

COT 5405 Homework 4 Solutions

March 25, 2009

1 Grading Policy

- Maximum Score is 100.
- Each completed question is worth 10 points.
- Question 5 is graded for additional 50 points. Understanding the key structure for bitonic sequences 10 points, algorithm analysis 15 points and algorithm itself is 25 points. Note that, if you did not explained to me why the bitonic sequence has at most two changes, which is due to circular shift, you lost 5 points. This is because that will show you understood the structure or not. I counted understanding the bitonic structure as a part of proof of correctness to your solution, but you also required to use the uniqueness of the edge weights and path relaxation lemma to make sure algorithm gives the correct answer which is also 5 points out of the algorithm's 25 points share. By the way, apart from this question if you did not explain implementation in Problem 23-4, I cut 1 point for each one you did not explained.
- For exceeding the page requirement which is one sheet per question is penalized for 10 points.
- Please note that, no late submissions are accepted.
- If you have a similar answer to an answer from somewhere else or someone and if you did not give any citations/references/people who discussed related to your answer, the decision will be made soon, since I have already warned you before by saying that "This penalty will be increased for the upcoming homeworks, in case it happens again!". Note that, you can discuss your answers or explore resources, but you need to give references to your sources. **THIS DOES NOT MEAN YOU ARE ALLOWED TO COPY SOME SOLUTIONS WORD BY WORD AS LONG AS YOU REFERENCED! YOU NEED TO UNDERSTAND AND USE YOUR OWN WORDS!!!** Remember, homework description says "**Feel free to consult your textbooks, journal and conference papers and also discuss the problems with each other, but *write the solutions***

yourself and cite all your sources". You are responsible to write your own solutions.

- Please read the regrading policy from the course syllabus. Contact TA Hale Erten regarding issues about grading (via email or during her office hours).
- Please let us know, if you find any mistakes in this solution.

2 Problem 17-2

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.

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

$$O(\log 2^{k-1}) + O(\log 2^{k-2}) + \dots + O(\log 2^1) + O(\log 2^0) = (k-1) + (k-2) + \dots + 2 + 1 = O(k^2) = O(\log n)^2$$

(b) INSERT: 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 get a new 4-array. Finally, use Merge routine to merge this new 4-array to the existing 4-array to 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? Note that, let i be the location of the rightmost 0 in the binary representation of n where $n_j = 1$ for all $j = 0, 1, 2, 3, \dots, i - 1$. So, the cost of an insertion when n items have already been inserted is $\sum_{k=0}^{i-1} 2 * 2^k = O(2^i)$. Well, about every alternate insertion costs 1, every 1/4-th insertion costs 2, every 1/8-th insertion costs 4 and so on. There will be at most $n/2^i$ insertions for each value of i . Now we can simply sum the cost over all the possible locations of i . So, the total cost is $O(\sum_{i=0}^{\log n} n/2^i * 2^i) = O(n \log n)$. Thus, the amortized cost of a single insertion is $O(n \log n)/n = O(\log n)$.

(c) DELETE:

- Find the smallest j for which the array A_j with 2^j elements is full. Let y be the last element of A_j .
- Let x be in the array A_i . If necessary, find which array this is by using the search procedure.
- Remove x from A_i and put y into A_i . Then move y to its correct place in A_i .
- Divide A_j (which now has 2^{j-1} elements left): The first element goes into array A_0 , the next 2 elements go into array A_1 , the next 4 elements go into array A_2 , and so forth. Mark array A_j as empty. The new arrays are created already sorted.

The cost of DELETE is $\Theta(n)$ in the worst case, where $i = k - 1$ and $j = k - 2$: $\Theta(\log n)$ to find A_j , $\Theta(\lg^2 n)$ to find A_i , $\Theta(2^i) = \Theta(n)$ to put y in its correct place in array A_i , and $\Theta(2^j) = \Theta(n)$ to divide array A_j . The following sequence

of n operations, where $n/3$ is a power of 2, yields an amortized cost that is no better: perform $n/3$ INSERT operations, followed by $n/3$ pairs of DELETE and INSERT. It costs $O(n \log n)$ to do the first $n/3$ INSERT operations. This creates a single full array. Each subsequent DELETE/INSERT pair costs $\Theta(n)$ for the DELETE to divide the full array and another $\Theta(n)$ for the INSERT to recombine it. The total is then $\Theta(n^2)$, or $\Theta(n)$ per operation.

3 Exercise 22.1-6

An universal sink will have two properties:

(i) All other vertices will have an edge into it. (ii) 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.

