CS180-Spring20 Midterm

Bryan Wong

TOTAL POINTS

99 / 100

QUESTION 1

1 T of F 20 / 20

✓ - 0 pts **Correct**

- **10 pts** No justification for any of the answers
- **5 pts** Part justification
- **5 pts** a wrong
- **5 pts** b wrong
- **5 pts** c wrong
- **5 pts** d wrong
- **7.5 pts** missing 3 justifications
- **2.5 pts** b justification is wrong

QUESTION 2

2 Order of growth 10 / 10

- **1 pts** 1 incorrect pair.
- **2 pts** 2 incorrect pairs.
- **3 pts** 3 incorrect pairs.
- **4 pts** 4 incorrect pairs.
- **5 pts** 5 incorrect pairs.
- **6 pts** 6 incorrect pairs.
- **7 pts** 7 incorrect pairs.
- **8 pts** 8 incorrect pairs.
- **9 pts** 9 incorrect pairs.

✓ - 0 pts **Correct**

- **10 pts** 10 incorrect pairs.

QUESTION 3

3 DAG 19 / 20

✓ - 0 pts **Correct**

- **20 pts** No answer
- **1 pts** Hard to read
- **2 pts** Correct idea, but needs a more formal algorithm description
- **2 pts** No explanation for algorithm's correctness

✓ - 1 pts **Needs better explanation for algorithm's correctness. Need to justify that your algorithm returns true if and only if G contains such a path.**

- **8 pts** Algorithm produces correct output, but with incorrect time complexity.
- **10 pts** Incorrect, but a complete answer is given
- **1 pts** Algorithm description needs some additional detail.

QUESTION 4

4 Array 20 / 20

✓ - 0 pts **Correct**

- **2 pts** no proof of correctness or explanation

QUESTION 5

5 Flying saucers 30 / 30

✓ - 0 pts **Correct**

- **10 pts** no correctness proof
- **2 pts** hard to read
- **5 pts** no time complexity proof
- **2 pts** incomplete correctness proof
- **5 pts** wrong time complexity proof
- **10 pts** incorrect algorithm
- **30 pts** no answer
- **5 pts** incomplete algorithm
- **0 pts** late submission
- **5 pts** Not a regious description (or hard to follow)
- **0 pts** Click here to replace this description.

CS 180 Midterm

Bryan Wong

May 7, 2020

Question 1

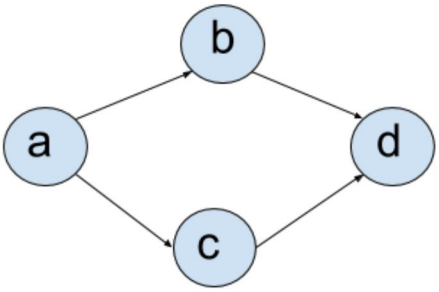
For each of the following problems answer True or False and briefly justify you answer.)

a. For a connected and undirected graph G, if removing edge e disconnects the graph, then e is a tree edge in DFS of G.

True. If removing edge e disconnects the graph, then that means there is some node v that is only connected to the rest of the graph by edge e. In a DFS, all connected nodes are part of the DFS tree. Since e is the only edge that connects to node v, edge e must be a tree edge in the DFS of G.

b. For a DAG G, if there is only one node with no incoming edge, then there exists only one topological ordering.

False, Counterexample:



The pictured DAG has only one node a with no incoming edge, but it has two valid topological orderings:
a b c d
a c b d

c. For the stable matching problem, if there is a man m_1 and woman w_1 such that w_1 has the lowest ranking in m_1's preference list and m_1 has the lowest ranking in w_1's preference list, then any stable matching will not contain the pair (m_1, w_1).

False, counterexample:
$m_1 : w_2 > w_1$
$m_2 : w_2 > w_1$
$w_1 : m_2 > m_1$
$w_2 : m_2 > m_1$

If use the Gale-Shapley algorithm with m_2 proposing before m_1, m_2 will propose to and be engaged to w_2 since w_2 was previously unmatched. Then, m_1 will propose to his first choice, w_2. However, w_2 prefers m_2 and the proposal fails. m_1 will then propose to his second choice, w_1, who accepts because she is unmatched. We end up with a stable matching containing (m_1, w_1) despite each other being their partner's lowest ranking.

If we run DFS on a DAG and node u is the first leaf node in the DFS tree, then u has no outgoing edge.

True. By definition of DFS, the algorithm will follow the first edge out of an arbitrary starting node v and continue until it finds the first leaf node u with a "dead end" where either:
1) It has no outgoing edge
2) It has neighbors, but they have already been explored

In the case of a DAG, case 2 is impossible because this would mean that there is an edge from u to some node w, where w is on the path from $v \rightarrow u$. This would create a cycle. This contradicts the fact that the graph is a DAG. Therefore, it is true that u has no outgoing edge.

