Solution to Final Exam for CS 25

Prof. Drysdale

December 1, 2003

1. **[15 points]**
   A *mixed* graph is graph in which some of the edges are directed and some of the edges are undirected. If a given mixed graph G has no directed cycle, then it is always possible to orient the remaining undirected edges so that the resulting graph has no directed cycle. Give an efficient algorithm for obtaining such an orientation if one exists. Prove the correctness of your algorithm and analyze its complexity.

   Solution: Create a graph G' consisting of the vertices of G and the directed subset of the edges of G. Topologically sort the vertices of G'. If a cycle is found during the DFS report this and return. Otherwise direct all of the undirected edges so that they are consistent with the direction given by the topological sort. This can be done by running through the vertices in topologically sorted order, directing each undirected edge away from the endpoint found first. Because all of the edges (both directed and originally undirected, but now given a direction) are consistent with the topological sort the new graph is a DAG. Time is $O(V + E)$, the time for topologically sorting the vertices and then running through them (and the edges leaving them) in order.

2. **[25 points]**
   Given n vertices $v_1, \ldots, v_n$, a random graph is formed as follows: for each pair of vertices v_i, v_j with $i \neq j$ we flip a fair coin, and include directed edge (v_i, v_j) in the graph if and only if the coin comes up heads. Two labeled graphs are *identical* if they have the same set of edges. Assume that the graphs are represented as adjacency matrices.

   (a) What is the probability that two random graphs G_1 and G_2 formed on the same n vertices $v_1, \ldots, v_n$ are identical? Justify your answer.

   Solution: There are $n(n-1)$ pairs of vertices in a random graph. The probability that corresponding edges agree in both graphs is the probability that the second graph has the same coin flip as the first, or $\frac{1}{2}$. The edges are chosen independently, so the odds that the graphs are identical is $2^{-n(n-1)}$.

   (b) Given two random graphs G_1 and G_2 formed on the same n vertices $v_1, \ldots, v_n$, we wish to determine if the two graphs are identical. Give an algorithm with the fastest expected running time that you can that solves this problem. What are the expected and worst case running times of your algorithm?

   Solution: Assume that $A1$ and $A2$ are the adjacency matrices for G_1 and G_2 respectively.

```
GraphTest(A1, A2)
for each possible pair (u, v)
    do if A1[u, v] = A2[u, v]
        then return false
return true
```

The algorithm is correct, because when it returns false it has found an edge that is different in the two graphs, and when it returns true all edges agree.

In the worst case all pairs match, and the loop is executed $n(n-1)$ times, for $\Theta(n^2)$ time.

In the expected case we note that each time through the loop the probability that the loop terminates on that step is $\frac{1}{2}$. Thus we have the same problem as determining the average number of flips before a head is found. The expected time is

$$\sum_{j=1}^{n(n-1)} \frac{j}{2^{j-1}} < \sum_{j=1}^{\infty} \frac{j}{2^{j-1}} = 2$$

Thus the expected time for this algorithm is $\Theta(1)$.

(c) Given m random graphs $G_1, \ldots G_m$ formed on the same n vertices $v_1, \ldots, v_n$, we wish to determine all pairs of identical graphs. Give an algorithm with the fastest expected running time that you can that solves this problem. What are the expected and worst case running times of your algorithm? *Hint:* You can do better than trying all pairs.

Solution: We modify Quicksort by replacing Partition so that it partitions graphs based on whether the edge between some pair of vertices (u, v) is present or absent. Each level of recursion choses a different pair to partition on. The base cases of the recursion are either that all $n(n-1)$ pairs have been tested (so all graphs in the partition agree on all edges) or that the size of the partition is 1. When we are done, we report all pairs in every subset with size greater than 1.

The algorithm is correct. Any two graphs with all edges the same will end up in the same partition at the end. Any two that disagree on an edge will be put into separate partitions when the first pair of vertices where they differ is considered as as a partitioning element.

In the worst case all m graphs are identical, and each of the $n(n-1)$ pairs will be tested on all m graphs. Thus the run time is $\Theta(mn^2)$.

For the expected run time, consider the tree of tests, where at each test a partition is (potentially) subdivided. If the tests always split the sets exactly in half, the depth of the tree would be $\lceil \lg m \rceil$. The work on each level would be $\Theta(m)$, so the total work would be $\Theta(m \lg m)$. Note that this is a best case for our algorithm, so will be a lower bound for the work done by our algorithm.

We are not guaranteed even splits, but because the splits are done on edges determined by flips of fair coins they should be close to even. Thus the expected run time of this algorithm should be $\Theta(m \lg m)$. We must prove this.

