Jason Lai
204995126
Dis 1B

CS 180
May 30, 2019

Homework 7

1 Problem 7.11

Decide whether you think this statement is true or false, and give a proof of either the statement or its negation.

> *There is an absolute constant $b > 1$ (independent of the particular input flow network), so that on every instance of the Maximum-Flow Problem, the Forward-Edge-Only Algorithm is guaranteed to find a flow of value at least $1/b$ times the maximum-flow value (regardless of how it chooses its forward-edge paths).*

The statement is false and we show this by showing that the *Forward-Edge-Only Algorithm* will not return a flow value of at least $1/b$ times the maximum-flow value for some constant $b > 1$, where this constant b is the same for any arbitrary instance of the *Maximum-Flow Problem.*

Suppose there is a directed graph G such that there are $2(b+1)$ internal vertices u_1, u_2, ..., u_{b+1} and v_1, v_2, ..., v_{b+1} and a source node s and sink node t. For every node u_1, u_2, ..., u_{b+1} and v_1, there is an edge coming from the source s, and for every node v_1, v_2, ..., v_{b+1}, there is an edge going to the destination t. Each pair of nodes (u_i, v_i) is connected by an edge, and each pair of nodes (v_i, u_{i+1}) is connected by an edge. The maximum capacity of each edge is 1.

Now consider an augmented path that starts from the source s and ends at sink t, visiting each node in the order s, u_1, v_1, u_2, v_2, u_3, ..., u_{b+1}, v_{b+1}, t. Here, we see that the flow of such a path is only 1, as the *Forward-Edge-Only Algorithm* only takes forward-edge only paths. However, from this graph, it is possible to achieve a maximum flow of $b+1$ by using all edges available between pairs of nodes (u_i, v_i), so the proposed "zig-zag" path from source s to sink t does not have a flow within $1/b$ times the maximum flow $b+1$, therefore the statement is false, with proof by counterexample.

Homework 7

Jason Lai
204995126
Dis 1B

2 Problem 7.14

(a) *Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.*

(b) *Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus we change (iii) to say "the paths do not share any nodes."*
With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.
Also, provide an example with the same G, X, and S, in which the answer is yes to the question in (a) but no to the question in (b).

a) To show if such a set of evacuation routes exists in graph G, when given G, X, and S, we construct a network flow problem. First, we assign unit capacities to all existing edges in the graph G. Next, we introduce a new source node s and connect all "populated" nodes in the set X to the new node s; all edges connecting the nodes in X to s have a unit capacity. Finally, we introduce a new sink node t and connect all "safe" nodes in S to the new node t; all edges connecting the nodes in S to t have a capacity equal to $|X|$. We call this new graph *G'* with the additional edges and nodes.

Now that a network flow problem has been constructed, to find if a valid set of evacuation routes exists, we use a maximum flow algorithm like *Dinic's Algorithm* to find the maximum flow of the new graph *G'*. If the maximum flow is found to be equal to $|X|$, then we say that such a set of evacuation routes that meets the given constraints exists; inversely, if the maximum flow is found to be not equal to $|X|$, then we say that there is no such set of evacuation routes that exists.

This algorithm runs in polynomial time as we see that constructing the new graph *G'* takes $O(\text{E}+|X|+|S|+2)$ time (to add the capacities, the nodes s and t, and draw the edges), running a max-flow algorithm like *Dinic's Algorithm* will take $O(V^2E)$ time, and comparing the resulting maximum flow to $|X|$ will take $O(1)$ time. So, the runtime for this algorithm is $\approx O(V^2E)$.

b) With the new (*iii*) constraint, we still use the same constructed network flow problem from **a)**, however, we use the *Ford-Fulkerson Algorithm* to, instead, count the unique paths from each node in X to the nodes in S. Once a path from a node in X to a node in S is found, remove all nodes and edges used in the path; if removing a node results in edges with only one node connected to them, then remove these edges as well. Perform this operation recursively, and if all nodes in X are exhausted, then that means that there is such as set of evacuation routes such that none of the paths for each node in X shares a node.