(i) 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. (ii) 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.

4 Exercise 22.2-7

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 from s . 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 .

$$d(t, u) \geq d(s, u) \tag{1}$$

$$d(t, u) \geq d(s, a) \tag{2}$$

$$d(t, u) \geq d(t, a) \tag{3}$$

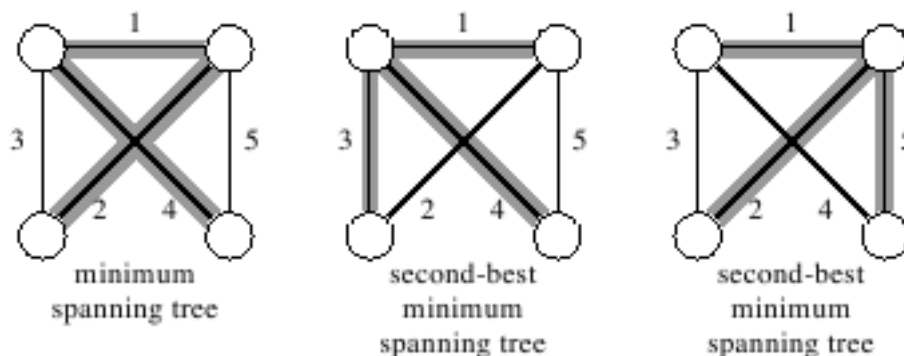
$$d(b, t) + d(t, u) \geq d(b, t) + d(t, a) \tag{4}$$

$$d(b, u) \geq d(b, a) \tag{5}$$

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.

5 Problem 23-1

(a) By Exercise 23.1-6, the minimum spanning tree is unique, since the graph is connected and all edge weights are distinct, then there is a unique light edge crossing every cut. The second-best minimum spanning tree need not be unique by the following counterexample which has a unique minimum spanning tree of weight 7 and two second-best minimum spanning trees of weight 8:



(b) Consider the fact that any spanning tree has exactly $|V| - 1$ edges, so any second-best minimum spanning tree must have at least one edge that is not in the (best) minimum spanning tree. If a second-best minimum spanning tree has exactly one edge, say (x, y) , that is not in the minimum spanning tree. This means it has the same set of edges as the minimum spanning tree with the exception that (x, y) replaces some edge. Let's denote that edge by (u, v) , which is one of the edges of the minimum spanning tree. Hence, $T' = T - \{(u, v)\} \cup \{(x, y)\}$.

Then, we need to show that by replacing two or more edges of the minimum spanning tree, we cannot obtain a second-best minimum spanning tree. Let T be the minimum spanning tree of G , and suppose that there exists a second-best minimum spanning tree T' that differs from T by two or more edges. There are at least two edges in $T - T'$, and let (u, v) be the edge in $T - T'$ with minimum weight. If we were to add (u, v) to T' , we would get a cycle c . This cycle contains some edge (x, y) in $T' - T$ (since otherwise, T would contain a cycle).

Let us claim $w(x, y) > w(u, v)$. We can prove this claim by contradiction, such that first assume the opposite, $w(x, y) < w(u, v)$. (Remember our assumption that edge weights are distinct, so that we do not have to concern ourselves with $w(x, y) = w(u, v)$.) When we add (x, y) to T , we get a cycle c' , which contains some edge (u', v') in $T - T'$, otherwise T' would contain a cycle. Therefore, the set of edges $T'' = T - \{(u', v')\} \cup \{(x, y)\}$ forms a spanning tree, and we must also have $w(u', v') < w(x, y)$, since otherwise T'' would be a spanning tree with weight less than $w(T)$. This gives us a contradiction. $w(u', v') < w(x, y) < w(u, v)$ contradicts with our choice of (u, v) as the edge in $T - T'$ of minimum weight.

Because the edges (u, v) and (x, y) would be on a common cycle c if we were to add (u, v) to T , the set of edges $T' - \{(x, y)\} \cup \{(u, v)\}$ is a spanning tree, and its weight is less than $w(T')$. Moreover, it differs from T . Note that, it differs from T' by only one edge. Thus, we have formed a spanning tree whose weight is less than $w(T')$ but is not T . This concludes our proof, T' was not a second-best minimum spanning tree.

(c) In order to fill in $max[u, v]$ for all $u, v \in V$ in $O(V^2)$ time, we can simply do a search from each vertex u , having restricted the edges visited to those of the spanning tree T . We can use any kind of search, like breadth-first, depth-first, and so on.