One way to bound the total work W done is:

$$W \leq \sum_{j=0}^{n(n-1)} mX_j$$

where X_j is an indicator random variable that indicates that some partition on level j contains more than one element. (Note that we call the top level of the tree level 0, so our sum starts at 0. The work done at level $n(n-1)$ is simply the test to see if there are more edges to compare.) This is an overestimate, because once a partition is a singleton it does not appear further down the tree. Therefore many levels will not require m work.

Linearity of expectation gives

$$E[W] \leq \sum_{j=0}^{n(n-1)} mE[X_j].$$

A crude way to bound $E[X_j]$ is to note that if two graphs are in the same partition at level j they must agree in the first j pairs tested, which happens with probability 2^{-j}. If we consider each possible pair, then the probability that two graphs agree on the first j tests so are in the same partition is upper bounded by $\binom{m}{2}2^{-j}$. This is a horrible overestimate, because we are adding probabilities of events that are not mutually exclusive. But it is adequate for our purpose.

We note that when $j = \lceil 2 \lg m \rceil$ that

$$\binom{m}{2}2^{-j} \leq \frac{m(m-1)}{2m^2} < \frac{1}{2}$$

so when $j > \lceil 2 \lg m \rceil$

$$E[X_j] < \binom{m}{2}2^{-j} \leq \frac{m(m-1)}{2m^2}2^{-j+\lceil 2 \lg m \rceil} < 2^{-j-1+\lceil 2 \lg m \rceil}$$

This suggests splitting the sum

$$E[W] \leq \sum_{j=0}^{\lceil 2 \lg m \rceil} mE[X_j] + \sum_{j=\lceil 2 \lg m \rceil + 1}^{n(n-1)} mE[X_j]$$

The first sum has $\lceil 2 \lg m \rceil + 1$ terms, so even if $E[X_j] = 1$ for each j this sum is at most $m\lceil 2 \lg m \rceil + m$. The second sum can be bounded:

$$\sum_{j=\lceil 2 \lg m \rceil + 1}^{n(n-1)} mE[X_j] < \sum_{j=\lceil 2 \lg m \rceil + 1}^{n(n-1)} m2^{-j-1+\lceil 2 \lg m \rceil} < \sum_{k=0}^{\infty} \frac{m}{4}2^{-k} = \frac{m}{2}$$

Together these show that $E[W] < m\lceil 2 \lg m \rceil + \frac{3m}{2} = O(m \lg m)$.

This assumes that the recursion will be stopped because each set will be a singleton, which will not be true if $\lg m$ is larger than $n(n-1)$. In any case we can say that the expected time is $\Theta(\min(m \lg m, mn^2))$.

3. **[20 points]**

An *independent set* of a graph G is a set of vertices, no two of which are adjacent. Finding an independent set is NP-complete, so we are unlikely to be able find independent sets quickly. Therefore we consider an approximation algorithm for a special class of graphs.

(a) Suppose the undirected graph G is regular of degree d. That is, every vertex of G has exactly d vertices adjacent to it. For any fixed d, give an algorithm to find a independent set of size $c|V|$ for as large a value of the constant c as you can. Express c as a function of d.

Solution: A simple greedy strategy will achieve a constant bound. Pick any vertex. Remove it and all of its neighbors from G. Repeat until no vertices are left. The set will be an independent set, because when we choose a vertex none of its neighbors will remain in the graph to be chosen later.

At each step we remove at most $d+1$ vertices, so we can repeat this at least $\frac{|V|}{d+1}$ times. Thus we have an algorithm with $c = \frac{1}{d+1}$. In the case where G consists of k complete graphs on $d+1$ vertices $|V| = k(d+1)$ and the maximum independent set has size k, so the size of the largest independent set is exactly $\frac{|V|}{d+1}$. Therefore the constant cannot be improved, although choosing a vertex of smallest degree at each step would be a good heuristic.

(b) For what ρ is your algorithm a ρ-approximation algorithm? To solve this come up with a good estimate of the maximum-sized independent set possible in a regular graph of degree d. Prove that your estimate is tight.

Solution: A bipartite graph with k vertices in each half and vertex u_i in the left half connected to vertices $v_i, v_{i+1}, \ldots, v_{i+d-1}$ in the right half (where the addition is done mod k) has an independent set of size k, namely the vertices in the left half. So an independent set can have size $|V|/2$.

