The Design and Analysis of Algorithms	**Solutions for the Midterm**
CS 681 Fall 2002	

The midterm was graded out of 30 points (15 on each of the two questions). The mean was 20 points. Any grade roughly around the mean (say at least 17 points and up) is reasonable. A grade under 10 is really bad.

(1a) When edge $e = (u, v)$ is deleted from the MST T^*, the tree falls into two connected components. We can get these two components (call them A and B) by doing a search from node u following the tree edges.

The MST in the graph with edge e deleted is the remaining edges of T^* with the minimum cost edge that crosses the cut (A, B). The easiest way to argue that this is true is, by using the cut-property: all edges of T^* are minimum cost edges in a cut, and this remains to be the case after deleting e^*, so they remain in the MST. The new edge added is the minimum cost edge in the cut (A, B).

Algorithm: after determining sets A and B we go through all edges in E to find the cheapest that crosses the cut.

(1b) Solution 1 The idea: sort all edges not in T^* by cost. This takes $O(m \log n)$ time. Now consider the edges in sorted order, and for each edge $e = (u, v)$ determine the unique cycle formed by e and the tree T^*. This cycle consists of edge e and the unique path from u to v in T^*. The vitality of an edge is determined by the first such cycle that the edge shows up on: namely, if e is the first edge in this order whose cycle contains tree edge f, then the edge f has vitality $c_e - c_f$. This is true, as the edges that cross the cut determined by deleting f are exactly the edges e whose cycle passes through f, and we considered the edges in order of their cost.

One difficulty with this idea is, that determining all these cycles makes us trace through the cycles repeatedly, so it seems to take $O(mn)$ time, and not $O(m \log n)$ time). We need to make sure, that after we set the vitality of an edge, we do not have touch it again.

First consider the case when T^* is a path, label the nodes of G so they occur in order along the path T^*. Now assume the first edge is (v_i, v_j), the path from v_i to v_j consists of all nodes v_k for $i \leq k \leq j$. After setting the vitality of all edges along this path, we contract the path. The simplest way to maintain the contracted graph is to use a union-find data structure for the nodes. Each contracted graph is a path, and the unique cycle formed by the path and an outside edge (u, v) can be found by simply following the path from u till we reach v. This clearly takes time proportional to the size of the cycle. So we spend $O(1)$ time on computing the vitality of each edge (after the initial sorting is done).

Now consider the case when T^* is not a path. We can do the same, but there is one additional detail: we need a way to find the cycle formed by an edge (u, v) and the tree, in time proportional to the size of the cycle. To do this we will root the tree in an arbitrary node r, and trace, in parallel, both from u and from v towards the root. The moment these two paths meet, we have the tree-path from u to v.

(1b) Solution 2 Idea: for each cut we want to select the minimum cost edge. This suggest that heaps might be useful. Again start with the case when T^* is a path. We will compute vitalities of the edges in order they appear on the path. Let $v_1, v_2, \ldots, v_n$ denote the nodes as they appear along the path. For the first edge, the vitality is the minimum cost edge leaving v_1. To determine this minimum, we will add all edges leaving v_1 in a heap (Fibonacci heap) H. As we go along the path T^* we want the heap H to always contain all edges crossing the current cut. Assume H is a heap of all edges crossing the cut between $\{v_1, \ldots, v_i\}$ and $\{v_{i+1}, \ldots, v_n\}$. To update the heap for the next edge, we need to delete all edges that connect v_{i+1} to an earlier node (as these edges no longer cross the cut), and add all edges that connect v_{i+1} to a later node. Each edge is added to the heap once, and deleted once. So in total we have $O(m)$ heap operations, for a total of $O(m \log n)$ time.

Now, for a general tree T^* we can do something similar. Root the tree a a node r. We start a heap H_v for each degree 1 node v of the tree, where the heap contains all edges leaving v. The minimum in this heap determines the vitality of the edge from v to its parent u. Then we proceed to u the parent of the node v. If u has other children, then before we can process the edge from u to its parent, we have to be done processing each child of u. Once all children are processed, merge the corresponding heaps, and add all edges leaving u.

One added difficulty in case of a general tree (compared to the case of the path), is that it is a bit harder to determine what edges should be deleted from the current heap. It appears easier to do this in a lazy manner, leave edges in the heap, and only delete them, when the minimum heap element corresponds to an edge that does not cross the current cut.

