

# COT 5405 - Summer 2008

## Homework 3(solution)

July 6, 2008

### Grading Policy:

- Please contact TA Shahed Nejhum by email or in his office hours for any grading issues.
- Maximum score is 100 points.
- Each completed question worth 10 points.
- Partial credit is given if you dont answer a question completely.
- Problem 3 is graded for 50 points.
- Please notify the TA if you find anything wrong in this solution.

### Problem 1 (Problem 15-4 in page 367)

The problem can be transformed into a tree coloring problem where we consider the supervisor tree and color each node red if the employee is attending and white otherwise. The parent of a red node must be white. We wish to color the tree so that the sum of the conviviality of the nodes is maximised. Let  $v = T(x, c)$  be the conviviality of the tree rooted at the node  $x$  that is colored with color  $c$ . We can construct the following recursion:

If  $x$  is a leaf with convivialty  $v$  and color  $c$  then:

$$T(x, c) = \begin{cases} v & \text{if } x = RED \\ 0 & \text{if } x = WHITE \end{cases}$$

If  $x$  is a leaf with convivialty  $v$  and color  $c$  then:

$$T(x, c) = \begin{cases} v + \sum_i T(x.child_i, WHITE) & \text{if } x = RED \\ \sum_i \max(T(x.child_i, RED), T(x.child_i, WHITE)) & \text{if } x = WHITE \end{cases}$$

The maximal conviviality  $v_{max}$  is then given by  $v_{max} = \max(T(root, WHITE), T(root, RED))$ .

### Problem 2 (Problem 17-2 in page 462)

Basically, you need to understand that you take the binary representation of  $(n+1)$ , which obviously takes  $\lceil \log(n+1) \rceil$  bits, and based on whether  $k - th$  bit is one or zero, you do have (or do not have) a subarray of size  $2^k$ . For example, if  $n = 6$  (binary representation of 6 is 110), then there would be two arrays: size 4, and size 2. Array of size 1 is absent since its bit is zero.

And pay close attention to this: each array is sorted individually, but no other relationship between any two arrays exist.

## Part a: Performing SEARCH

For searching, potentially you need to search all the arrays. Since binary-searching of a  $s$ -sized array takes  $O(\lg s)$  time, in the worst case where you need to search all the arrays would take:

$$\begin{aligned} & O(\lg 2^{(k-1)}) + O(\lg 2^{(k-2)}) + \dots + O(\lg 2^1) + O(\lg 2^0) \\ &= (k-1) + (k-2) + \dots + 2 + 1 \\ &= O(k^2) \\ &= O(\lg n)^2 \end{aligned}$$

## Part b: Performing INSERT (Worst-case and Amortized)

Remember that ALL the subarrays (of sizes in powers of 2) must be FULL. Therefore, take some practical examples to identify the trend:

- when there is 0 element (0), to add one you will have to just create a single 1-array and then place the integer in it. Cost = 1.
- when there is 1 element (1), to add one you will have to change the 1-array into a 2-array and then sort it. Cost = 2.
- when there are 2 elements (2), to add one you will have to just create a single 1-array and then place the integer in it. Cost = 1.
- when there are 3 elements (2 + 1), to add one you will have to merge them to create a single 4-array and then sort it. Cost = 4.
- when there are 4 elements (4), to add one you will have to just create a single 1-array and then place the integer in it. Cost = 1.
- when there are 5 elements (4 + 1), to add one you will have to change the 1-array into a 2-array and then sort it. Cost = 2.
- when there are 6 elements (4 + 2), to add one you will have to just create a single 1-array and then place the integer in it. Cost = 1.
- when there are 7 elements (4 + 2 + 1), to add one you will have to merge all of them into a single 8-array and then sort it. Cost = 8.
- when there are 8 elements (8), to add one you just create a new 1-array and place the new integer in it. Cost = 1.
- when there is 9 elements (8 + 1), to add one you will have to change the 1-array into a 2-array and then sort it. Cost = 2.
- when there are 10 elements (8 + 2), to add one you will have to just create a single 1-array and then place the integer in it. Cost = 1.
- when there are 11 elements (8 + 2 + 1), to add one you will have to merge the 1-and 2-array to create a single 4-array and then sort it. Cost = 4.

To understand how we can sort  $n$  elements in  $O(n)$  worst-case time, follow the logic. You can start doing pairwise sorting of already sorted lists of equal sizes. For example, when you have 7 elements (1-array + 2-array + 4-array) and want to add one more, you can first just sort the new element with 1-array (giving rise to a new 2-array). Now, use Merge routine to merge this new 2-array to the existing 2-array to a get a new 4-array. Finally, use Merge routine to merge this new 4-array to the existing 4-array to a get a new 8-array (with no other array remaining). This way, the worst cost is  $2 + 4 + 8 + \dots + n = 2n = O(n)$ .

So how do you calculate the amortize cost? Well, about every alternate insertion costs 1, every 1/4-th insertion costs 2, every 1/8-th insertion costs 4 and so on. So, the total cost is  $O(\sum_{i=0}^{\lg n} \frac{n}{2^i} * 2^i) = O(n \lg n)$ .

Thus, the amortized cost of a single insertion is  $O(n \lg n)/n = O(\lg n)$ .

