CSOR 4231 Sketch of Solutions to Final Exam
December 17, 2003, 9AM

Rules; Answer each question completely and concisely. When you give an algorithm, be sure to give the most efficient one you can, to prove that it is correct, and to analyze its running time. Any NP-completeness proof must be a reduction from one of the problems we studied in class (SAT, 3-SAT, Clique, Independent Set, Vertex Cover, Hamiltonian Path, Set Cover, Subset Sum). When you give an algorithm, be sure to give the most efficient one you can, to prove that it is correct, and to analyze its running time.

The first two problems are loosely related, but do not depend on each other.

Answer 4 of the following 5 questions.

Exam

Problem 1. In a fit of patriotism, the president decides that we need to emphasize the the theme of independence. He therefore signs the *Extremely Independent Graph Act* which decrees that all $|V|$-node graphs that are studied with federal money must have an independent set of size at least $|V|/2$.

You have been hired to enforce this law. Show that enforcement is difficult, i.e. prove that the problem of deciding whether a graph has an independent set of size at least $|V|/2$ is NP-complete.

Solution First we have to show that independent set is in NP. The algorithm to verify that a graph has an independent set of size at least $V/2$ is the same algorithm that verifies that a graph has an independent set, with an additional check that the independent set is actually of size at least $V/2$. Clearly this can be done in polynomial time.

For the reduction, we reduce from independet set with input $(G = (V, E), k)$ to extremely independent set with input $(G' = (V', E'), k')$. The idea of the reduction is to add isolated vertices to G so that if G has an independent set of size at least k, G' has an independent set of size at least $V'/2$. More precisely, if $k \geq V/2$, then just set $G' = G$ and $k' = k$, as we are already asking whether there is a set of size at least $V'/2$. Otherwise, let $x = V - 2k$, let V' be V plus x isolated vertices, let $E' = E$ and let $k' = k + x$. We call the set of isolated vertices I'. Note that $k' = k + x = k + (V - 2k) = V - k$ and $V' = V + x = V + (V - 2k) = 2(V - k)$ and thus $k' = V'/2$. This reduction can clearly be done in polynomial time.

Now we show that G has an independent set of size at least k iff G' has an independent set of size at least k'.

If G has an independet set I of size k and then in G' $I \cup I'$ is clearly an independent set, and it has size $k + x = k'$. If G' has an independet set of size at least k', at most x of the vertices in the indepdent set are from I', and therefore, there are at most $k' - x = k$ vertices from the rest of the graph in the independent set. But then, by the construction, this independent set is clearly an indepdent set in G.

Problem 2. In an effort to save face, and to placate the lumber industry, the president declares that all graphs studied with federal money must be trees (but not necessarily binary trees). Suppose that each node v in a tree has a weight $w(v)$. Give a polynomial time algorithm to find an independent set I that maximizes the sum of the weights of the nodes in the independent set, $\sum_{v \in I} w(v)$.

Prove your algorithm is correct and analyze its running time.

Solution We can find the independent set via dynamic programming. Root the tree arbitrarily. Suppose I knew that a node v was in the optimal solution, then I know that none of its children can be in the optimal solution, and I want to find the optimal solution to the tree rooted at each of its grandchildren. Let $C(v)$ denote the set of children of v and let $CC(v)$ denote the set of grandchildren. If either does not exist, let the set be empty. Define $A(v) = 0$ for the empty set and assume all weighted are non-negative. Thus, the recurrence for a tree rooted at v is

$$A(v) = \max\{w(v) + \sum_{w \in CC(v)} A(w), \sum_{w \in C(v)} A(w)\}.$$

Proof of optimal substructure. Assume that the recurrence was not correct and the optimal solution consisted of non-optimal solution to either a child or a grandchild. Then, since the objective function is a sum, if we replaced the non-optimal solution to a child or grandchild by an optimal one, we would have a bigger total sum, contradicting the optimality of the supposed solution.

One can then give pseudocode. Note that by working up the tree, it is possible to solve this problem in $O(n)$ time. A more brute force solution should yield $O(n^2)$.

Problem 3. You are given a connected graph which has n nodes and m edges. The edges are colored either red or blue. Give the most efficient algorithm you can to find the spanning tree that has the maximum possible number of blue edges.

Solution The simplest solution is to give the blue edges weight 0 and the red edges weight 1 and run a minimum spanning tree algorithm. It is easy to see that the minimum spanning tree will maximize the number of red edges. The running time is $O(E \log^* V)$, since you the sorting at the beginning of Kruskal's algorithm can be done in $O(E)$ time, since the only numbers are 0 and 1.

