CMPS 102 Homework 3: Network Flow

Benjamin Gordon / bcgordon@ucsc.edu

5/27/2014

1 Problem 1: Dining Cows

1.1 Algorithm

First, we assume that $f > n$ and $d > n$, otherwise there can be no perfect matching. Then, we create two bipartite matching networks; one with cows on the left and dishes on the right, and one with cows on the left and drinks on the right. Each cow has an edge coming into it from the source, and each dish and drink have an edge going from it to the sink. The capacity of each edge in either network is 1. We start with just one of the two networks (which one you choose is arbitrary). Then, perform Ford-Fulkerson's Maximum Flow algorithm on that network; if the maximum flow is equal to the number of cows, then we have found a perfect matching. Otherwise, terminate the algorithm; it is impossible to give every cow a complete meal. Then, simply repeat this process for the other network. If both networks have maximum flow equal to the number of cows, then it is possible to give every cow a complete meal.

1.2 Proof of Correctness

Ford-Fulkerson's algorithm is proven to find the maximum flow between the source and the sink for a network flow problem. Since every edge has capacity 1, the maximum amount of flow through a graph is bounded by the number of edges leaving the source, which is equal to the number of cows. This is why we know we have a perfect matching if the maximum flow is equal to the number of cows; it means every cow has found a corresponding drink/dish. Since each cow's choice of drinks does not affect their choice of dish, and vice versa, we can treat the problem as two separate subproblems. If both networks have a maximum flow equal to the number of cows, then every cow must have both a dish and a drink corresponding to their preferences, and the problem is therefore solved.

1.3 Proof of Runtime

Ford-Fulkerson's algorithm performs a series of augmentations, and terminates when no further augmentations are possible. Therefore, the runtime of this

algorithm is bounded by: (upper bound of number of augmentations) $\times$ (upper bound of runtime of a single augmentation). We can use either BFS or DFS to find augmenting paths; this has the runtime $O(|V| + |E|)$. We can then say that, since the capacities are all integer values, and each augmenting path increases the amount of flow through the network by at least 1, there will need to be at most F augmentations, where F is the maximum amount of flow through the network. F is also equal to n, the number of cows being considered. Thus we have a runtime of $O(n \times (|V| + |E|))$.

2 Problem 2: Outsourcing

Note: I discussed this problem with Jordan Hyman, but it did not provide any insights. I later discussed it with Joshua Nussbaum and Evan Hughes, but the only insight I gained was that I was on the right track already.

2.1 Algorithm

We will set up this problem as a bipartite network with some intermediate nodes. On the left side, we have the cows; the source has an edge connecting it to each cow, and each of these edges has a capacity of 3. On the right side, we have the projects; each project has an edge connecting it to the sink, and each of these edges has capacity k. We also have some intermediate nodes; each project has C intermediate nodes, where C is the total number of colors of cows. Each of these intermediate nodes has an edge flowing into its corresponding project, with capacity $k - 1$. Each cow has an edge connecting it to each intermediate node that corresponds both to the cow's color and the projects that it would be interested in working on; each of these edges has capacity 1. Then we can simply run Ford-Fulkerson's algorithm on this bipartite graph; if the flow into the sink is equal to $k \times P$, where P is the number of projects, then we have successfully found a matching which fulfills all our constraints. Let us define such a matching to be an "optimal solution."

2.2 Proof of Correctness

To begin with, assume $k < \frac{m}{n}$, where m is the number of cows and n is the number of projects; otherwise, there can be no optimal solution, because there simply are not enough cows. Ford-Fulkerson's algorithm is proven to find the maximum flow between the source and the sink for a network flow problem. Given this, I will prove the algorithm's correctness by explaining why this layout forces the maximum flow to comply with all 4 of the given constraints.

- Each cow can only work on problems in its set P_i.
  - Since each cow only has edges to the projects in its list, it can only end up working on projects in said list.

