CS180 Midterm

Ryan Lin

May 2020

UID: 005131227 Discussion: 1B

Problem 1

(a) (5pt) For a connected and undirected graph G, if removing edge e disconnects the graph, then e is a tree edge in DFS of G.

Solution: True. Proof by contradiction: assume there exists a DFS tree that does not contain an edge e that, when removed, disconnects the G. We know that DFS traversal visits every reachable node from a particular starting node, and since it is given that G is connected, DFS from any node should produce a DFS tree containing every node. This produces a contradiction because if e is not included as a tree edge it is impossible for the DFS to visit every node, as every node on the other side of e ("other side" meaning the nodes that would be disconnected from the starting node if e were to be removed) is only reachable through a path containing e.

(b) (5pt) For a DAG G, if there is only one node with no incoming edge, then there exists only one topological ordering.

Solution: False. A counterexample: Let's say a graph consisting of nodes A,B,C,D has edges $A \to B$, $A \to C$, $C \to D$, $B \to D$. Then A is the only node with no incoming edges. However there are two possible topological sorts: A,B,C,D and A,C,B,D.

(c) (5pt) For the stable matching problem, if there is a man m1 and woman w1 such that w1 has the lowest ranking in m1's preference list and m1 has the lowest ranking in w1's preference list, then any stable matching will not contain the pair (m1, w1).

Solution: False. A counterexample: Preference Lists:

$$m_1 : w_2 > w_1$$
$$m_2 : w_2 > w_1$$
$$w_1 : m_2 > m_1$$
$$w_2 : m_2 > m_1$$

$(m_2, w_2), (m_1, w_1)$ is a valid matching for this particular preference list. m_1 and w_1 both ranked each other as lowest in their respective lists but they can still be paired together if the other men and women rank m_1 and w_1 as their lowest too.

(d) (5pt) 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.

Solution: True. A leaf node in a DFS tree is created if the visited node has no outgoing edges or if the adjacent nodes have already been visited. Since we are given that node u is the first leaf node in the DFS tree, we know that only a single path has been taken so far. If the leaf node has an adjacent node that has already been visited, this will form a cycle, which contradicts what we have been given which is that the graph is a DAG. Therefore the leaf node must be the case where it has no outgoing edges.

Problem 2

(10pt) 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)).

Note: in the following derivations < is synonymous with "upper bounded by".

I took the log of the last three functions to make them easier to compare.

- $f_3(n) = 2^{n \log n} \to n \log n \cdot \log 2$
- $f_4(n) = 2^{\sqrt{n}} \to \log 2^{\sqrt{n}} = \sqrt{n} \log 2$
- $f_5(n) = 2^{0.8 \log n} \to \log 2^{0.8 \log n} = 0.8 \log(n) \cdot \log 2$

It is then easy to conclude that $f_5 < f_4 < f_3$ since $\log n$ is upper bounded by $\sqrt{n}$ which is upper bounded by $n \log n$. Next, we consider the first two functions in the following manner:

$f_1(n) \overset{?}{=} f_2(n)$
$3n^3 \overset{?}{=} n(\log n)^{100}$
$3n^2 \overset{?}{=} (\log n)^{100}$
$3n^{2/100} \overset{?}{=} \log n$
$3 \cdot 10^{n^{2/100}} \overset{?}{=} n$
$3 \cdot 10^{n^{2/100}} > n$

We know that exponential functions grow faster than linear ones so $f_1 > f_2$. Next we show that $f_2 < f_4$

$n(\log n)^{100} \overset{?}{=} 2^{\sqrt{n}}$
$\log n + \log((\log n)^{100}) \overset{?}{=} \sqrt{n} \log 2$
$\log n + 100 \log(\log n) \overset{?}{=} \sqrt{n} \log 2$
$\log n < \sqrt{n}$

In the last line, we ignore the log(log n) since log n is greater and the log 2 on the r.h.s. since we don't care about constants. Finally, we show that $f_5 < f_1$:

$f_5 \overset{?}{=} f_1$
$2^{0.8 \log n} \overset{?}{=} 3n^3$
$0.8 \log n \cdot \log 2 \overset{?}{=} 3 \log 3n$
$\log n^{0.8} \overset{?}{=} \log 3n$
$n^{0.8} \overset{?}{=} n$
$n^{0.8} < n$
$f_5 < f_1$

