95

Name(last, first): Li, Justin

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** ID :

Midterm **Total Time: 1.5 hours** **October 31, 2017**

Each problem has 20 points.

All algorithm should be described in English, bullet-by-bullet

20

1 Consider a set of intervals I1, …, In. **a.** Design a linear time algorithm (assume intervals are sorted in any manner you wish) that finds a maximum subset of mutually non-overlapping intervals. **b.** Prove the correctness of your algorithm.

Let s_i be the start of interval I_i, and f_i the end of interval I_i

a) · Assume intervals are sorted by f_i
· While there are still intervals left
· Choose the interval I_s with the smallest f_i
· Add I_s to the list of intervals returned
· For every interval that overlaps with I_s, remove it from consideration
· endWhile
· Return the list of intervals chosen.

b) Assume by contradiction that there exists an optimal solution I^* with m intervals, while our greedy solution I^G has only l intervals, with $m > l$

· Suppose the first k intervals of both solutions are the same
· For I^*, I^*_{k+1} must have a finish time at least as great as I^G_{k+1} since our greedy algorithm chose the interval with the smallest possible finish time
· We can transform I^* by changing I^*_{k+1} to I^G_{k+1} and still maintain a valid interval scheduling, since if $s^*_{k+2} > f^*_{k+1}$ and $f^*_{k+1} \geq f^G_{k+1}$, clearly $s^*_{k+2} \geq f^G_{k+1}$

- So we can go from the first k intervals in common to the first k+1 intervals in common while maintaining the same # of total intervals

- Inductively, we can make the first l intervals in common
- At this point, the optimal solution I^* cannot possibly have more intervals, since if it did, the greedy solution would have picked it as well.
- But the greedy solution didn't, so the optimal solution I^* does not have more intervals than our algorithm I^G, contradiction
- Therefore, the greedy algorithm is optimal.

Name(last, first): Li, Justin

18

Dijkstra

2 . a. Design an efficient algorithm better than O (n^2) to be used in sparse graphs for finding the shortest path between two vertices S and T in a positive weighted graph. **b.** Justify the correctness of your runtime analysis.

a) This is just Dijkstra's algorithm

- Define a set of explored nodes N_E
- For each node n, keep track of their current minimum distance d[n] to S with a heap
- Add S to the N_E
- Set d[s] to 0
- While($\exists$ a node N s.t. $n \notin N_E$ and $d[n] < \infty$)
  - Choose node n^* with the smallest $d[n^*]$
  - Fix $d[n^*]$ by removing it from the heap
  - Add n^* to N_E
  - For every node $u \notin N_E$ s.t. there is an edge between n^* and u,
    - update d[u] How?
  - If $n^* == T$
    return $d[n^*]$

Endwhile

return ∞ //no path between S and T

b) This runs in O(elogn), Since the graph is sparse, $O(e) < O(n^2)$

So overall the algorithm runs better than $O(n^2)$

For every edge e in the graph, we must update a value in the heap and reheapify the heap. Heapifying the heap takes O(logn) time, so this part of the algorithm takes O(elogn)

For each node we only do constant work, but we only check nodes that are connected to S, so the work done here is $O(n) \cdot O(e) < O(elogn^2)$

Name(last, first): Li, Justin

20

3. Consider a sequence of positive and negative (including zero) integers. Find a consecutive subset of these numbers whose sum is maximized. Assume the weight of an empty subset is zero. a. Design a linear time algorithm. b. Prove the correctness of your algorithm.
Example: For the sequence 2 -3 5 -12 the maximum sum is 4.

a) · let s be the current sum.
- Let m be the maximum sum encountered so far
- Set s=0
- Set m=0
- Loop through the sequence in order:
- while we have not reached the end of the list
  - Let n be the next element
  - Set s=s+n
  - If m<s
    * set m=s
  - If s<0
    * set s=0

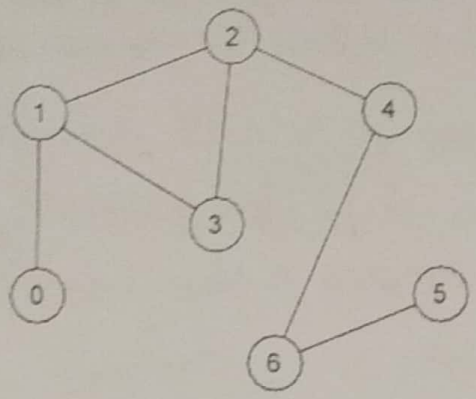
endwhile
return m

b) Suppose there exists an optimal solution O* that is greater than ours
· It suffices to show that our greedy solution considered the optimal solution since if it considered it, the greedy solution couldn't have had a worse solution, contradiction.
· Since O* is optimal, there couldn't have been continuous subsets next to O* that summed to positive values, or we could just add those to O* and get a better solution
· Similarly, there couldn't have been continuous subsets starting at one of the ends in O* and continuing to the middle of O* that are negative, or else we could subtract those out from O* and get a better solution.