- Each cow can work on at most 3 projects.
  - Each cow gets only 3 units of flow, each edge leaving the cow has capacity 1, and flow must be in decimal values (because you can't program with a fraction of a cow). Therefore, each cow will work on at most 3 projects.
- There must be at least 2 different colors of cows working on every project.
  - Because the capacity of the edges from project $\rightarrow$ sink is k, but the capacity of the edges from color $\rightarrow$ project is $k-1$, in order for each project to have k flowing into it, there must be more than one color of cow working on it.
- There must be at least k cows working on each project.
  - Once a project has reached k cows (since we've already proven that it must have at least 2 colors of cow working on it by that point), there is no reason to ever add another cow to it. Thus, each project $\rightarrow$ sink edge has capacity k, in order to prevent cows working on a project that already has all of its constraints fulfilled. Based on this, our assumption that $k < \frac{m}{n}$, and the proven correctness of Ford-Fulkerson's algorithm, we can say that if this constraint is not fulfilled, it is because there is no optimal solution.

2.3 Proof of Runtime

Ford-Fulkerson's algorithm performs a series of augmentations, and terminates when no further augmentations are possible. Therefore, the runtime of this algorithm is bounded by: (upper bound of number of augmentations) $\times$ (upper bound of runtime of a single augmentation). We can use either BFS or DFS to find augmenting paths; this has the runtime $O(|V| + |E|)$. We can then say that, since the capacities are all integer values, and each augmenting path increases the amount of flow through the network by at least 1, there will need to be at most F augmentations, where F is the maximum amount of flow through the network. F is also equal to $3n$, the number of cows being considered $\times$ the number of jobs each cow can take. Thus we have a runtime of $O(3n \times (|V| + |E|))$.

3 Problem 3

Note: I discussed this problem with Steven Chou, who gave me the idea of how to set up the network flow.

3.1 Part A

An example of a non-rearrangeable matrix is: $\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$ Because no matter how you swap rows and columns, two of the 1's will always share a row, and the other two will always share a column.

3.2 Part B

3.2.1 Algorithm

Set up a bipartite matching with each row being a node on the left, and each column being a node on the right. The source has edges to each row, and each column has an edge to the sink. If the matrix entry represented by the intersection of row R_i and column C_j is a 1, then there is an edge between R_i and C_j. All edges in the graph have capacity 1. Now run Ford-Fulkerson's algorithm on this network; if the maximum flow is n, then the matrix is rearrangeable.

3.2.2 Proof of Correctness

Consider the fact that any 1's in an nxn matrix which share a row, must always share a row despite any number of row or column swaps. Why is this? Consider that row swaps simply rearrange the row in relation to the other rows, without changing the contents of any row; also, column swaps can rearrange the order of entries in a row, but they cannot move any entries into a different row. The inverse of this also holds for 1's which share a column. If the matrix is rearrangeable, it must have at least n 1's which do not share a vertex or column with each other. Otherwise, based on the above consideration, there is no way to rearrange the matrix such that there are 1's on the diagonal. However, if there are n such 1's in the matrix, then the matrix must be rearrangeable, because no amount of swaps will change the condition that there are n 1's which do not share a row or column. Now we must prove that if the maximum flow is equal to n, then there are n 1's that do not share a row or column with each other. Our graph is set up such that any 1's which share a row will be represented as multiple edges leaving that row; similarly, any 1's which share a column will be represented as multiple edges entering that column. However, since each edge has capacity 1, in order to have a max flow of n, there must be at least 1 edge leaving each row and at least 1 edge entering each column. This arrangement represents the case in which there are n 1's which do not share a row or column. Therefore, max flow $= n$ implies that there are n 1's which do not share a row or column, which in turn implies that the matrix is rearrangeable.

3.2.3 Proof of Runtime

Ford-Fulkerson's algorithm performs a series of augmentations, and terminates when no further augmentations are possible. Therefore, the runtime of this

algorithm is bounded by: (upper bound of number of augmentations) $\times$ (upper bound of runtime of a single augmentation). We can use either BFS or DFS to find augmenting paths; this has the runtime $O(|V| + |E|)$. We can then say that, since the capacities are all integer values, and each augmenting path increases the amount of flow through the network by at least 1, there will need to be at most F augmentations, where F is the maximum amount of flow through the network. F is also equal to n, the number of rows/columns in the matrix. Thus we have a runtime of $O(n \times (|V| + |E|))$.