The final answer is then $f_5 < f_2 < f_1 < f_4 < f_3$

Problem 3

(20pt) 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

Solution: My solution will take advantage of the fact that every DAG has a topological order. For the DAG to have a path that visits every node exactly once (also known as a Hamiltonian Path), every two nodes listed adjacent to each other in the topological sort needs to be connected by an edge directed from the left to right. For example, if the topological sort of a particular graph was A, B, C, D, there would need to be an edge $A \rightarrow B$, $B \rightarrow C$, and $C \rightarrow D$ for that graph to have a Hamiltonian Path.

Proof by Contradiction: Assume there exists a DAG with a Hamiltonian Path with its topological ordering consisting of two arbitrary nodes A and B listed in succession such that there is no edge directed from A to B. By property of topological sort of a DAG, there cannot be any backwards pointing edges, since this would form a cycle. This suggests that the only way for A and B to be connected to each other is through an edge directed from A to B since $B \rightarrow A$ would form a backwards edge and trying to jump from A to another node to B will form a backwards edge as well. This forms a contradiction with our initial statement because without any path to reach node B, no Hamiltonian Path can be formed since it would require a path that visits all nodes once. We can then conclude that to show that a graph has a Hamiltonian Path, there needs to be an edge directly between any adjacent two nodes listed in the graph's topological ordering.

The algorithm is then as follows: perform topological sorting on the graph, then check every pair of adjacent nodes listed in the topological order and check if there is a directed edge from the left to right node.

Algorithm 1 Check if given graph has a Hamiltonian Path

```
1: procedure hasHamiltonian(g)
2:     sorted = toposort(g)
3:     idx = 0
4:     for idx < sorted.len-1 do
5:         if no edge from sorted[idx] to sorted[idx+1] then
6:             return false
7:     return true
```

It has been shown in class that topological sorting runs in $O(m)$ time. Looping through the ordering takes $O(n)$ time and checking for an edge is $O(1)$ with an adjacency array, adding up to a total time of $O(m + n)$ which is simplified to $O(m)$ as we are told that $m \geq n$.

Problem 4

(20pt) 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.

Solution: This problem can be solved in a simple recursive binary-search-like technique. We first examine the middle element of the array by finding its index with integer division $floor(size/2)$. If the value of the middle element is greater than the index, we know that an element where $a[i] = i$ can only possibly exist in the lower half of the array. Similarly, if the value is less than the index, we know that the element where $a[i] = i$ can only exist in the upper half of the array. This is logical because we are told that the array given is sorted in ascending order. We can thus recursively call this procedure on the half that we have determined might have the target element, narrowing down the range of our search by half every iteration. We stop once we have found an element satisfying $a[i] = i$ or if our search range reaches 0, which indicates such an element does not exist. Because this solution is effectively a binary search, it runs in $O(\log n)$ time. Below is the pseudocode implementing the solution.

Algorithm 2 Check if given array has an element matching its index

```
procedure indexMatchesValue(a,n,s)
    if n == 0 then
        return false
    range = floor(n/2)
    idx = s+range
    if a[idx] > idx then
        return indexMatchesValue(a, range, s)
    else if a[idx] < idx then
        return indexMatchesValue(a, range, s+range)
    else
        return true
```

The inputs to $indexMatchesValue$ are a, the input array; n, the current search size; and s, the index of the element marking beginning of the range of elements that still have a possibility to contain an element satisfying $a[i] = i$. The base case for this procedure is if n is 0, indicating that the range of candidate elements have reached 0 and the binary search has terminated without finding the desired element, in which we return false and the algorithm terminates. Otherwise, we examine the middle element of our current search range given by adding the current size of range divided by 2 to the starting index s. If the value of this element is greater than its index, we recursively call $indexMatchesValue$ on the lower half of the current search range by keeping the starting element s the same and decreasing the range of search by a factor of 2. If the value of this element is greater than its index, we recursively call $indexMatchesValue$ on the upper half of the current search range by moving the starting element s to the found middle element and decreasing the range of search by a factor of 2 as well. If the element value matches its index, we have found the value satisfying the condition $a[i] = i$ and we return true. To use this algorithm we call $indexMatchesValue(a, a.len, 0)$, where a is the given array and $a.len$ is the length of the array. We give the starting search element 0 initially because we want to consider the entire array in our first iteration.

