Problem Set 3A Solutions

(20) Problem 14 We claimed that the stable marriage algorithm could be implemented to run in $O(n^2)$ time. Describe data structures which will allow this run time. In particular you should describe how to store the prefence lists, the current marriage, and the men who are unpaired.

Justify that each proposal now takes O(1) time.

solution We can store the unpaired men in a queue or a stack. Then, checking to see if there are any unpaired men and getting the next unpaired man can be done in constant time. We can store the current spouse of a women in a length n array (assuming we have numbered the women and men 1 to n. Clearly, setting and retreiving this can be done in constant time.

This leaves the problem of the preference lists. For the men, it is easy enough to store the list in an array in the "normal" order, keeping around the last index looked at. For the women, however, storing the lists this way presents a problem. Whenever we look up the ranking of the current proposing groom, we must take $\Theta(n)$ to do a linear search to find his position. To overcome this, we can store the preference list "inside-out." Instead of listing the grooms in rank order, we can list the ranks in groom order. Thus, when we want to look up a prospective bride's rank of groom x, we can merely look in cell x of her preference table.

To be thorough, we need to show that we can make this conversion in no more than $\Theta(n^2)$ time, the overall running time of the algorithm. This is easy. We have n brides, each with a size n array to convert. Each step of this conversion takes constant time (reading groom x at position i in the old array, causes us to set postion x to i in the new array). Thus the pre-processing takes $\Theta(n^2)$.

Given that each part of the proposal takes constant time, we claim that the proposal takes constant time overall.

(25) Problem 15 Example showing least duration doesn't work: (1,6) (5,8) (7,12) Here we would pick (5,8) but (1,6) (7,12) is better

---

Example showing fewest overlapping... (1,5) (2,9) (3,10) (4,11) (6,13) (12,15) (14,22) (16,24) (17,25) (18,26) (23,27)

First four overlap, second four overlap, last four overlap, next to last four overlap, but so all except (12,15) overlap at least 3 others, and (12,15) only overlaps 2. So, we would pick (12,15) then (1,5) then (23,27)

but (1,5) (6,13) (14,22) (23,26) is better

Example showing earliest start time doesn't work (1,15) (2,10) (11,15) Here we would pick (1,15) but (2,10) (11,15) is better

(25) Problem 16 Suppose that we implement the union-find data structures using trees as described in class (and in 21.3) and using *union-by-rank* where when we union two trees, we have the one of smaller height point to the one of larger height (as in the Link operation on p. 508).

Prove that any tree with height h has at least 2^h nodes (so the maxinum height is at most log n). Note that a tree with one node has height zero.

solution This is easily proven by induction. Note that for values of h=0 and 1, the trees have at least 2^0=1 and 2^1=2 nodes respectively. Let us assume that this is true for some height k, ie, a tree of height k has at least 2^k nodes. A tree of height k+1 is created when a tree of height k is made to point to another tree of height k. For this new tree of height k+1, the minimum number of nodes is the sum of the minimum number of nodes in each of the trees of height k, which is $2^k + 2^k = 2^{k+1}$. Therefore, if the formula is true for h=k, then it is also true for h=k+1. Since it is true for h=0 and 1, it is also true for all values of h.

(25) Problem 17. After finding the strongly connected components of a directed graph G we can then form the *condensed graph* G_c which has a vertex for each strongly strongly connected component (SCC) in G. For a vertex x in G and U in G_c we say that x *is in* U. For two vertices U and V in G_c, there is an edge (U, W) if and only if there is a vertex x in U and a vertex y in W and an edge (x, y) in G.

(a) Prove that G_c is always acyclic.

solution Try a proof by contradiction: Suppose (for contradiction) that G_c contains a cycle. Let $A_1, A_2 \ldots A_k$ be the be vertices in the cycle in G_c, and thus strongly connected components in G. Thus there are arcs $(A_i, A_{i+1}),\ i = 1, 2, \ldots k-1$ and (A_k, A_1) in G_c. By the definition of G_c there are vertices a_i and b_i in $A_i, i = 1, 2, \ldots, k$ such that (a_i, b_{i+1}) and (a_k, b_1) are arcs in G. We know that there are paths between any two vertices within a SCC, so there is a path from b_i to a_i within each SCC A_i on the cycle. Thus there must also be a cycle in $G = a_1 \to b_2* \to a_2 \to b_3 \ldots b_k* \to a_k \to b_1* \to a_1$, corresponding to the one in G_c, Following this cycle, we can find a path between any two vertices in A_i and A_j putting them in the same SCC of G, contradicting the fact that A_i and A_j are different vertices in G_c.

