which all the people can move from East Coast cities to West Coast cities.

Given $G = (V, E)$, construct G_t based on time t. For each $v \in V$, make t copies of v: $v_1, \ldots, v_t$. These t extra nodes represent the existence of the city at each time step. A person can choose to stay in the city, represented by constructing an edge from v_i to v_{i+1} for all sequential nodes with infinite capacity. Alternatively, a person can board a flight from the city to any connecting city each hour. In the case where all of our flights take 1 hour, we can construct an edge from v_i to w_{i+1} with capacity C if there exists an edge from v to w with capacity C in G. To test if all the people can get from the source to the sink in t timesteps, we can check if the max flow in G_t is equal or greater to the number of people initially at the source. We can find the max flow using the Ford-Folkerson.

For the cases where flights take integer time steps, we create edges between the layers differently. Instead, we can construct an edge from v_i to w_{i+L} with capacity C if there exists an edge from v to w with capacity C in G and flight length L.

For cases where there are more than one East Coast city or West Coast city, we can create a source s and a sink t. For all of the East Coast cities, we can create an edge from the source s to each of the East Coast cities with capacity equal to the number of people who start at that city. For each West Coast city, we can create an edge from the West Coast city to the sink with infinite capacity.