No larger independent set is possible in a regular graph of degree d. Suppose there is an independent set of size k in G. Then there are exactly kd edges coming out of the independent set. Because every vertex has degree d, at most d of these edges can end at a single vertex not in the independent set. Thus there must be at least $kd/d = k$ vertices not in the independent set. This shows that at most half the vertices can be in an independent set.

Given this, we can argue that $\rho =$(size of optimal independent set)/(size of set we find) is at most

$$\frac{\frac{|V|}{2}}{\frac{|V|}{d+1}} = (d+1)/2$$

.

4. **[20 points]**
You are given a weighted directed graph $G = (V, E)$, with a source s and edge weights $w(u, v)$ for each edge $(u, v) \in E$. Furthermore, you know that the *only* edges with negative weights leave the source vertex s. Give the most efficient algorithm that you can to compute the single-source shortest paths from s to all other vertices (or report that G has a negative cycle, if one exists). Why does your algorithm work, and what is its run time?

Solution: The reason that Dijstra's algorithm cannot handle negative edge weights is that they can violate the loop invariant given on p. 597 of CLRS: At the start of each iteration of the while loop of lines 4-8, $d[v] = \delta(s, v)$ for each vertex $v \in V$. There are exactly two places where the non-negativity of edge weights is used. The first is on p. 597, where the proof notes that s is the first vertex added to S, and when that happens $d[s] = \delta(s, s) = 0$. If there is a negative cycle involving s the claim that $d[s] = \delta(s, s) = 0$ would be false.

The second place where non-negativity of edge weights is needed is in the second paragraph of the proof on p. 598. It is used to prove that if y occurs before u on a shortest path from s to u, then $\delta(s, y) \leq \delta(s, u)$.

However, all that is really needed to prove the claim is that there are no negative weight edges on the shortest path from s to u in the section between y and u. But y cannot be s (because $s \in S$ and $y \in V - S$) and the only negative weight edges in G leave s. Therefore unless s appears between y and u on the shortest path from s to u there will be no negative weight edges between y and u and the claim that $\delta(s, y) \leq \delta(s, u)$ still holds. There is always a simple shortest path from s to u (so s will not appear between y and u) unless there is a negative weight cycle in G.

So Dijstra's algorithm will work correctly in the presence of negative edges leaving s as long as there is no negative weight cycle. Any negative weight cycle must include s, because all cycles not including s have no negative weight edges. In fact, since all paths start at s a negative weight cycle will result in $d[s]$ being set to a negative value during the call to Relax when u is s's predecessor on a negative cycle. So we need to modify Dijkstra's algorithm by adding a test at the end to make sure that $d[s]$ is not negative. If it is, we report a negative cycle. If not, we have a valid shortest path graph.

The run time of Dijkstra is $O(E + V \log V)$ using Fibonacci heaps.

5. **[15 points]**
A graph G has edges which are colored either red or blue. Give the fastest algorithm that you can to compute a spanning tree with as few blue edges as possible. What is the run time of your algorithm?

   Solution: Assign the red edges weight 1 and the blue edges weight 2 and run an algorithm to compute an MST. The MST will be a spanning tree, and minimizing the total weight is equivalent to minimizing the heavier blue edges.

   This leaves the question of what algorithm to use. Kruskal's algorithm can sort the edges in $O(E)$ time using counting sort, but the disjoint set union algorithm is still needed, so its run time is $O((V + E)\alpha(V))$. Prim's algorithm can use a priority queue consisting of an array. We saw in Exercise 23.2-5 that when weights run from 1 to W that Prim's algorithm can be made to run in time $O(E + WV)$. Because $W = 2$ this is $O(E + V)$, which is as fast as we could hope to get the algorithm to run.

6. **[20 points]**
You are given a directed network $G = (V, E)$ in which each edge (u, v) has a positive integral capacity $c(u, v)$. G has a source s and a sink t. An edge is called *sensitive* if it crosses *some* minimum cut (S, T) of the network, going from S to T.

   (a) Show that there is a graph with an exponential number of different minimum cuts.

   Solution: Consider the graph with vertices s, t, and $u_1, \ldots, u_n$. Connect s to each of the u_i by an edge of capacity 1 and connect each u_i to t by an edge of capacity 1. Then the maximum flow is n. Let $U = \{u_1, \ldots, u_n\}$ and $W \subseteq U$. Then $S = \{s\} \cup W$ and $T = \{t\} \cup U - W$ is a minimum cut. There are 2^n subsets of U, so 2^n minimum cuts.

(b) Give as efficient a polynomial time algorithm as you can to find all sensitive edges. Prove its correctness and analyze its runtime.