We will give pseudocode for breadth-first search approach. Given code is different than the pseudocode given in Chapter 22 in that we do not need to compute d or f values, and we will use the max table itself to record whether a vertex has been visited in a given search. In other words, $max[u, v] = NIL$ iff $u = v$ or we have not yet visited vertex v in a search from vertex u . Note also that since we are visiting via edges in a spanning tree of an undirected graph, we are guaranteed that the search from each vertex u will visit all vertices. Our pseudocode assumes that the adjacency list of each vertex consists only of edges in the spanning tree T .

Following is the pseudocode using breadth-first search:

```

BFS-FILL-MAX(T, w)
for each vertex  $u \in V$  do
  do
    for each vertex  $v \in V$  do
      do  $max[u, v] = NIL$ 
      Q = Empty
      ENQUEUE(Q, u)
      while Q is not Empty do

```

```

     $x = DEQUEUE(Q)$ 
    for  $v \in Adj[x]$  do
        if  $max[u, v] = NIL$  and  $v = u$  then
            if  $x = u$  or  $w(x, v) > max[u, x]$  then
                 $max[u, v] = (x, v)$ 
            else
                 $max[u, v] = max[u, x]$ 
            end if
             $ENQUEUE(Q, v)$ 
        end if
    end for
end while
end for
return  $max$ 

```

This algorithm fills in $|V|$ rows of the max table. Total time of operation can be calculated as follows: Since the number of edges in the spanning tree is $|V| - 1$, each row takes $O(V)$ time to fill in. Thus, the total time to fill in the max table is $O(V^2)$.

(d) Notice that in part (b), we showed that we can find a second-best minimum spanning tree by replacing just one edge of the minimum spanning tree T by some edge (u, v) not in T . If we create spanning tree T' by replacing edge $(x, y) \in T$ by edge $(u, v) \notin T$, then $w(T') = w(T) - w(x, y) + w(u, v)$. For a given edge (u, v) , the edge $(x, y) \in T$ that minimizes $w(T')$ is the edge of maximum weight on the unique path between u and v in T . Also, we can simply compute the max table from part (c) based on T , then the identity of this edge is precisely what is stored in $max[u, v]$. Now, we need to determine an edge $(u, v) \notin T$ for which $w(max[u, v]) - w(u, v)$ is minimum.

We can summarize the algorithm steps as follows:

- 1 Compute the minimum spanning tree T . Using Prim's algorithm with a Fibonacci-heap implementation of the priority queue, this will take $O(E + V \log V)$. Since $|E| < |V|^2$, this running time is $O(V^2)$.
- 2 Compute the max table for minimum spanning tree T , as in part (c) which takes $O(V^2)$.
- 3 Find an edge $(u, v) \in T$ that minimizes $w(max[u, v]) - w(u, v)$ in $O(E)$ time, which is again $O(V^2)$.
- 4 After finding the edge (u, v) in step 3, return $T = T - \{max[u, v]\} \cup \{(u, v)\}$ as a second-best minimum spanning tree.

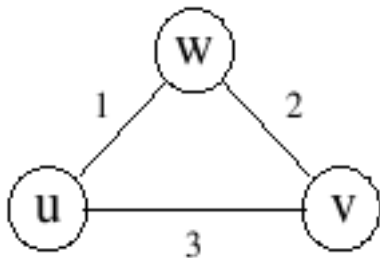
The total time is $O(V^2)$.

6 Problem 23-4

(a) Observe that MAYBE-MST-A removes edges in non-increasing order as long as the graph remains connected. Then, the resulting T is a minimum spanning tree. The correctness of the algorithm can be shown as follows: let S be a MST of G . When an edge e is about to be removed, either $e \in S$ or $e \notin S$. If $e \notin S$, then we can just remove it. If $e \in S$, removing e will disconnect S into two trees. Note that, this does not disconnect the tree. It is clear that no other edge can connect these trees with smaller weight than e , because by assumption S is a MST, and if a larger edge existed that connected the trees the algorithm would have removed it before removing e . Since the graph is still connected, this means a path exists. So, there must be another edge with equal weight that has not been discovered yet, which means we can remove e .

An adjacency list representation for T would be used for implementation. We can sort the edges in $O(E \log E)$ like using MERGE-SORT. We can check if $T - \{e\}$ is a connected graph by using BFS or DFS in $O(V + E)$. Then, the total running time is $O(E \log E + E(V + E)) = O(E^2)$.

(b) MAYBE-MST-B will not give a minimum spanning tree, that can be proven by a counterexample.