## Problem 3

### Part a: Exercise 22.1-6

An universal sink will have two properties:

1. All other vertices will have an edge into it.
2. It will not have any outgoing edge, including a self-loop.

However, doing the check above takes  $O(V)$  time for any vertex alone, so the question of running the above check for every vertex is ruled out. Thus, you must devise an algorithm where you keep on excluding vertices from your consideration and finally you perform the check for only one vertex. Here is a sketch below.

Start traversing the individual row for a vertex  $u$  in the adjacency matrix from the cell  $(u, u)$  (you can start from  $u = 1$ ). You keep on checking if  $(u, v)$  is 1 or 0.

1. If edge  $(u, v)$  exists,  $u$  can be excluded, so we resume scanning from  $(u + 1, u + 1)$ . Observe that there is no point scanning the elements in the  $(u + 1) - th$  row to the left of  $(u + 1, u + 1)$ , as those vertices (less than  $u + 1$ ) have already been excluded.
2. If edge  $(u, v)$  does not exist,  $v$  can be excluded, so we keep on searching in the same row (basically, move on to the  $(u, v + 1)$  cell).

This way, for any vertex  $w$ , if you reach the right-hand end of the row, that means all the vertices before  $w$  have already been excluded and  $w$  did not have any outgoing edges to any vertex greater than  $w$ . So for that special vertex  $w$ , perform the check described above in  $O(V)$  time. If it succeeds, there is an universal sink; otherwise there is none.

Why is it not possible that we might have a sink for a vertex bigger than  $w$ ? After all, we did not even examine the rows from  $(w + 1)$  onwards. The reason there cannot be any such rows is because if there were to be any such sink  $s$ , there must have been an edge  $(w, s)$ , which contradicts our assumption that all the cells  $(w, k)$  where  $k > w$  are 0.

### Part b: Exercise 22.2-7 in page 539

Run BFS on any node  $s$  in the graph, remembering the node  $u$  discovered last. Run BFS from  $u$  remembering the node  $v$  discovered last.  $d(u, v)$  is the diameter of the tree.

Correctness: Let  $a$  and  $b$  be any two nodes such that  $d(a, b)$  is the diameter of the tree. There is a unique path from  $a$  to  $b$ . Let  $t$  be the first node on that path discovered by BFS.

If the paths  $p_1$  from  $s$  to  $u$  and  $p_2$  from  $a$  to  $b$  do not share edges, then the path from  $t$  to  $u$  includes  $s$ .

$$\begin{aligned}d(t, u) &\geq d(s, u) \\d(t, u) &\geq d(s, a) \\d(t, u) &\geq d(t, a) \\d(b, u) &\geq d(b, a)\end{aligned}$$

Since  $d(a, b) \geq d(u, b)$ , we conclude  $d(a, b) = d(u, b)$ .

If the paths  $p_1$  and  $p_2$  share edges, then  $t$  is on  $p_1$ . Since  $u$  was the last node found by BFS,  $d(t, u) \geq d(t, a)$ . Since  $p_2$  is the longest path  $d(t, a) \geq d(t, u)$ . Thus  $d(t, a) = d(t, u)$  and  $d(u, b) = d(a, b)$ . Now,  $d(a, b) \geq d(u, v)$ , and  $d(u, v) \geq d(u, b) = d(a, b)$ , so all three are equal. Thus  $d(u, v)$  is the diameter of the tree.

## Problem 4

### Part a:

We shall prove that an MST of a graph is also a bottleneck spanning tree (BNST). Suppose not; i.e., suppose  $T$  is an MST of a graph  $G$  and that there is a BNST  $T'$  such that -

$$\max\{wt(e) : e \in T\} = \alpha > \beta = \max\{wt(e') : e' \in T'\}.$$

Let  $e_m$  be an edge in  $T$  of weight  $\alpha$ . The above condition implies that  $e_m \notin T'$  and that  $e_m$  is the heaviest edge in  $cyc_{T'}(e_m)$ , because all other edges in that cycle are of weight  $\beta$  or less. But  $e_m$  cannot belong to any MST, a contradiction.

### Part b:

Given a graph  $G$  and a value  $\beta$ , we wish to determine, in linear time, whether  $G$  has a BNST with maximum edge weight  $\leq \beta$ . This is almost trivial: just delete all edges in  $G$  with weight  $> \beta$  to obtain a subgraph  $G'$ . Then  $G$  has a BNST of the required type iff  $G'$  is connected. To check this connectedness condition, use any linear time search algorithm, such as DFS, on  $G'$ .

### Part c:

We wish to find a BNST of a given graph connected graph  $G$  in linear time. The key step is to call the algorithm from part (b) passing it the median edge weight,  $w_m$ , of  $G$ . Recall that median finding can be done in linear time, so this does not require sorting the edges of  $G$  by weight.

Let  $E_>$  denote the set of edges with weight  $> w_m$ . If  $G$  happens to have a BNST of weight  $\leq w_m$ , we can safely delete all edges in  $E_>$ ; note that this gets rid of about half the edges of  $G$ . On the other hand, if  $G$  does not have such a BNST, then we can use DFS to find a spanning forest of  $G - E_>$  and, in linear time, contract all such components to single vertices. Note that this also gets rid of about half the edges of  $G$ . Therefore, in either case, we may safely recurse on the remaining graph, for a total running time of something like

$$O(m + m/2 + m/4 + \dots) = O(m).$$

We keep track of all spanning forest edges we find during this algorithm and return the union of these edges as our BNST.