Solution: A sensitive edge crosses a minimum cut. Reducing its capacity will therefore reduce the total flow in the network. Conversely, if reducing an edge's capacity does not reduce the total flow in the network then the edge cannot cross any minimum cut and is therefore not sensitive. This gives a test for finding sensitive edges.

Use the fastest available network flow algorithm to compute a maximum flow. For each saturated edge, reduce its capacity by one in the original graph and determine if that reduces the flow. The solution to Problem 26-4b shows how to do this in $O(E)$ time. If the flow reduces, then the edge is sensitive and is added to the list of sensitive edges. If reducing the capacity does not reduce the flow then the edge is not sensitive.

Thus we can solve the problem in time $O(E^2)$ plus the time to solve the network flow problem. The chapter notes on p. 689 give choices for solving network flow.

7. **[25 points]**
You are to design a data structure that supports the following operations:

CreateSet$(x_1, x_2, \ldots, x_k)$ - Creates a k-element set containing $x_1, x_2, \ldots, x_k$. It returns a reference to the set.

DeleteSet(r) - Removes the set that r refers to.

Union(r_1, r_2) - Forms the union of the two sets referred to by r_1 and r_2 and returns a reference to the merged set.

FindLargest() - Returns a reference to one of the sets that has the largest number of elements. Thus if the sets are $\{a, b\}$, $\{c, d, e\}$, and $\{x, y, z\}$ it should return a reference to either $\{c, d, e\}$ or $\{x, y, z\}$.

The amortized time for CreateSet should be $O(k)$ and the amortized time for all other operations should be $O(1)$.

Your structure should include an array of references, one for each possible set size, and you should bucket the sets by size.

(a) Assume that no set will ever contain more than M items. Describe your data structure in detail and explain how to perform each operation. (Note that you do not have time to initialize the array of M elements. You can allocate it in constant time, but cannot initalize it.)

Solution: Each set will be represented by a circular doubly linked list of elements. The list will also have a field containing its size (number of elements). The data structure will implement a priority queue using an array Q with indices 1 to M, as recommended above. Item $Q[i]$ will eventually hold the list head for a doubly linked list of sets of size i. The variable *biggest* will hold the largest i for which $Q[i]$ is a non-empty list, and is initially 0. We also have a variable *maxAssigned*, which keeps track of the maximum position in array Q which has been initialized. At the start *maxAssigned* is set to 0.

```
CreateSet(x1, x2, ..., xk)
    r = new circular linked list containing items x1, ..., xk.
    size[r] = k
    while  maxAssigned <  k
       do maxAssigned = maxAssigned +1
          Q[maxAssigned] = new empty list
    insert r into the list Q[k].
    if k > biggest
        then biggest = k
    return r

DeleteSet(r)
    remove r from the list in Q[size[r]]
    while biggest > 0 and Q[biggest] is empty
        do biggest = biggest - 1

Union(r1, r2)
    remove r1 from the list in Q[size[r1]]
    remove r2 from the list in Q[size[r2]]
    r = new circular linked list formed by joining the elements in r1 and r2
    size[r] = size[r1] + size[r2]
    while maxAssigned < size[r]
        do maxAssigned = maxAssigned +1
           Q[maxAssigned] = new empty list
    insert r into the list at Q[r[size]]
    if r[size] > biggest
        then biggest = r[size]
    return r

FindLargest()
    if biggest = 0
        then throw "no sets" error
    else return first item in Q[biggest]
```

This works because *biggest* always keeps track of the largest i such that $Q[i]$ is not empty, which contains the list of largest sets. Thus FindLargest can return a largest set immediately, or throw an error if all the lists are empty. The obvious places where biggest has to be updated are in CreateSet, when a k-set is created with $k > biggest$, and in Union, when the union is bigger than the former *biggest*. But it also has to be updated in DeleteSet, because if the last item in $Q[biggest]$ is deleted then we have to find the next largest non-empty $Q[j]$ and set $biggest = j$. This is done in the while loop. (Note that if $Q[biggest]$ does not become empty then *biggest* does not change.)

The other tricky item is keeping track of which positions in Q have been initialized, and initializing new positions when necessary. Originally none of Q is initialized. Whenever a

new, larger set is created it is necessary to initialize all positions up to that new size when necessary. The loops that do this appear in CreateSet and Union, and they also update the value of *maxAssigned*.

(b) Justify that the amortized running time of each operation is as stated above. (Don't worry about time taken to free up memory.)

Solution: First we give the actual run times for each operation, and then amortize.