1 T of F 20 / 20

✓ - **0 pts** **Correct**

- **10 pts** No justification for any of the answers
- **5 pts** Part justification
- **5 pts** a wrong
- **5 pts** b wrong
- **5 pts** c wrong
- **5 pts** d wrong
- **7.5 pts** missing 3 justifications
- **2.5 pts** b justification is wrong

Question 2

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n)).

- $f_1(n) = 3n^3$
- $f_2(n) = n(logn)^{100}$
- $f_3(n) = 2^{nlogn}$
- $f_4(n) = 2^{\sqrt{n}}$
- $f_5(n) = 2^{0.8logn}$

Let us take the log (base 2) of all functions:

- $f_1(n) = log(3) + 3logn$
- $f_2(n) = logn + 100log((logn))$
- $f_3(n) = nlogn$
- $f_4(n) = \sqrt{n}$
- $f_5(n) = 0.8logn$

If we substitute $z = logn$ into each function, we get:

- $f_1(n) = log(3) + 3z$
- $f_2(n) = z + 100log(z)$
- $f_3(n) = nz$
- $f_4(n) = \sqrt{n}$
- $f_5(n) = 0.8z$

We see that f_5 has the lowest coefficient of z, and thus grows the slowest. The next largest coefficient is f_2 (where the z term dominates the $100log(z)$ term), followed by f_3. $logn$ grows at a slower rate than $\sqrt{n}$, so the next function in our list is f_4. Of course, $\sqrt{n}$ grows slower than nz, so f_5 is the last item in our list.

The final order is as follows:
f_5, f_2, f_1, f_4, f_3

2 Order of growth 10 / 10

- **1 pts** 1 incorrect pair.
- **2 pts** 2 incorrect pairs.
- **3 pts** 3 incorrect pairs.
- **4 pts** 4 incorrect pairs.
- **5 pts** 5 incorrect pairs.
- **6 pts** 6 incorrect pairs.
- **7 pts** 7 incorrect pairs.
- **8 pts** 8 incorrect pairs.
- **9 pts** 9 incorrect pairs.

✓ **- 0 pts Correct**

- **10 pts** 10 incorrect pairs.

Question 3

For a DAG with n nodes and m edges (and assume $m \geq n$), design an algorithm to test if there is a path that visits every node exactly once. The algorithm should run in $O(m)$ time.

Assume we know the number of nodes n, and we have a list of edges E.

Algorithm:

Run a topological sort to get the topological ordering $v_1, v_2, \ldots, v_n$
For i $= 1, 2, \ldots, n-1$:
    if $(v_i, v_{i+1}) \notin E$):
        return false
return true

By running a topological sort, the topological ordering of nodes will contain a path that visits every node exactly once, if such a path exists. Running the topological sort takes $O(m+n)$ time, and searching for an edge between every pair of topologically adjacent nodes takes $O(m)$ time with an adjacency list implementation. Since m dominates n, the overall time complexity is $O(m)$.

3 DAG 19 / 20

✓ **- 0 pts Correct**

- 20 pts No answer

- 1 pts Hard to read

- 2 pts Correct idea, but needs a more formal algorithm description

- 2 pts No explanation for algorithm's correctness

✓ **- 1 pts Needs better explanation for algorithm's correctness. Need to justify that your algorithm returns true if and only if G contains such a path.**

- 8 pts Algorithm produces correct output, but with incorrect time complexity.

- 10 pts Incorrect, but a complete answer is given

- 1 pts Algorithm description needs some additional detail.

Question 4

Given an array A of n distinct integers and assume they are sorted in increasing order. Design an algorithm to find whether there is an index i with $A[i] = i$. The algorithm should run in O(log n) time.

Find(A, l, u) :
    mid $\leftarrow l + \lfloor \frac{u-l}{2} \rfloor$
    if A[mid] = mid:
        return true
    if $u - l = 0$:
        return false
    if A[mid] $<$ mid:
        return Find(A, $mid + 1$, u)
    if A[mid] $>$ mid:
        return Find(A, l, mid)

Since we have a sorted array of distinct integers, a value of i such that $A[i] = i$ will cause:
Indexes $j < i$ to have values $A[j] \leq j$
Indexes $j > i$ to have values $A[j] \geq j$

Therefore, we can use a binary search algorithm to locate such an index i if it exists. By taking the midpoint index and comparing it with the value of A at that index, we can recursively search either the top or bottom half. The base case where we know our search has failed is if we get down to searching a single element that does not match its index. Therefore, the maximum runtime for this algorithm is $O(log_2 n)$, giving us a runtime complexity of $O(logn)$.

