CS180, Winter 2018
Homework 7
Problems: 7.11; 7.14; 7.17; 7.29

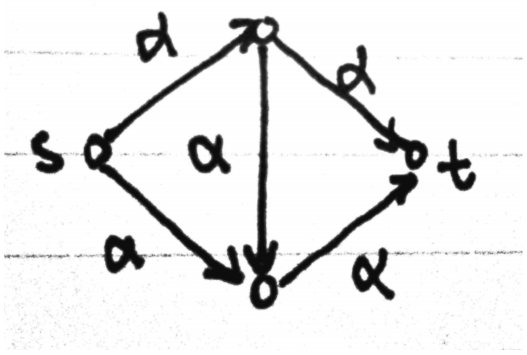
Derek Xu

March 9, 2018

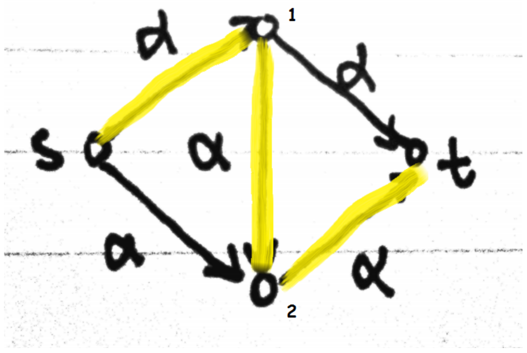
1 Problem 7.11

The statement is false. To see why, we will try to find a counter example. In order for the statement to be false, we must find a max flow that is greater than $1/b$ times the result of the friend's algorithm, for b as an arbitrary constant. Thus, we can assume that we have a max flow of $(b+\gamma)\alpha$, where $\gamma > 1$, and the algorithm returns a flow of α... $\frac{b+\gamma}{b}\alpha > \alpha$. If such is the case, there must be a path for that flow, or in other words the flow through the saturated minimum cutset of the graph is also $(b+\gamma)\alpha$. We must find a graph such that the algorithm returns α flow through this cutset.

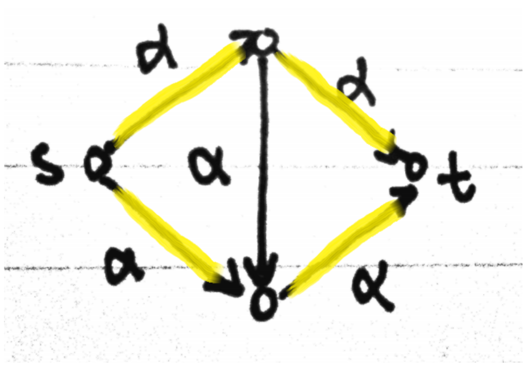
Before continuing further, let us first see the problem introduced by not finding reverse edges. Consider the following graph:



All the edges in this graph is of cost α. Because the algorithm chooses arbitrary paths. The following may be a path that it chooses:

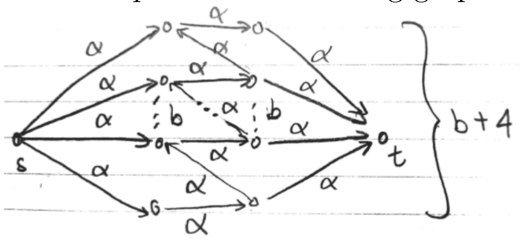


If such is the case we can see that the friend's algorithm can no longer choose any more paths because the edge $(2,t)$ is saturated. Thus, this is the flow that the friend's algorithm will return. Now consider the proper answer, through inspection:

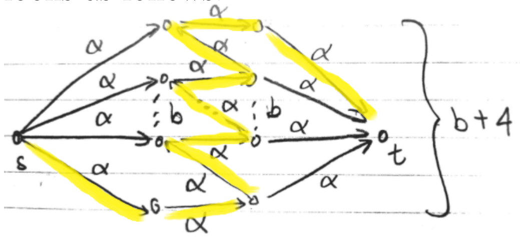


We see that in this particular case the max flow is 2α, but the friend's algorithm returns α.

