

CS180-Fall20 Midterm-10am

CHARLES XIAN ZHANG

TOTAL POINTS

93 / 100

QUESTION 1

1 Q1 18 / 20

- 0 pts Correct
- ✓ - 2 pts **BFS may not find all nodes**
- 2 pts No need to start BFS from a node without incoming edge in DAG
- 4 pts Proof of correctness is not complete or partially correct
- 8 pts Proof of correctness is wrong/missing
- 2 pts Time complexity analysis is partially correct
- 4 pts Time complexity analysis is wrong/missing
- 8 pts BFS search is wrong

QUESTION 2

2 Q2 20 / 20

- ✓ - 0 pts **Correct**
- 10 pts incorrect algorithm
- 5 pts incorrect time complexity
- 5 pts incorrect proof
- 3 pts unclear algorithm statement
- 2 pts time complexity can be more precise
- 2 pts informal/incomplete proof

QUESTION 3

3 Q3 20 / 20

- 8 pts Incorrect/No algorithm
- 4 pts Unclear/partially correct algorithm
- 8 pts Incorrect/No proof
- 4 pts Unclear/informal/incomplete proof
- 4 pts Incorrect/Not optimal time complexity
- ✓ - 0 pts **Correct**

QUESTION 4

4 Q4 15 / 20

- 0 pts Correct

✓ - 5 pts **Incorrect stack operations (if stacks drawn) or incorrect edges traversed (if trees drawn)**

- 10 pts Vertices explored in incorrect order
- 10 pts Step-by-step not shown, but some reasoning given
- 15 pts Missing or fully incorrect steps
- 2 pts Vertices in final DFS tree not added via discovery
- 5 pts Missing or fully incorrect DFS tree

QUESTION 5

5 Q5 20 / 20

- 10 pts Incorrect Algorithm or did not attempt algorithm
- 3 pts No discussion of correctness
- 3 pts No discussion of runtime
- 20 pts Did not attempt
- 5 pts Approach attempted but insufficient.
- ✓ - 0 pts **Correct**
- 3 pts incorrect runtime analysis

1.

- Algorithm: // Assuming a root node is given and adj. list representation
 - Initialize an empty tree T
 - Initialize a list L and iterator variable i as 0
 - Take the root node, mark it as discovered, and add it to $L[i]$
 - Add the root node to T
 - While $L[i]$ is not empty:
 - Initialize $L[i + 1]$
 - For all nodes in $L[i]$:
 - Search each edge (x, y)
 - If y is undiscovered:
 - Set y as discovered
 - Add y to $L[i + 1]$
 - Add y to T
 - Add the edge (x, y) to T
 - Else:
 - Ignore y
 - Increment i by 1
- Proof: // Assuming proof that levels = distance in BFS tree
 - Assuming our BFS tree contains a node whose level doesn't reflect its shortest distance, there must be some node x on level i whose shortest distance is j , such that $j < i$
 - For this to occur there must be some path in the original graph that reaches x in j edges
 - Due to how BFS searches all neighbors of each node, we know that for each step k in the "shortest path", the node at that step will be found at level k or earlier in the BFS tree
 - As a result, we know that the step j where x implies that x will be found at level j or earlier in the BFS tree
 - This contradicts the assumption that $j < i$, as this result shows that $j \geq i$
- Time Complexity:
 - For each node in the given graph, the algorithm will analyze each of its neighbors
 - Since we are analyzing each node, we automatically have an $O(n)$ runtime at minimum
 - In the worst case, each node has to search $n - 1$ nodes, but that's overcounting, so instead we can say that, throughout the course of the algorithm's execution, we explore every edge once, adding $O(e)$ to our runtime
 - Our overall runtime for BFS is $O(n + e)$, where n is the number of nodes and e is the number of edges

1 Q1 18 / 20

- 0 pts Correct
- ✓ - 2 pts **BFS may not find all nodes**
 - 2 pts No need to start BFS from a node without incoming edge in DAG
 - 4 pts Proof of correctness is not complete or partially correct
 - 8 pts Proof of correctness is wrong/missing
 - 2 pts Time complexity analysis is partially correct
 - 4 pts Time complexity analysis is wrong/missing
 - 8 pts BFS search is wrong

2.