Problem 5

Solution:

Algorithm 3

```
1: procedure FindLaserPositions(intervals[])
2:     sorted = SortByStartDistance(intervals)
3:     laserPositions = []
4:     range = [$L_0$, $R_0$] //first element in sorted
5:     for each interval ($L_i$,$R_i$) in sorted excluding the first element do
6:         if $L_i$ < range[1] then
7:             range[0] = $L_i$
8:             range[1] = min(range[1], $R_i$)
9:         else
10:            laserPositions.append((range[0]+range[1])/2)
11:            range[0] = $L_i$
12:            range[1] = $R_i$
13:    return laserPositions
```

FindLaserPositions first performs a sort on the intervals in ascending order by the start (left) distance of the saucer, indicated in the procedure as *SortByStartDistance*(*intervals*). Next, we initialize an array called *range*, which holds the minimum and maximum of the range of x values which are candidates of being chosen as an optimal laser firing location. This is initially set to the saucer range with the smallest L distance. Next, we iterate through the rest of the saucer intervals. If the left value of the saucer being processed is less than the right value of the current range, this indicates that the saucer intersects with at least a portion of the range of locations under consideration to be the laser-firing location. In this case, we constrict the range to ensure any value within the range will also intersect with the saucer in the current iteration. This is done by setting the L distance of the range to the L value of the current saucer (the L distance side of the current saucer is guaranteed to be larger since we sorted the saucers by increasing L value) and setting the R value of the range to the minimum of the current range R value and the processed saucer R value. On the other hand, if the saucer's L value is greater than the current range R value, the range does not overlap with any part of this next saucer and thus the range of the laser is finalized and a new laser needs to be used to hit the new saucer. We can technically choose any value from *range* to be the firing location, but I chose to pick the middle point by averaging the R and L value of *range*. After the laser location is determined and appended to our final result *laserPositions*, we set *range* to the R and L value of the new saucer to find the next laser location. After all saucer intervals have been processed, our final result is given as a set of x locations such that if lasers are fired from them all saucers will be hit. Since sorting is an $O(n \log n)$ operation and the rest of the algorithm is done in linear time, the overall runtime is $O(n \log n)$.

Now I will show that this greedy algorithm produces an optimal solution. I first define the optimal solution to be one that has exactly the amount of laser firing positions as the size of the maximum set of disjoint saucers. Concretely, let's say we have k saucers with their start and endpoints disjoint from each other. Then logically we would need at least k lasers to hit all of them; we cannot hit more than one of these k saucers with a single laser since at no point do any of them overlap with one another. Futhermore, I mentioned *maximum set* to indicate that we need to consider the set of disjoint saucers with the largest number of saucers, as considering anything less will leave one of these saucers untouched by a laser due to their disjoint nature. This in turn indicates that an optimal solution to this problem consists of one laser per saucer in the largest set of disjoint saucers.

From the description of my algorithm, we know that every time a new saucer is processed we try to narrow down our current range of values to ensure that this saucer will also be hit if a value from our range is chosen as the firing position. When a saucer encountered is disjoint to the current range, we logically need another laser for it. In particular, I argue that whenever such a partition (partition meaning an x value separating two disjoint saucers) creating this disjointness is encountered in my algorithm, it is one that belongs to the maximum set of disjoint saucers for our given set of saucers. Because we visit each saucer in the order of ascending start position, we will always encounter the earliest partition (and therefore find the smallest disjoint saucers) between disjoint saucers first. This property can be proven through contradiction: let's assume there exists a maximum set of disjoint saucers such that the first partition encountered is not part of it. This would not be the maximum set of disjoint saucers, as we can simply add this partition to indicate more disjoint saucers, thus proving this property.

Because we know that our algorithm only adds a new laser position when it encounters a partition and we know this partition is present in defining maximum set of disjoint saucers, we can conclude that our algorithm produces only as many laser positions as one in the optimal solution, making our solution optimal as well.