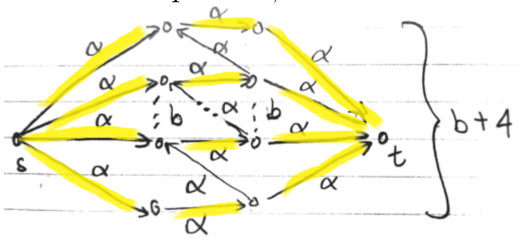
Following this example, we must simply create a graph where a certain $b+1$ edges that should be in the max flow are being saturated by a bad choice of path. The following graph is one such example:



Let the friend's algorithm choose a path such that the graph returned looks as follows:



From inspection, the max flow should be the following:

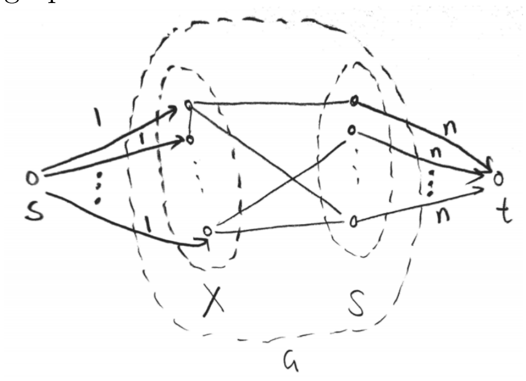


Notice, the friends algorithm returns a graph of flow α, but the max flow returns a flow of $(b+4)\alpha$. This validates our condition above with $\gamma = 4$.

2 Problem 7.14

(a) We connect a starting node, s, to each of the nodes in set X with an edge of cost 1. We then connect an end node t, to each of the nodes in set S with an edge of cost n, where n is the total number of nodes in set X. We run max flow from s to t, and if a flow exists of size n then that means all of the edges connected to the starting node s is saturated, and thus there exists a path from every nodes in X to S.

Proof of Correctness:
If there exists paths from the populated nodes, X, to the safe nodes, S, the all of the populated nodes need to flow to the safe nodes. The populated nodes derive their flow from the starting node, s, each with saturated flow, 1. This constitutes the minimum cutset. The final node, t, should therefore have a flow of n, where $n = |X|$. We connect a edge of capacity n to t from each of nodes in S to accomplish this. Thus, the flow must go from X to S to reach t. The flow to the terminal node, t, shows how many flows from X went to each safe node in S, which again should sum to n. We can see this better in the following graph:

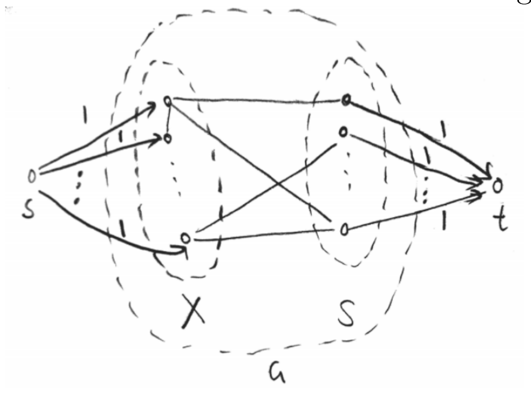


Proof of Time Complexity:
We can construct the graph in polynomial time by simply adding edges to each node and adding s and t. Max flow is also a polynomial time algorithm. Thus, the total runtime is polynomial: O(n) + O(mn) = O(mn).

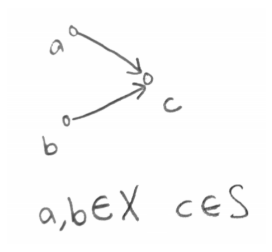
(b) We notice this problem is similar to that of bipartite matching. For all nodes in X to reach all nodes in S with no intercepting nodes, each

node in X must reach a unique node in S.
If $|S| < |X|$, there cannot exist a path such that each node in X reaches a unique node in S by bipartite graph properties from discrete math. Thus, we first check this condition is false, if not we say that no such matching exists.

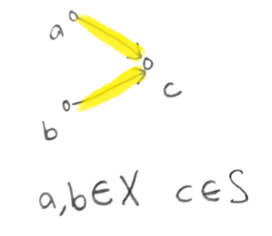
If such is false, we apply the following:
We create a start and end node s and t. We connect the start node s to every node in X with an edge of cost 1, and the end node t to every node in S with an edge of cost 1. We then try to find the max flow that pushes everything from X to S. If a flow of size n exists, where $n = |X|$, that means all of the edges connected to X is saturated and thus there exists a path from every node in X to S (same as above). We can see this better in the following graph:



Proof of Correctness:
We are following the same reasoning as in part A, but here all of the edges connected to t are of cost 1, meaning that each node in S can correspond to only one node in X, which is what we want since no path can share nodes. Another way to say this is we are finding a loose version of bipartite matching where all of the nodes in X must be matched to one in S but not vice versa.
Proof of Time Complexity:
We can construct the graph in polynomial time by simply adding edges to each node. Max flow is also a polynomial time algorithm. Thus, the total runtime is polynomial: O(n) + O(mn) = O(mn).
The following graph shows a contradiction between a and b:



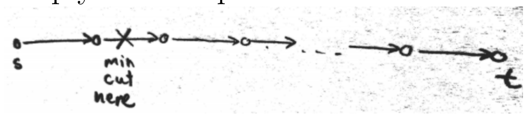
The following path exists for a:



No path exists for b, because the $|X| > |S|$.