(b) Suppose that we have formed G_c. Describe how we can use the SCC's of G and G_c to answer questions of the form: is there a path from x to y for two vertices x and y in G.

solution There is a path from x to y in G if and only if there is a path in G_c from the vertex (SCC) containing x to the vertex (SCC) containing y. This can be determined by a DFS or BFS in G_c from each vertex in $O(n_c(m_c + n_c)$ time, or we can precompute the transitive closure of G_c, look up the SCC's for x and y (using a table if fixed, or a FIND if not) and then look up the entry.

(c) Suppose we add a new edge (x, y) to G. Describe how to update G_c quickly (you may assume you have stored things in a form to help support this operation). Give the running time of your solution in terms of the number of vertices and edges in G and G_c (let m_c, n_c be the number of edges , vertices in G_c).

solution Let X and Y be the vertices in G_c containing x and y respectively. We need to either add the edge (X, Y), if it is not already an edge, or combine the two vertices (and possibly others) into a single vertex if a path from Y to X already exists in G_c.

If (X, Y) is a new edge in G_c, we need to check for the possiblilty that we have added a cycle to (and thus changed the vertex set of) G_c

Finding the vertices (SCC's) of G_c containing x and y can be done in worst case $O(logn)$ using union find (union by rank) on the SCC's of G. Checking for (X, Y) in G_c is $O(m_c)$.

To find the vertices to merge into a single new vertex, just run the SCC algorithm on the new version of G_c. The effect will be to collapse some of the vertices of G_c into a new SCC and the rest of the vertices will remain untouched. This takes a total of $O(n_c + m_c)$ time (assuming G_c is stored in adjacency list form). The only thing to update with respect to G besides the original edge (x, y) is the sets in the union find structure for the SCC's. Unioning the sets corresponding to the vertices in the new SCC is $O(n_c)$ extra time since we already know which sets to union and need only set at most n_c pointers.

The total running time to update G_c when an edge is added to G is worst case $O(logn + n_c + m_c)$.

(25) Problem 18. For video conferencing you sometimes need to find a route in a computer network which has enough capacity for the connection. Suppose you have a network with n nodes and $b_{i,j}$ is the *bandwidth* of the link connecting nodes i and j (assume bandwidth is symmetric, so $b_{i,j} = b_{j,i}$). The *bandwidth* of a connection depends on the smallest link in the route. Thus if you find a route, say $i - j - k$ connecting nodes i and j, the *bandwidth* of this route is the minimum of $b_{i,j}$ and $b_{k,j}$.

a) Suppose you are given two nodes i and j and a needed bandwidth B, how would you efficiently find a route from i to j which has bandwidth $\geq B$?

solution Note that there are several possible solutions to these problems. In particular, parts b,c can also be done using binary search and BFS.

For this problem we are merely asked to find a route, not an optimal one. In this setting, running a BFS or a DFS from node i until we encounter node j will work as long as we handle the restriction

on edge weights. We can make sure that no edges of weight less than B are used if we first modify the graph by eliminating any edges that have too small a weight (we can do this by just having our DFS/BFS ignore all such edges).

b) Now suppose you want a route of maximum bandwidth from i to j. How would you efficiently find such a route?

solution Now we need to find an optimal path. For this, it is easiest to modify an existing optimal path algorithm. We choose Dijkstra's. In the old version we calculated distances as sums of path weights and made a change to a node's optimal distance when a lower one was found. Given this setting, we should now calculate distances as the minimum edge weight on a path and update a node's optimal distance when we find a higher one. To make these changes, we need only alter the Relax method from page 520, as follows.

```
RELAX
1 if d[v] < min{d[u],w(u,v)}
2.   then d[v] <- min{d[u],w(u,v)}
3.     pi[v] <- u
```

Further, because we are now looking for a maximum, we should now initialize all our vertex distances to negative infinity (save the start vertex) and organize our priority queue to give us the outstanding vertex with the maximum distance. As in part (a), we can terminate the algorithm early. For this algorithm, we can do this as soon as node j is given a permanent distance.

We should note that other solutions are possible. For example, if we get a Maximum Spanning Tree, we can use the links in the tree to form a path in between any two nodes. Note that there will never be an alternate path because there are no cycles. There is also a method that is much like binary search on graphs. This method is not as easy to describe as the given solution, but it can run in time linear to the number of edges given a precise implementation.

c) Now suppose you want to find routes from a node i to all other vertices in the network which have bandwidth $\geq B$. Describe how to find such routes quickly.

solution This is very much like part (a) except we do not have a particular vertex in mind. We approach it the same, except we now let the BFS or DFS run its course. At the end we will have a tree that connects node i to all other possible destinations. The links of the tree represent possible paths to any node in the tree.