The running time for this algorithm is polynomial, but is dependent on the *Ford-Fulkerson Algorithm*, which means the algorithm runs roughly in $\approx O(max\_flow * E * |X|)$ time, as we recursively call the *Ford-Fulkerson Algorithm* for each of the nodes in X.

Homework 7

Jason Lai
204995126
Dis 1B

3 Problem 7.17

Give an algorithm that issues a sequence of ping commands to various nodes in the network and then reports the full set of nodes that are not currently reachable from s. Give an algorithm that accomplishes this task using only O(klogn) pings.

An algorithm that issues a sequence of ping commands to various nodes in the network and then reports the full set of nodes that are not currently reachable from s in $O(k\log n)$ time is as follows:

We apply a divide and conquer approach, like a binary search, to find the full set of nodes that are not currently reachable from s. Since every node is on at least one path from s to t and the minimum number k edges were cut so that no path from s to t now exists, we can say that there were at most k paths severed in the original graph.

First, we have to obtain these k paths from the original, unsevered graph, by recursively running the *Ford-Fulkerson Algorithm* to obtain unique paths from s to t. After these k paths have been found, we use the monitoring tool provided by the network administrators to find all nodes not currently reachable from s. To do so, we use a binary-search-like algorithm, wherein, we take one of the k paths not yet explored and divide it in half, starting at the middle node of the path. From here, we ping this middle node to see if it is reachable from s. If it is, then we move to the midpoint of the upper half of this path; if it's not, then we move to the midpoint of the lower half of this path. We continue to ping this way, by divide and conquer, until we find the very last node in the path that is reachable, and we know all earlier nodes in the path are reachable, so we can then store these reachable nodes in a set S. We repeat this process for all k paths until we have a final set S which stores all reachable nodes in the severed graph.

If we store the reachable nodes in the set S in a data structure like a hash table, we can then take the original, unsevered graph and traverse all nodes in it using a breadth-first search or depth-first search and check for any nodes that aren't in our hash table. If a node is not found in our hash table (set S), then we add it to another set *S'*; after the entire graph has been traversed, we then return this set *S'* as the required, full set of nodes that are not currently reachable from s in the severed graph.

We can see that this algorithm uses only $O(k\log n)$ pings, since we use the pings in a divide-and-conquer algorithm similar to binary search, which is $O(\log n)$; we apply this algorithm on k paths, hence we have $O(k\log n)$ pings.

Jason Lai
204995126
Dis 1B

CS 180
May 30, 2019

Homework 7

4 Problem 7.29

Which of the remaining applications, if any, should be moved? Give a polynomial-time algorithm to find a set $S \subseteq$ 2, 3, ..., n for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized.

A polynomial-time algorithm to find a set S for which the sum of the benefits minus the expenses of moving the applications in S to the new system is maximized is as follows:

We construct a directed graph G such that there is a sink node t representing the new system and internal nodes v_1, v_2, ..., v_n representing the collection of n software applications. From every internal node, except v_1, we connect an edge that flows from the internal vertices to the sink t, where the weight of all of these edges is the benefit b_i of moving each respective application v_i. Between every pair of internal nodes (v_i, v_j) representing applications that have an interaction, connect them with an edge of weight x_{ij} representing the expense to port one of the applications, but not the other.

To get the set S for which applications we should move to the new system, we use the *Ford-Fulkerson Algorithm* for finding maximum flows to find the minimum cut of the graph G, which we know can be done by the max-flow min-cut theorem. From this, we get the desired set S by computing the v_1-t minimum cut of G, which will yield the maximum flow, and by extension the maximum benefit, possible for moving applications from the old system to new system. This works as we find the minimum cut to be the smallest possible difference between expense minus benefit (expense - benefit), which is effectively minimizing the overall expense of moving applications that are in the collection of n applications, which in turn, is maximizing the benefit, thereby finding all required applications for the desired set S.

This algorithm runs in polynomial-time (dependent on *Ford-Fulkerson*) as it takes $O(n+1)$ to draw the nodes, $O(E)$ to draw the edges, and $O(max\_flow*E)$ to run the *Ford-Fulkerson Algorithm*. So, the final running time of this algorithm is $O((max\_flow+1)E + n + 1) \approx O(max\_flow*E)$.