4 Problem 4

4.1 Part A

Set up the problem as a matrix, with each row representing a set and each column representing a driver. From there, we can adapt that matrix into a bipartite network flow, and solve the feasible rounding from there. There must be a feasible rounding, because the integrality theorem states that if there is a solution to a network flow problem, then there must be an integral solution to it as well.

4.2 Part B

4.2.1 Algorithm

First, Set up the problem as a matrix, with each row representing a set of people and each column representing a person. Each member in the matrix is that person's likelihood that they drive on that given day. Have the sums of each row and column represented on the right and the bottom, respectively. Then adapt this matrix into a bipartite network flow. Have each person represented as a node on the left and each day represented as a node on the right. The source has an edge to each person; that edge's maximum capacity is equal to the ceiling of that person's total driving obligation, and its minimum flow requirement is equal to the floor of that person's total driving obligation. Each person has an edge to a day if that person is going to carpool on that day; each of these edges has capacity 1. Finally, each day is connected to the sink by a node with capacity 1. Run Ford-Fulkerson's algorithm on the resulting graph, and when it is complete, the people will all have flows connecting to the days they should drive.

4.2.2 Proof of Correctness

This is essentially a feasible rounding problem, as we went over in class. By the integrality theorem, any network flow with capacities which are all integers MUST have a maximum flow which is integer-valued and in which every value is an integer. Each person will receive flow units equal to either the floor or ceiling

of their total obligation, therefore each person must be driving on a number of days which is an acceptable rounding of their obligation.

4.2.3 Proof of Runtime

We are essentially just using Ford-Fulkerson's algorithm, which has runtime $O\left(|E|^2 \times log\left(C\right)\right)$, where E is the number of edges and C is the largest arc capacity. This is polynomial in k and d because E is equal to the sum of the number of days each person carpools, and C is equal to the ceiling of the largest total driving obligation from our set of drivers.

5 Problem 5: Anything For A Good Bath

5.1 Algorithm

In order for there to be a solution to this problem, there must be an edge in the input graph which is single-handedly preventing an additional augmenting path from existing; all that would be necessary to find an additional augmenting path is to upgrade that edge. Therefore, in order for a solution to exist, there must be at least $d-1$ edges in the final input graph which have not reached capacity, where d is the minimum depth from source to sink, because an augmenting path must have depth at least d. Consider only those edges in the residual graph with 0 remaining capacity, i.e. edges which have reached capacity in the input graph. For each of these edges in the residual graph, do two things: traverse the graph forward, considering only edges leaving each node with capacity > 0; and traverse the graph backward, considering only edges entering each node with capacity > 0. As you traverse this graph in this way, remove each edge with capacity > 0 from the list of edges to consider - this way we only visit each edge in the graph once. If both the source and the sink cannot be reached in this way from any edge with capacity 0 in the residual graph, there is no solution to the problem. However, if from such an edge s in the residual graph, both the source and the sink can be reached by this traversal, considering only edges with capacity > 0, then edge s is a solution.

5.2 Proof of Correctness

By traversing the graph forward and backward from each edge in the residual graph with capacity 0, we are finding out whether or not each such edge would be a member of an augmenting path if its capacity were upgraded. Since we only consider forward paths with capacity > 0 during this traversal, we can be sure that if both the source and the sink can be reached, that the edge we are considering is the sole bottleneck in what would otherwise constitute an augmenting path. And since we are considering every edge with capacity 0 in the residual graph, we know that if such a bottleneck exists, we will find it. Also, as we traverse each edge, if it has capacity > 0, we prevent it from being considered in traversals in the future. We can be sure that this will not

affect future iterations of this algorithm from finding a solution, because if we find another edge with capacity 0 in the residual graph during our traversal, then we know the current edge being considered is not the only bottleneck on this potential augmenting path; we don't want to waste time reconsidering this augmenting path in the future.

5.3 Proof of Runtime

This proof is very simple: since we prevent each edge with capacity > 0 from future consideration as we traverse it, each edge in the graph can be visited at most once. Therefore, this algorithm has runtime $O(|E|)$, where E is the number of edges in the graph.