You can get an $O(E)$ time algorithm by first doing a breadth first search on the blue edges (restarting if you don't reach all vertices). This takes $O(E)$ time. You now consider each connected component of blue vertices to be one new supervertex, and put an edge between two supervertices if there is a red edge between any of the constiuent vertices in the original graph. Now do a breadth first search on this new graph, and output the union of the two trees found. (The details of how to construct the graph for the second phase are omitted, but if done carefully, can be done in $O(E)$ time.

Problem 4. You need to maintain a graph dynamically, that is you want to allow vertices and edges to be inserted and deleted over time. You need to support the following operations:

- InsertVertex(v) - add a vertex v to the graph.
- InsertEdge(u, v, w) - add an edge (u, v) with weight w to the graph.
- DeleteMaxEdge(v) - Let F be the set of edges incident to vertex v. Delete the maximum weight edge in F from the graph.

You can assume that each edge or vertex inserted is unique, that the vertices are labeled with integers between 1 and n, and that a when DeleteMaxEdge(v) is called, v has at least one incident edge. Let n be the total number of operations performed.

a) Explain how to implement this data structure so that each operation has a worst-case cost of $O(\log n)$ time.

b) Explain how to implement this data structure so that each InsertVertex and InsertEdge each has a worst case cost of $O(1)$ time and each operation has an amortized cost of $O(\log n)$.

Solution a) For each vertex v, maintain a heap $H(v)$ of edges keyed by weight. Also, we each edge (u, v) in $H(u)$ maintain a pointer to the other copy of the edge in heap $H(v)$ (and vica versa).

Now we can implement the operations as follows:

- InsertVertex(v) - initialize $H(v)$ to an empty heap.
- InsertEdge(u, v, w) - Insert (u, v) with weight w into heaps $H(u)$ and $H(v)$ and set the pointers between the two copies.
- DeleteMaxEdge(v) - Perform FindMax(v) to get the edge (u, v). Use DeleteMax(v) to remove this edge from the heap $H(v)$. Now take the copy of edge (u, v) in $H(u)$ and perform an IncreaseKey to infinity and then DeleteMax(u) to remove it from $H(u)$.

Each step is a heap operation or an $O(1)$ operation, so they each take $O(\log n)$.

b) The basic idea is to be lazy about inserting. We will maintain, in addition to a heap $H(v)$ a queue $Q(v)$ of edges that need to be inserted into the heap. We will only move edges from Q to H when we need to find the minimum and delete.

Now we can implement the operations as follows:

- InsertVertex(v) - initialize $H(v)$ to an empty heap and $Q(v)$ to an empty queue.
- InsertEdge(u, v, w) - Insert (u, v) with weight w into queues $Q(u)$ and $Q(v)$ and set the pointers between the two copies.
- DeleteMaxEdge(v) - Take all the vertices in $Q(v)$ and insert them into $H(v)$ using standard heap insert. Set $Q(v)$ to be empty. Perform FindMax(v) to get the edge (u, v). Use DeleteMax(v) to remove this edge from the heap $H(v)$. Take all the vertices in $Q(u)$ and insert them into $H(u)$ using standard heap insert. Set $Q(u)$ to be empty. Now take the copy of edge (u, v) in $H(u)$ and perform an IncreaseKey to infinity and then DeleteMax(u) to remove it from $H(u)$.

InsertVertex and InsertEdge are both clealy $O(1)$ worst case time as queue operations are all $O(1)$ time. Now, DeleteMaxEdge may actually have as much as a linear worst case time, but we can get an amortized $O(\log n)$ time by using the bankers method. For each insertedge, place $2 \log n$ dollars on the edge (u, v). Use this credit to pay for the two HeapInserts that will eventually take place on this edge during some call to DeleteMaxEdge. So each DeleteMaxEdge may insert many edges into the two heaps, but you can use the credits from the InsertEdges to pay for it. This leads to an amortized cost of $O(\log n)$. (Note that you can just delete directly from the queue $Q(u)$ also, with a careful implementation.

Problem 5. Consider the following scenario. You are given a number k, n numbers to sort, code for insertion sort and code for heapsort. You must flip k coins, and if $k - 1$ come up heads, you run insertion sort, otherwise you run heapsort. It takes $O(1)$ time to flip a coin.

a) What is the expected running time of this algorithm when $k = 2$.

b) What is the expected running time of this algorithm for arbitrary k and n, in terms of k and n.

c) Is there a value of k so that the expected running time is $O(n \log n)$? If so, please give such a value for k.

Solution a) $.5 * O(n \log n) + .5 * O(n^2) = O(n^2)$

b) $O(\frac{k}{2^k} n^2 + \frac{2^k - k}{2^k} n \log n)$

c) When k is around $\log n$ $k/2^k = \log n/n$, and so both terms are $O(n \log n)$ above.