CS 180, Fall 2019
Problem Set 2
Due October 16, 2019

Tian Ye
UID: 704931660

October 15, 2019

Exercise 3 Page 107

We will repurpose the topological sort algorithm, with the one difference of removing nodes with a degree of 0. If there are no nodes with a degree of 0 left in the graph after the algorithm finishes running, but there are still nodes, then a cycle exists. This holds true as in a directed acyclic graph, there exists at least 1 node with a degree of 0; this holds true as you continually remove nodes with degrees of 0 from the graph, until there are no nodes left.

- Create an empty list to later store nodes into, and a queue that holds all nodes with no incoming edges
- While the queue is not empty
- Remove node u from queue and append it to the list
- For each node v with an edge (u, v), delete edge (u, v) from the graph
- If v has no incoming edges, insert v into queue
- Once the above finishes running, if the graph still contains any nodes, it contains a cycle
- Otherwise, the list contains the DAG topological sorted order list

Proof. The algorithm runs in $O(m + n)$ time complexity as it is a small variation of the topological sort algorithm, the only difference being that once the sort runs, it checks the original graph for any remaining nodes, adding a coefficient to the n term. □

Exercise 4 Page 107

We will construct an undirected graph using butterflies as nodes and the judgements as the edges; arbitrary judgements have no edges. Using BFS, we will traverse the entire graph and label the butterflies. Once we finish labeling the entire graph, we will visit every edge and confirm the prediction. To prove the vailidity of this approach, we will use a proof by contradiction. Suppose the algorithm fails to detect an inconsistency between two nodes. We know this cannot be the case as the algorithm will first iterate through all elements of the graph during its initial run, labeling all nodes. When it checks through all m judgements in its second iteration, the algorithm will by definition make the comparison and detect the inconsistency. Hence, the algorithm will not fail to detect the inconsistency.

- Create an empty queue to store nodes into
- Choose an arbitrary starting node, give it the label A, and mark the node as visited
- Visit all incident unvisited nodes, marking them as A if the judgement is the same and B otherwise. Add these nodes to the queue and mark them as visited.
- While the queue is not empty, dequeue a node and repeat the previous step using the dequeued node
- Once the above finishes running, check every edge in the graph for consistency between the two nodes
- This check will determine whether all judgements are consistent

Proof. The algorithm runs in $O(m + n)$ time complexity as first it runs a BFS, which has $O(m+n)$ time complexity. When it validates all judgements, it will iterate through all m judgements. Hence, it will iterate through all n nodes once and all m judgements twice, resulting in an $O(m + n)$. □

Exercise 9 Page 110

In order for a path to be "unbreakable", there must be at least 2 nodes in every layer between s and t. Suppose via proof by contradiction that every level between s and t has at least 2 nodes. Since the distance between s and t is strictly greater than $n/2$, there will be at least $n/2$ levels between s and t. From this, we can say that there will be at minimum n nodes between s and t, which is impossible since there are n nodes total. Hence, there must be at least 1 level with only 1 node. We will be using a variant of a BFS to search for the layer with a size of 1

- Create an array of length n that marks the discovered nodes, setting it to false for all nodes except node s
- Set layer counter i to equal 0, and initialize a list $L[0]$ to consist of only s
- While $L[i]$ is not empty, create a new empty list $L[i+1]$
  - For each node within $L[i]$, add any undiscovered incident nodes to $L[i+1]$ and mark the nodes as discovered
- If the size of $L[i+1]$ is equal to 1, return the node v within $L[i+1]$
- Increment i

Proof. Since this is a BFS with almost no modifications other than checking the size of the layer, the algorithm will run with a time complexity of $O(m+n)$, as that is the time complexity of a BFS. □

Exercise 11 Page 111

We will construct a graph with the nodes being computers, and an adjacency list that is constructed of a list of tuples containing two elements: the time of communication and the computers communicated with. We will be using a variant of DFS to search for a path with strictly increasing timestamps between computer C_a and C_b, as that must hold true for the infection to occur. By updating the minimum time x with each iteration of the DFS, we ensure that we do not iterate the DFS with a node that is not yet infected. Assume for the purposes of a proof by contradiction that there exists some computer C_c that is infected and communicated with computer C_b between time x and y. It follows that if C_c was infected, there was some computer that previously infected it, C_d, earlier in time. We can follow this chain all the way to the original infected computer, C_a, being the path we are searching for. However, this is the path the algorithm that would have been found with a DFS as it followed infected node to infected node. Hence, this situation is impossible.

- Start with node C_a
- If the node u is already visited, return false
- Set the u as visited
- For each tuple (v, t) within the adjacency list $A[n]$
  - If t is between x and y
    - If v is C_b, return true
    - Else, repeat at step 2, replacing u with v and x with t
- Return false

Proof. Since this is a DFS that ignores certain edges in its implementation, we know the time complexity is at most $O(m + n)$, the time complexity of a DFS. □

Exercise 12 Page 112

We will initialize a graph with $2n$ nodes, two for each person P. If there is an overlap in lifespan, there will be a directed edge between the birth node of P_1 and the death node of P_2, and one between the birth node of P_2 and the death node of P_1. Otherwise, there will be a directed edge between the birth node of P_1 and the death node of P_1, as well as a directed edge between the death node of P_1 and the birth node of P_2. If a cycle exists in the graph, the data is inconsistent, as the cycle indicates that there is no internally consistent ordering of the facts that were presented.

- Create graph G with $2n$ nodes, 2 for each person P
- For each person P, add a directed edge from the birth of P to the death of P, incrementing the incoming edge counter of the death of P
- For each fact (P_1, P_2)
  - If P_1 died before P_2
    - Add a directed edge between the death node of P_1 and the birth node of P_2
    - Increment P_2's birth node incoming edge counter by 1
  - If P_1 died after P_2
    - Add a directed edge between the birth node of P_1 and the death node of P_2, and one between the birth node of P_2 and the death node of P_1
    - Increment P_1's and P_2's death node incoming edge counter by 1
- Create an empty queue Q and a counter i to 0
- For each node P, if the incoming edge counter is 0, add P to L
- While Q is not empty
  - Dequeue node P and remove it from the graph G
  - Increment i
  - For each outgoing edge from P to P'
    - Decrement P''s incoming edge counter by 1
    - If P''s incoming edge counter is 0, add it to Q
- If i is equal to $2n$, return true
- Else, return false

Exercise 6a

Given the array is sorted, we can perform a binary search to look for the missing element in the array.

- Let A be the array of n numbers from 1 to $n+1$, $l = 0$, $r = n$
- While $l < r$
  - Let $m = \frac{l+r}{2}$
  - If $r - l == 1$ and $A[r] - A[l] == 2$, return $A[l] + 1$
  - Else if $A[m] - m$!= $A[l] - l$, $r = m$
  - Else $l = m$

Proof. The algorithm is a binary search; hence the worst case time complexity is $O(\log n)$. □

Exercise 6b

Given the array is not sorted, we can put the elements into a hash set and search for the missing element.

- Let A be the array of n numbers from 1 to $n+1$, and initialize a hash set S, inserting each element a within A into S
- Initialize a counter i
- While $i \leq n + 1$, if i is not an element of S, return i. Otherwise, increment i

Proof. Populating the hash table requires a time complexity of $O(n)$. Accessing an element has a time complexity of $O(1)$; however, since we are searching through n elements for the missing element, the worst case time complexity to find the missing element is $O(n)$, resulting in an overall time complexity of $O(n)$. □