CSOR 4231
Homework #5 Solution

April 21, 2015

1 Problem 1

(a) We can prove this using the accounting method introduced in part 17.2 in textbook. Here we charge the operations as follows:

- Setting a bit from 0 to 1: \$3, while the actual cost is \$1
- Resetting a bit from 1 to 0: \$0, while the actual cost is \$1

So, whenever a bit is set to 1, the bit 'carries' a credit of \$2 (i.e. 3 -1). Out of this a credit of \$1 can be used for resetting the bit to 0 (during the INCREMENT operation). Each bit that has been set or reset, still carries a credit of \$1, which can be used in the RESET operation. Thus the remaining credit after a sequence of n INCREMENT or RESET operations is always bigger or equal to 0. So we can conclude that starting from an initially zero counter, any sequence of n INCREMENT and RESET operations takes time $O(n)$, that is, the amortized time per operation in $O(1)$.

(b) We can create a tree of height $\Omega(logn)$ by $n-1$ UNIONs: : first UNION(x_1, x_2), UNION(x_3, x_4),. . . , UNION(x_{n-1}, x_n) of size 2; then create sets of size 4 by pair-wise UNION of these, etc. This gives a binomial tree, which has $\binom{k}{i}$ of its 2^k nodes at depth i, for $i = 0, .., k$. Hence at least half of the n nodes are at depth $\geq (logn)/2$, so each FIND on these nodes takes $\Omega(logn)$ time. Letting $m \geq 3n$, we can have more than $m/3$ FINDs, so the total cost is $\Omega(mlogn)$.

2 Problem 2

This problems is similar to the minimum spanning tree. Instead of wanting to minimize the value of each edge, instead we want to maximize the value (bottleneck rate).

We start with a root node s and try to greedily grow a tree from s outward. At each step, we simply add the node that can be attached with the highest bottleneck rate to the partial tree we already have. In other wods, we maintain a set S (which is a subset) of V on which a spanning tree has been constructed so far. Initially, $S = s$. In each iteration, we grow S by one node, adding the node v that maximizes the achievable bottleneck rate, and including the edge e = (u,v) that achieves this maximum bottleneck rate in the spanning tree.

We want to show, that in each iteration, it only adds edges belonging to the tree with the best bottleneck rate. In each iteration of the algorithm, there is a set S (subset) of V on which a partial spanning tree has been constructed, and a node v and edge e are added that maximize the bottleneck rate. By definition, e is the edge that has the best maximum bottleneck rate with one end in S and the other end in $V - S$, and so by the Cut property (provided in the lecture notes) it is in every minimum spanning tree.

3 Problem 3

(a) Suppose there are two MSTs, call them T_1 and T_2. Let the edges of T_1 be $(e_{i_1}, e_{i_2}, ..., e_{i_{n-1}})$ and the edges of T_2 be $(e_{k_1}, e_{k_2}, ..., e_{k_{n-1}})$. If the trees are different, then there must be some different edges between the two edge sets. So, let e^* be the smallest cost edge in T1 that is not in T2. Deleting that edge from T_1 would disconnect T_1 into two components. Since T_1 chose that edge, it must be the smallest cost crossing edge for those two components. But by the cut property, that edge must therefore belong to every minimum spanning tree. Thus it must belong to T_2 as well. But this contradicts the definition of e^*. Therefore, if all edge weights are distinct, then the minimum spanning tree is unique.

(b) Multiply all the edge weights by -1. Since both Kruskal's and Prim's algorithms work for positive as well as negative edge weights, we can find the minimum spanning tree of the new graph. This is the same as the maximum spanning tree of the original graph.

(c) For a connected graph, removing the feedback arc set leaves a spanning tree. Hence, to find the minimum feedback arc set, we need to find the maximum spanning tree for every connected component of G. To do that, we can run Kruskal's algorithm, negating the edge weights of the original graph. Once we have the maximum spanning tree T, we can output $E^{'}$ = the set of edges that don't belong to T. Running time here is the same as Kruskal's algorithm.

4 Problem 4

(a) Run the network flow algorithm to compute the max-flow. The critical edge has to be saturated; otherwise a small capacity reduction will not affect the flow. However, saturated edges may be not critical. Assuming e is saturated in the original graph, if there is a circle that passes through the reverse e in the residual graph, we can push a flow through the circle to reduce the flow in e. The flow conservation law will still be satisfied.

The algorithm is: For each saturated edge (u, v) in the original network, run the depth first search to check whether there is a path from u to v in the residual network. If there is a path, (u, v) is not critical, and otherwise it is.

(b) Algorithms:

1. Let E' be the edges $e \in E$ for which $f(e) > 0$, and let $G' = (V, E')$. Find in G' a path P_1 from s to u and a path P_2 from v to t.
2. [Special case: If P_1 and P_2 have some edge e in common, then $P_1 \cup (u, v) \cup P_2$ has a directed cycle containing (u, v). In this case, the flow along this cycle can be reduced by one unit without changing the size of the overall flow. Return the resulting flow.]
3. Reduce flow by one unit along $P_1 \cup (u, v) \cup P_2$.
4. Run Ford-Fulkerson with this starting flow.

Justification and running time:
Say the original flow has size F. Let's ignore the special case (2). After step (3) of the algorithm, we have a legal flow that satisfies the new capacity constraint and has size $F - 1$. Step (4), Ford-Fulkerson, then gives us the optimal flow under the new capacity constraint. However, we know this flow is at most F, and thus Ford-Fulkerson runs for just one iteration. Since each of the steps is linear, the total running time is linear, that is, $O(|V| + |E|)$

5 Problem 5

(a) **Algorithms:**

Construct the flow network from the bipartite graph to solve the maximum matching problem. Initialize the flow with the edges. Now try to augment the flow k times to find M'. If it is not possible to augment the flow by k (or more) then return "no" otherwise return the set of edges constituting the new flow.

Justification: Note that each augment increases the flow by 1 and hence increases the size of the matching by exactly once per iteration. In addition, since the residual edges from y_i to the sink do not exist in the residual flow, therefore for such a vertex to lie on the augmenting path, there has to be an edge from a vertex in X to y_i which means it is still covered. If it does not lie on the augmented path, it stays in the matching as before the augmentation.

Alternative solution:

construct a flow network G' with demands. That is, first add a supersource s^* and directed edges (s^*, x) for all $x \in X$. Direct all edges from X to Y. Every node $y \in Y$ that is covered by M becomes a sink with demand $d(y) = 1$. Connect every other node $v \in Y$ by a directed edge (v, t^*) to a supersink t^* with demand $d(t^*) = k$. Finally, all edges have capacity 1 and s^* has demand $(|M| + k)$.

It is now easy to show that the desired matching M' exists in the bipartite graph G if and only if there is a flow that satisfies the demand in G'.

(b) Consider a very simple graph. The graph has four vertices x_1, x_2, y_1, y_2 and edges $(x_1, y_1), (x_2, y_1)$ and (x_2, y_2).

An example of what is asked in this problem is $M = (x_2, y_1), M' = (x_1, y_1), (x_2, y_2)$.

(c) Try to augment M' until you get M'' in the same way as in (a). If K_1 is not equal to K_2, then it means $K_1 < K_2$ (otherwise M'' is not maximum) and hence you should be able to augment M'. But as in (a), any augmentation also respects the membership of y_i which contradicts the assumption that M' was the maximum matching satisfying (a) and (b). Therefore $K_1 = K_2$.