- Algorithm:
 - Initialize a set of lists s with 1 list, which contains the first interval
 - Set a counter variable i at 1
 - Delete the first interval
 - While there are intervals to check:
 - Take the start time of the next interval
 - For all lists in s :
 - If the end time of the last interval of the current list is less than or equal to the start time of the current interval:
 - Add the current interval to the list
 - Delete the current interval
 - If the current interval is not yet deleted:
 - Initialize a new element of s , $s[i + 1]$
 - Add the current interval to $s[i + 1]$
 - Increment i by 1
 - Return i
- Time Complexity:
 - This algorithm will take each interval and perform operations on it, resulting in an $O(n)$ runtime at least
 - The for loop in the algorithm will run through all existing lists to check if the interval can fit into any of them
 - In the worst case, this will take $O(n)$ runtime
 - The creation of a new list will take constant time
 - As a result, this algorithm has a worst case runtime complexity of $O(n^2)$, but it is likely to be less, unless the number of processors needed is similar to the number of total intervals
- Proof:
 - In order for this algorithm to break, one of 2 things must occur
 - We undercount:
 - For this to happen, an interval must have been assigned to a list where it cannot actually exist
 - Since we check the end times and start times each time we add an interval to the list, this error is simply impossible
 - We overcount:
 - For this to happen, the lists must have created a new list when a new list wasn't needed
 - This can only occur if the algorithm decided that an interval that could be added to a list wasn't
 - However, we iterate through each list and check start times and end times, therefore each relevant end time is checked
 - It is impossible for our algorithm to skip a valid insertion for our interval
 - Proof by contradiction

2 Q2 20 / 20

✓ - 0 pts Correct

- 10 pts incorrect algorithm
- 5 pts incorrect time complexity
- 5 pts incorrect proof
- 3 pts unclear algorithm statement
- 2 pts time complexity can be more precise
- 2 pts informal/incomplete proof

3.

- Algorithm:
 - Initialize an empty tree T
 - Initialize a list L and iterator variable i as 0
 - Take an arbitrary node, mark it as discovered, add it to $L[i]$, and color the vertex color 1
 - Set the color c to color 2
 - While $L[i]$ is not empty:
 - Initialize $L[i + 1]$
 - For all nodes in level i :
 - Search each edge (x, y)
 - If y is undiscovered:
 - Mark it as discovered
 - Add y to $L[i + 1]$
 - Else:
 - If y is not the same color as c :
 - Return false
 - Mark the edge (x, y) as discovered
 - Change the color c to the other color
 - Increment i by 1
 - Return the graph
- Proof:
 - The algorithm is essentially constructing a BFS tree, with some added operations
 - We know by the definition of a BFS tree that if we guarantee there are no odd cycles in the graph, we can 2 color the graph
 - This is because odd cycles cannot be bipartite, and are therefore impossible to 2 color
 - We also know this because in a BFS tree, we can simply alternate coloring each level of the tree
 - The algorithm attempts to construct a BFS tree, alternating color assignments per level
 - Base Case:
 - There is only one node
 - The algorithm colors it appropriately
 - Base case is correct behavior
 - Induction: when it discovers a new node, there are 2 possibilities:
 - The node hasn't been discovered yet, which means it will be on the next level
 - By the definitions we laid out above, this node can then be colored appropriately
 - The node has been discovered and colored
 - From here there are 2 possibilities:
 - The node is the right color

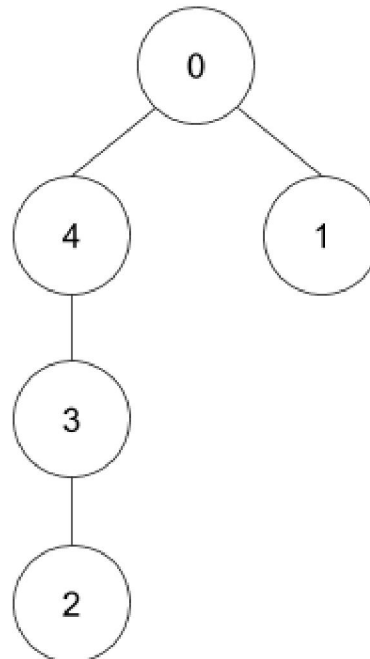
- This simply means we've discovered an even cycle, and since it's the right color, we're fine
 - The node is the wrong color
 - This means we've discovered an odd cycle, and since you can't 2 color an odd cycle, we return false from the algorithm
 - All of these possibilities exhibit the correct behavior, and therefore the algorithm will work for all general cases
- Time Complexity:
 - For each node in the given graph, the algorithm will analyze each of its neighbors
 - Since we are analyzing each node, we automatically have an $O(n)$ runtime at minimum
 - In the worst case, each node has to search $n - 1$ nodes, but that's overcounting, so instead we can say that, throughout the course of the algorithm's execution, we explore every edge once, adding $O(e)$ to our runtime
 - For each of these nodes, color assignments and checks are performed, which are simply constant time operations
 - Our overall runtime for 2 coloring is $O(n + e)$ like BFS, where n is the number of nodes and e is the number of edges

3 Q3 20 / 20

- **8 pts** Incorrect/No algorithm
- **4 pts** Unclear/partially correct algorithm
- **8 pts** Incorrect/No proof
- **4 pts** Unclear/informal/incomplete proof
- **4 pts** Incorrect/Not optimal time complexity
- ✓ - **0 pts** Correct

4.