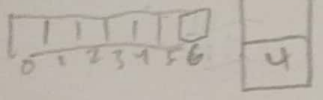
Name(last, first): Li, Justin

20

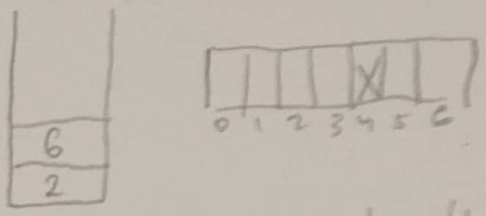
4. Consider an unweighted graph G shown below. **a.** Starting from vertex **4**, show every step of DFS along with the corresponding stack next to it. **b.** What is the run time of DFS if the graph is not connected (no proof is necessary)?



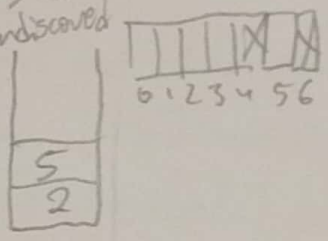
a) ① Discovered



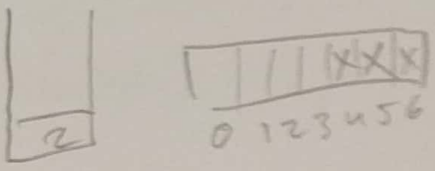
② Pop 4 from the stack and add its undiscovered neighbors
- Mark 4 as discovered



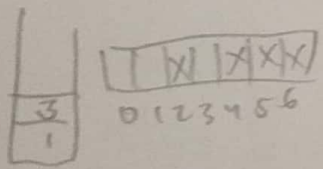
③ Pop 6 from the stack and add its undiscovered neighbors Mark 6 as discovered



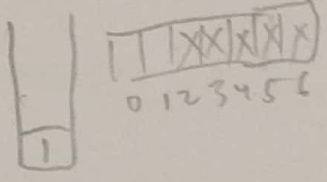
④ Pop 5 from the stack, add its undiscovered neighbors, mark it as discovered



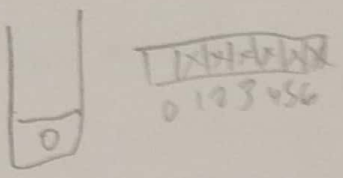
⑤ Pop 2 from the stack, add its undiscovered neighbors mark as discovered



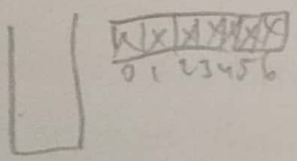
⑥ Continue in this way with 3



⑦ With 1



⑧ With 0



⑨ The stack is empty, so you're done

b) The run time is still O(n+e)

17

Name(last, first): Li, Justin

5. Consider a binary tree (it is not necessarily balanced). The tree is not rooted. Its diameter is the distance between two vertices that are furthest from each other (distance is measured by the number of edges in a simple path). Design a linear time algorithm that finds the diameter of a binary tree.

From now on we consider only the BFS tree

- Perform BFS on the tree, but each time a node is discovered, add it to a stack S.
- For each node n, maintain d[n], the maximum distance we have seen so far, and m[n], the maximum distance from that node to a leaf
- Initially, d[n]=m[n]=0 ∀n
- While S is not empty
  - pop a node n from S
  - If n has no child nodes, set d[n]=m[n]=0
  - If n has 1 child node c_1, set m[n]=m[c_1]+1, d[n]=max(m[n], d[c_1])
  - If n has more than 1 child node, find the two largest m[c_i], call them m[c_1] and m[c_2] set m[n]=max(m[c_1], m[c_2]) set d[n]=max(d_i, m[c_1]+m[c_2]+2)

endwhile

return d[root of tree]

Analysis:
- Each node is considered only constant number of times by each parent
- When n has more than 1 child node, it can have a max of 3, since the original tree was binary, so those computations are done in constant time
- BFS runs in linear time

- Whenever we pop a node from a stack, we already have the necessary information from all its children, since we are looping through the nodes in the reverse order we added them to the BFS tree
- For each node n, d[n] is the maximum distance for any path in the subtree of the BFS tree rooted at n
  - So by the time we get to the root of the BFS tree, we have considered the entire tree
- It should be noted that the BFS tree and binary tree have exactly the same nodes and edges