3 Problem 7.17

First, let us clarify the problem... We are only restricted by O(klogn) calls to ping. In other words, we are NOT restricted in total running time. Before considering the general case, let us first observe the case, where the graph is simply a linear path from s to t:



Note, since the hackers destroyed the minimum number of edges needed to connect s to t, they in effect destroyed the minimum cutset (see textbook section on 'Min-Cut is Max Flow' for proof of this). In our special case, the cutset can be any edge, thus we simply perform a linear scan from s to t, to find the minimum cutset. In other words, we ping the index which is 1/2 the length of the path and if it is connected we ping the 3/4 mark, else we ping the 1/4 mark. We continue to do this until we iterate the index up or down by 1 and the ping returns different from the previous ping. This is in effect a pseudo-binary search across the path to find where the cutset is. Correspondingly, the number of pings is O(logn), as we are recursively locating where the missing edge is by dividing the search space in half each iteration.

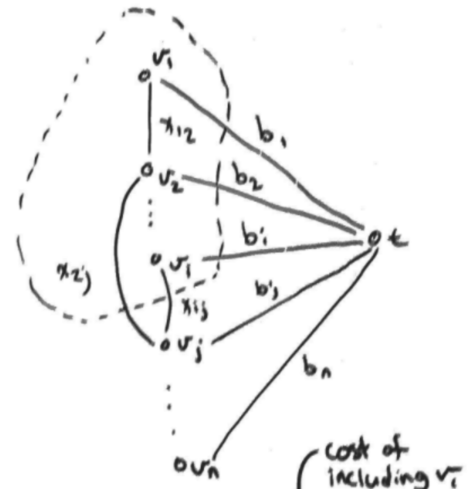
We now generalize the case. Let us assume we find the k unique paths from s to t. We simply run the same algorithm above on each of the k paths. The number of pings used will be O(klogn) since we use O(logn) pings on the O(k) paths.

The question now is how do we find the k paths, and how we locate the middle element of the path. To find the k paths we can simply run the max flow algorithm, which runs in polynomial time, and choose k arbitrary paths that are unique in the parts of the minimum cutset that is included within the path. Since we have the paths, it is easy to keep track where the middle of that path is. Thus, we simply run the max flow algorithm O(mn) to find the paths, and then the pseudo-binary search ping algorithm O(klogn) on those paths.

4 Problem 7.29

In max flow, we are trying to maximize flow between 2 points. In this problem we are trying to maximize benefit. Thus when modeling the problem as max flow we let the edges denote the benefit. The question now is how to impose the parameters of benefits (easy just let b_i be the cost of the edge) and the expenses (not so trivial).

After a lot of deliberation, I realize the fundamental issue: having a black box that imposes a condition that outputs flow based on if the input if 1 of 2 or 2 of 2 flows is difficult (without some external "device"). Thus we alter our strategy to finding minimum cut. The minimum cut will find the minimum cost of edges forming a cutset between two subsets in the graph. In our problem, we are trying to maximize the benefits from softwares we choose and minimize the costs from the softwares we do not choose. This is the same as saying we are trying the minimize the benefits and costs of the softwares we do not choose. We must now form a graph from this. Observe the following graph:



$$\min\Big(\underbrace{\sum_{V'} b_i}_{\text{benefit of including } v_i} + \overbrace{\sum_{V,V'} x_{jk}}^{\text{cost of including } v_i}\Big)$$

$v_i \in V'$

$v_i \notin V$

We run the max flow or min cut algorithm from node s to node v_1. This graph find the minimum of $\sum_{i \epsilon V'} b_i + \sum_{i \epsilon V, j \epsilon V'}$, where V' is the set seperated by the cutset including node v_1 and, V the other subset. Basically, we are finding a cutset that minimizes what we are looking for above. In this case 1 subset includes v_1 which was mentioned in the problem statement as not being ported. Thus the subset formed from the cut set that includes v_1 and not t is the subset of softwares we choose.

More rigorously we define the graph as follows:
Let every software be denoted by a node v_i. Let s be the starting node. Link s to the node v_i with an edge of cost b_i. Link v_i to v_j with an edge of cost x_{ij}.
As proved in the textbook, the minimum cut is equal to the maximum flow. Thus, we are simply run max flow on the constructed graph to get the correct answer.

Proof of Correctness:
See above.

Proof of Time Complexity:
We can construct the new graph in polynomial time, by representing the softwares with nodes and adding edges. We can find the minimum cut in polynomial time. Thus the total runtime is polynomial: O(n+m) + O(mn) = O(mn).