For the given graph G in figure, the MST would have edges (w, u) and (v, w) with weight 3. MAYBE-MST-B could add edges (u, v) and (v, w) to T , then try to add (w, u) which forms a cycle, then return T (weight 5), since the algorithm takes edges in arbitrary order.

This idea is similar to Kruskal's algorithm, then similarly we can use a UNION-FIND data structure to maintain T . For each vertex v we need to MAKE-SET(v). For each edge $e = (u, v)$, if $FIND - SET(u) \neq FIND - SET(v)$ then there is no cycle in $T \cup \{e\}$, and we UNION-SET(u, v).

In total, there are V MAKE-SET operations, $2E$ FIND-SET operations, and $V - 1$ UNION-SET operations. If we use an improved UNION-FIND data structure we can perform FIND-SET in $O(1)$ and UNION-SET in $O(\log V)$ time amortized to give a running time of $O(V) + O(E) + O(V \log V) = O(V \log V + E)$.

(c) MAYBE-MST-C successfully gives a minimum spanning tree. The algorithm adds edges in arbitrary order to T . If a cycle c is detected removes the maximum-weight edge on c . When an edge e is about to be added, either we form a cycle c or not. Let us assume e is added to some tree T' in T and

a cycle is c formed. Then we remove a maximum-weight edge e' from c and $T' - e' + e$ is of less or equal weight than T' . This is because e' is of greater or equal weight than e . In case no cycle is formed, we either just add e to some tree in T or e connects two trees in T . In the second case if e is not the smallest weight edge that connects the trees then we have not discovered it yet, and when we eventually form a cycle we have already shown MAYBE-MST-C performs correctly.

Implementation would use an adjacency list representation for T , similar to part (a). For each edge, we add it to T and check $T \cup \{e\}$ for cycles. There can be at most one cycle which we can detect by DFS and return it. Otherwise, when we have no cycles, we are done with this edge. When there is a cycle, we need to put returned cycle, find the maximum-weight edge and delete it from T .

Time complexity can be analyzed as follows: $O(1)$ time for adding an edge. DFS takes $O(V + E) = O(V)$ if we break it as soon as T has a cycle, so number of edges in T at any point is no greater than V . Finding the maximum-weight edge on the cycle would take $O(V)$ time and deleting an edge would take $O(V)$ time. For each edge we added, we will perform DFS, and might find a cycle, so the total running time becomes $O(EV)$.

7 Problem 24-6

Observe that a bitonic sequence can increase, then decrease, then increase, or it can decrease, then increase, then decrease, since a cyclic shift of a bitonic sequence is again bitonic. That is, there can be at most two changes of direction in a bitonic sequence. A bitonic sequence would be a subsequence of any sequence that increases, then decreases, then increases, then decreases.

First, we assume our problem has the following property instead of bitonic sequence one: for each vertex $v \in V$, the weights of the edges along any shortest path from s to v are increasing. In that case, in order to find the shortest paths we could simply call INITIALIZE-SINGLE-SOURCE and then just relax all edges one time, going in increasing order of weight. The edges along every shortest path would be relaxed in order of their appearance on the path. Note that, we assume the uniqueness of edge weights to ensure that the ordering is correct. The path-relaxation property in Lemma 24.15 would guarantee that we would have computed correct shortest paths from s to each vertex.

Second, we assume our problem has another property instead of bitonic sequence one: the weights of the edges along any shortest path increase and then decrease. In that case, we could simply relax all edges one time, in increasing order of weight, and then one more time, in decreasing order of weight. Since edge weights are unique, this order would ensure that we had relaxed the edges of every shortest path in order. Hence, the path-relaxation property would guarantee that we would have computed correct shortest paths.

Let us apply the above mentioned ideas to solve our problem, which ensures bitonic sequences. We need to perform four passes to cover all possible bitonic

sequences as mentioned earlier. We will relax each edge once in each pass. In particular, the first and third passes relax edges in increasing order of weight, and the second and fourth passes in decreasing order. We ensure the correctness by the path-relaxation property together with the uniqueness of edge weights. So, we have computed correct shortest paths.

The total time for this procedure is $O(V + E \log V)$ which consists of sorting $|E|$ edges by weight in $O(E \log E) = O(E \log V)$ (since $|E| = O(V^2)$), INITIALIZE-SINGLE-SOURCE in $O(V)$ and $O(E)$ for each relaxation passes. Hence, this will give $O(E \log V + V + E) = O(V + E \log V)$ time complexity.

Note: Solutions given are not the only possible solutions, there can be many different solutions.