CreateSet($x_1, x_2, \ldots, x_k$) - $\Theta(k)$ time to create a circular linked list of size k and $O(k)$ to initialize any uninitialized positions (at most k of them). Inserting into list $Q[k]$ and updating *biggest* are $O(1)$ time.

DeleteSet(r) - $O(1)$ to do the actual removal, because we have a reference to the set, and the sets are in a doubly linked list. Updating *biggest* could cost time $size[r]$.

Union(r_1, r_2) - Removals are $O(1)$ as noted for DeleteSet. Joining two circular linked lists into one bigger one, computing $size[r]$, inserting r into $Q[size[r]]$, and updating *biggest* are $O(1)$. Updating *maxAssigned* and initializing new lists can take time $\min(size[r1], size[r2])$. (Since both sets existed, everything up to $\max(size[r1], size[r2])$ was already initialized.

FindLargest() - $O(1)$

Note that all operations except creating the circular linked list to represent the set in CreateSet, initializing more positions in Q (which updates *maxAssigned*) in CreateSet and Union, and updating *biggest* in DeleteSet have $O(1)$ actual cost. Creating the circular linked list in CreateSet is no problem, because CreateSet can have $\Theta(k)$ amortized time, which is the time needed. We need to amortize the cost of initializing Q and of updating *biggest* in DeleteSet.

I will use the accounting method to amortize the costs. CreateSet is charged $3k$. We use k of the tokens to pay for creating the circular linked list that represents the set. Two tokens are put on each list element. One is labeled "array initialization" and the other is labeled "updating *biggest*". Note that CreateSet is $\Theta(k)$, as required.

Whenever we need to initialize positions in Q it is because we have a set r with $n = size[r]$ elements to insert in $Q[n]$. Each of these n elements originally had a token placed on it labeled "array initialization". I claim that the number of "array initialization" tokens currently on the elements of set r are sufficient to carry out the cost of the initialization loop. If an element currently does not have a token for this purpose it can only be because it was spent earlier. Therefore if we spend all the "array initialization" tokens currently on elements in set r we will have spent at least n tokens on array initialization, either now or earlier. No position in Q needs to be initialized more than once, so there are enough "array initialization" tokens on elements in set r to pay for the necessary array initialization.

When we call DeleteSet(r) we may have to spend $size[r]$ time updating *biggest*. However, none of the elements in set r have been deleted before, so there are $size[r]$ "updating biggest" tokens on the $size[r]$ elements in r. Use these to pay for updating *biggest*.

(c) If you had no upper limit M how could you change things so that you could still achieve the same amortized running times? Outline the difficulty and how you would get around it.

Solution: The problem is that when a set gets larger than M we need to allocate a larger array Q for the priority queue and reset M to the larger size. If we increased the size by a

constant amount each time a bigger set was created by a Union it could be too expensive. The solution is to double the size when a larger set is created via a Union and to more than double the size when a larger set is created by a CreateSet.

If a CreateSet is to create a set of size $k > M$, first create a new array of size $2k$ and copy the references to the linked lists in items $Q[1]$ to $Q[maxAssigned]$ to the corresponding locations in the new array. Then set M to $2k$ and Q to the new array. Proceed with the code for CreateSet as given above.

When $size[r1] + size[r2] > M$ in Union, first create a new array of size $2M$ and copy over the references to the linked lists in items $Q[1]$ to $Q[maxAssigned]]$ to the corresponding locations in the new array. Then set M to $2M$ and Q to the new array. Proceed with the code for Union as given above. Note that $size[r1] + size[r2] \leq 2M$, so there will be a place for the new set in the new array.

To pay for this recopying we charge CreateSet $5k$ instead of $3k$. Two tokens labeled "array doubling" are placed on each token. Because allocating the new array, resetting M, and resetting Q take $O(1)$ time total all that remains to be paid for using these tokens is copying entries in the array.

When CreateSet creates a new array it is because the number of new items created is greater than the current M. Therefore there are at least twice as many tokens on the elements of the new set marked "array doubling" as are needed to copy all of the assigned elements. Take enough tokens to pay for the copying.

When Union creates a new set r with $size[r] > M$, there must be at least $M + 1$ elements in set r. The maximum number of copyings would be if the size of the array doubled each time, which would be $M/2 + M/4 + M/8 + \ldots < M$ total copyings. That would have cost M "array doubling" tokens. But there were at least $2(M + 1)$ "array doubling" tokens assigned to the $M + 1$ elements in set r. Therefore there must be at least $2(M + 1) - M = M + 2$ "array doubling" tokens on the elements in set r, so we can take enough to pay for this copying.