4 Array 20 / 20

✓ - 0 pts Correct

- 2 pts no proof of correctness or explanation

Question 5

There are several flying saucers on the sky to attack the Earth. For simplicity, we assume Earth surface is 1-D and the flying saucers are on the sky, as shown in Figure 1. We know there are n flying saucers and each of them occupies the open interval (L_i, R_i) (assume L_i, R_i are integers). To destroy those flying saucers, we are going to fire the laser canon at some locations. If the laser canon is fired at position x to the sky, it will destroy all the saucers that intersects with this vertical line, i.e., all the flying saucers with $x \in (L_i, R_i)$ will be destroyed, as illustrated in Figure 1. However, firing the laser canon is expensive so we want to find a way to destroy all the flying saucers using as few laser canons as possible.

Mathematically, given n intervals $(L_i, R_i)|i = 1, \ldots, n$, our goal is to find a minimum set of numbers $X = x_1, \ldots, x_k$ such that for every interval i, there is at least one x_j in X contained in the interval $(L_i < x_j < R_i)$. Give an $O(nlogn)$ time algorithm to solve this problem, and prove the correctness of your algorithm.

Algorithm:

Sort all flying saucers in ascending order by L_i into a min heap S. Now, i refers to each saucer's sorted order, where $i = 1$ is the smallest L_i and $i = n$ is the largest L_i. Ties are broken arbitrarily.

Initially let S be the sorted min heap of all saucers $S_i = (L_i, R_i)$ and let X be empty
right $\leftarrow \infty$ // holds the maximum overlap value
While S is not empty
  $S_k \leftarrow S$.top(): // (the one that currently has the smallest L_i)
  If $L_k <$ right: // If S_k is compatible
    S.pop() // remove compatible saucer from set
    right $\leftarrow \min(R_k,$ right) // Pick more restrictive of the two
  Else:
    Add $\frac{right+L_{k-1}}{2}$ to X
    right $\leftarrow \infty$ // Ready for new laser
return X

This algorithm contains a heap sort, which takes $O(nlogn)$. Then, it pops the root value from the heap (each pop takes $O(logn)$) until the heap is empty. This means it must delete n nodes, for a time complexity of $O(nlogn+nlogn)$. Overall, the time complexity is $O(nlogn)$.

Proof of Correctness:
To begin, we sort all saucers by the x coordinate of their left edge L_i. This means that we know for indexes $k > j$ that $L_k > L_j$ is always true. To prove that the saucers at k and j overlap, all we need to show is that $L_k < R_j$. This is the basis for our algorithm. We initially choose the first sorted saucer S_1 and set the variable $right$ to hold R_1.

Then, we iterate through the remaining saucers S_n in order. If $L_n < right$, we know the saucers overlap (and thus can be destroyed with a single laser). We then set $right$ to be the smallest R_i of any saucer encountered thus far. The reasoning behind this is that if the latest saucer is compatible with the right edge of the most restrictive saucer, it will also be compatible with less restrictive saucers. In addition, since L_i is always increasing due to the saucers being in sorted order, our left edge will continually get more restrictive as well. As we add more saucers to our laser, the possible coordinates of the laser get more restrictive as L_i increases and $right$ decreases.

Once we find the first non-compatible saucer S_m such that $L_m \geq right$, subsequent saucers will also be incompatible. This is because the value of L_i is always increasing, due to being in sorted order. At this point, we have a set of saucers that can be destroyed with a single laser at a x coordinate between the left edge of the latest compatible saucer L_{k-1} and the most restrictive right edge so far, $right$. We fire at the exact midpoint of this, $\frac{right+L_{k-1}}{2}$.

This algorithm is optimal because it continues adding saucers to the current laser set until it finds a saucer that is incompatible. At this point, there is at least one saucer S_w in the set that is incompatible with all following saucers, so we must fire a laser for it. By grouping other compatible saucers with S_w, we can only make our solution more efficient. We continue this process, forming an optimal, minimized number of coordinates to shoot lasers at.

5 Flying saucers 30 / 30

✓ **- 0 pts Correct**

- 10 pts no correctness proof

- 2 pts hard to read

- 5 pts no time complexity proof

- 2 pts incomplete correctness proof

- 5 pts wrong time complexity proof

- 10 pts incorrect algorithm

- 30 pts no answer

- 5 pts incomplete algorithm

- 0 pts late submission

- 5 pts Not a regious description (or hard to follow)

- 0 pts Click here to replace this description.