- Steps:
 - Stack S is initialized
 - 0 pushed onto stack, $S = \{0\}$
 - 0 discovers 4 and pushes it onto stack, $S = \{4, 0\}$
 - 4 discovers 3 and pushes it onto stack, $S = \{3, 4, 0\}$
 - 3 discovers 2 and pushes it onto stack, $S = \{2, 3, 4, 0\}$
 - 2 has no more children to discover, 2 is popped from stack, $S = \{3, 4, 0\}$
 - 3 has no more children to discover, 3 is popped from stack, $S = \{4, 0\}$
 - 4 has no more children to discover, 4 is popped from stack, $S = \{0\}$
 - 0 attempts to discover 3, 3 has already been discovered, $S = \{0\}$
 - 0 discovers 1 and pushes it onto stack, $S = \{1, 0\}$
 - 1 attempts to discover 2, 2 has already been discovered, $S = \{1, 0\}$
 - 1 attempts to discover 3, 3 has already been discovered, $S = \{1, 0\}$
 - 1 has no more children to discover, 1 is popped from stack, $S = \{0\}$
 - 0 has no more children to discover, 0 is popped from stack, $S = \{\}$
 - Algorithm terminates
- Final tree:



4 Q4 15 / 20

- 0 pts Correct
- ✓ - 5 pts Incorrect stack operations (if stacks drawn) or incorrect edges traversed (if trees drawn)
- 10 pts Vertices explored in incorrect order
- 10 pts Step-by-step not shown, but some reasoning given
- 15 pts Missing or fully incorrect steps
- 2 pts Vertices in final DFS tree not added via discovery
- 5 pts Missing or fully incorrect DFS tree

5.

- Algorithm:
 - Initialize a max and min variable to the first element of L
 - Initialize an iterator to i
 - While $i + 1 < \text{size of } L$:
 - If $L[i] > L[i + 1]$:
 - Set variable local max to $L[i]$
 - Set variable local min to $L[i + 1]$
 - Else:
 - Set variable local min to $L[i]$
 - Set variable local max to $L[i + 1]$
 - Set max to maximum of local max and current value
 - Set min to minimum of local min and current value
 - Increment i by 2
 - If i doesn't equal the size of L:
 - Set max to maximum of $L[i]$ and current value
 - Set min to minimum of $L[i]$ and current value
 - Return max and min
- Proof:
 - Base Case:
 - An array of size 1:
 - The 1 element is both the maximum and minimum
 - The first element is assigned as both max and min
 - The correct answer is returned
 - Base case correct
 - Inductive Step:
 - Assume the algorithm works for an array of size n
 - Prove it works for an array of size $n + 1$
 - 2 cases:
 - $n + 1$ is even:
 - This means that n was an odd value
 - Since the first value is handled separately, the loop handles an odd number of values
 - Assuming the first n values are handled correctly, the $n + 1$ st value is handled by the if statement outside the loop, which is a brute force comparison to the global max and min
 - If the $n + 1$ st value should be a max or min, it will be assigned as such
 - $n + 1$ is odd:
 - This means that n was an even value
 - Since the first value is handled separately, the loop handles an even number of values

- For each pair of values, we perform a comparison to determine a local maximum and minimum
 - It is impossible for a local maximum to be a global minimum and vice versa
 - As a result, only the local maximum needs to be compared to the global maximum, and the same applies for the minimums
 - At this point, we effectively have a brute force comparison to the global max and min
 - If the $n + 1$ st value should be a max or min, it will be assigned as such
- Proof by induction
- Time Complexity:
 - For each 2 elements, a simple approach would make 4 comparisons: both elements would be compared to the global maximum and minimum
 - By first comparing the 2 elements once, we know one is greater than the other, invalidating one for maximum consideration and one from minimum consideration
 - This allows us to gain the information of 2 comparisons in 1 comparison
 - We then follow by performing a single comparison to the global max and a single comparison to the global min
 - As a result, for every 2 elements, our method uses 3 comparisons, resulting in approximately $3n/2$ total comparisons

5 Q5 20 / 20

- **10 pts** Incorrect Algorithm or did not attempt algorithm
- **3 pts** No discussion of correctness
- **3 pts** No discussion of runtime
- **20 pts** Did not attempt
- **5 pts** Approach attempted but insufficient.
- ✓ - **0 pts** **Correct**
- **3 pts** incorrect runtime analysis