How do we determine if an edge crosses the cut corresponding to an edge from u to its parent? To be able to determine this fast, we may want to do a DFS on the tree, and label the nodes as we go along. If a node v is the ith in DFS order, and the highest node number assigned before DFS returned from i is node number j, then all nodes numbered between i and j are descendants of i, and these nodes form the v side of the cut formed by deleting the edge between v and its parent. Alternately, a union find data structure can also be used for doing this fast.

Common bad solution. Many people tried to use a divide and conquer type algorithm. For a path, it seems natural to compute the vitality of the middle edge, and try to recurse on each side. There are two problems with this approach. (1) Unfortunately, determining the vitality of the edges on a side, is not a problem that only depends on that side: edges that reach across the middle can also be important. As a result, I see no way to set up the two sides as two disjoint subproblems. (2) In the case of general trees, there may not be a middle edge, for example, when the tree, T^* is a star.

(2a) There are many ways to use the project selection algorithm to solve this problem. Maybe the easiest is to set $p_i = +\infty$ (or something really big). Alternatively, add an edge (s, i) with infinite capacity to make sure that node i remains on the s side of the min cut. Another option is to determine all projects that MUST be undertaken due to taking i (by a search for reachable projects from i in G), delete all these from the problem (as we must do them), and use project selection on what is left.

(b) We know from class that feasible set of projects of maximum profit are in one-to-one correspondence to minimum cuts in the flow graph defined in the problem. So the question is whether there is a minimum cut (A, B) where $i \in A$. Let B be the set of nodes v such that there is a path from v to t in the residual graph. We claim that i is contained in a feasible set of projects of maximum value if and only if $i \notin B$. This can clearly be tested in $O(m)$ time.

To see the "if" direction, note that (A, B) is a minimum cut, if we set A to be the complement of B. This is true as all edges $e = (u, v)$ crossing from A to B must be saturated (else e would be a forward edge in G_f, and hence u would be in B), and all edges $e = (v, u)$ crossing from B to A must have $f(e) = 0$ (else e would give rise to a backwards edge in G_f, and so again u would be in B). This implies that the flow value across this cut is the same as the cut's capacity, and hence this is a min-cut.

To see the "only if" direction, we need to show that if $i \in B$ then no minimum cut can have i on the source side. Let P be the path from i to t in the residual graph, and assume by contraction that i is on the source side of a min-cut. But then some edge of the path P must cross from the source side to the sink side, and this contradicts the fact that the residual graph of a max-flow cannot have edges across a min-cut.

Common bad solution. Many solution involved some greedy style algorithm along the following lines. Let S^* be a set of feasible projects of maximum total profit. If $i \notin S^*$ consider adding i and all projects that are reachable from i in G (and hence are prerequisites for i). We indeed, have to add these projects. However, we may want to add some more, for example, if this set of projects contains a project j with negative profit, and other projects require j as a prerequisite, than they may be added now. In general, project selection appears to need the strength of a minimum cut computation, and greedy type algorithms are not strong enough.

(c) Adding 1 to the profit of project i increases p_i by 1, and hence the capacity of the (s, i) edge by 1. This increase of 1 in a capacity, can cause the max flow value to increase by 1 also. We can find a new max-flow by possibly using one augmenting path to increase the value of the original maximum flow f^*. (Why can the max-flow only increase by 1? One way to see this is via the min-cut, as the max-flow value is the same as the min-cut: adding 1 to the capacity of an edge can only increase the capacity of any cut by at most 1.) One augmentation takes $O(m)$ time.

Once we have a max-flow in the flow graph of the new project selection problem, we need to determine if there is a set of feasible projects of maximum value that contains i. We use the algorithm for part (b) to do this.

Common mistakes. Many students who used a greedy style algorithm for part (b) used the same algorithm here too.

Some flow based algorithms decided the answer based on whether or not f^* was the maximum flow in the new graph. I don't see a way to do this immediately. However, the following is good to notice. The total profit associated with the best set of project increases (with the increased profit of project i) if and only if the maximum flow value does **not** increase. This is, as the flow value in this construction was the *lost profit* not the profit itself. So, if there is **no** augmentation, than the profit increased, and hence i must have been included in the set of selected projects (in fact, in this case i was already in a set of feasible projects of maximum value before adding 1 to its profit). However, it may be possible to include i in a set of feasible projects of maximum value, even if there is a flow augmentation (and hence the total profit achievable does not increase).