Preyasi Gaur Disc 1A

Time: 8:00AM - 9:50 AM

TA: Vincent Li

Computer Science 180

Homework 2

**Question 1**

* Algorithm:
  + Make a hashmap of all the n vertices in the directed graph, which will help us to keep track of the indegree of each vertex.
  + Next, we will initialize the hashmap to loop through all the edges of the DAG, and for every edge that we encounter, we can increase the counter for indegree by 1 for the specific vertex.
  + After traversing through all the vertices n, we are going to check every vertex that has a 0 value associated to its number of indegree edges and store these vertices in a set.
  + Now, in the set we will run a while loop. We will check that while the set of 0 indegree vertices is not none, we will pick a vertex from the set, remove it from the set, and place it next in the topological ordering
  + For this arbitrarily picked vertex from the set, we will also reduce the indegree for all the edges starting from this vertex and being indegree to another vertex. If there becomes a case, where while decrementing the number of indegree edges in this process, results in an vertex with 0 indegree, we will add to our set.
  + Now, if by this stage all the vertices in the DAG have been put in topological order, we will return the ordering.
  + If we reach a stage where there are no vertices with indegree 0, and we still have vertices that have to put in the topological ordering then:
    - We pick a vertex with a positive indegree, and locate an edge that leads to v, then trace the edge’s origin. We will keep doing the same until we reach a node that we have previously visited.
    - Now, since the graph is not a DAG, we return this cycle.
* Proof:
  + We know that if G is a DAG then it must have a topological ordering (Theorem 3.20 from the Book). And thus, if G does not have a topological ordering then it is not a DAG and thus must have a cycle.
  + We also know that if G has a topological ordering, then G is a DAG (Theorem 3.18 from the Book). Proof:
    - Suppose, by way of contradiction, that G has a topological ordering v1, v2,..., vn, and also has a cycle C.
    - Let vi be the lowest-indexed node on C, and let vj be the node on C just before vi—thus (vj, vi) is an edge.
    - But by our choice of i, we have j > i, which contradicts the assumption that v1, v2,..., vn was a topological ordering.
* Time Complexity Analysis:
  + Runs in O(V+E)
  + Proof: Since the longest cycle in a directed graph that is not a DAG requires visiting every node once, the worst case runtime for identifying a cycle in a directed graph that is not a DAG is O(V)

**Question 2**

* Algorithm:
  + Set up a graph such that:
    - Nodes: Butterflies caught
    - Edges: judgment of similarity or difference
  + Now we pick a vertex, and do Breadth First Search from this vertex through different edges and connections
  + Once the judgements have been made, we go ahead and connect the vertices with the same judgements from the leaves of the tree on the same level, and continue to run this until all the nodes are exhausted.
  + We repeat this process until all the edges have been processed to check for the possibility that G is not fully connected.
  + Now that we have constructed our tree, we are going to assign all the nodes on the tree a label based on the level it is on. We will have to kinds of labels so to speak: “same” and “diff”
    - For all the “same” labels: All of the judgements are inconsistent if they require connecting nodes from levels with different parities in the "same" judgment and if a vertex was left unlabeled, we can give it the same label as the other vertex in the "same" judgment that corresponds to it.
    - For all the “diff” labels: Ifthere occurs a “different” judgment such that it connects nodes from the same levels, then the judgments are inconsistent.
  + Except for the above mentioned, all judgments are consistent.
* Proof:
  + The set of judgements is inconsistent iff there is a cycle that has an odd number of “diff” judgements. We will go ahead and prove this.
  + Assumption: algorithm labels a set of inconsistent judgements as consistent
    - Proof: This means that the algorithm gave a legitimate bipartite partitioning of the nodes of a graph that is not bipartite, a contradiction, based on the parity of the labels it provided.
  + Assumption: algorithm classified a consistent set of judgements as inconsistent
    - Proof: This indicates that a cycle with an odd number of "diff" edges was found by the BFS tree of the "diff" judgements. This is contradictory as a bipartite graph cannot contain such a cycle.
* Time Complexity:
  + Runs in O(V+E)
  + As the algorithm that we have constructed uses primarily Breadth First Search, we know that it runs in O(V+E) time.
  + The worst case for this algorithm is when it goes through all the judgments twice, thus, making the run time O(E).

**Question 3**

* Algorithm:
  + Do Breadth First Search and create a BFS tree with the vertex P as the root. For all the levels in the tree, if at a certain level, there is only one vertex, we return that vertex.
* Proof:
  + Take the BFS tree of a graph having n vertices such that the distance between two vertices, say P and Q is more that
  + In this case, the number of levels of the BFS tree are greater than .
  + Assume there isn't a node v that blocks off all pathways from P and Q. Thus, must exist 2 unique path with length greater than that connect P and Q
  + But by the property of the BFS tree that all nodes only appear once, and it takes to make one path between P and Q, there are not enough nodes available to construct another unique path such that it is greater in length than .
  + Because of this contradiction, we can deduce that there must in all cases exist at least one node that is on a level alone, and that node with the unique must thus be the result, as the path from P to Q must pass through it.
* Time Complexity:
  + The algorithm simply uses BFS which we already know runs in O(V+E) time. For the worst case, we will have to search through every level which will take O(V) time. Which means that the full algorithm will run in O(V+E) time.

**Question 4**

* Algorithm:
  + Start by building a hashmap that maps each of the computers to a value that indicates if they are healthy or not.
  + We will disregard all the triples regarding communication before the given input time
  + Now for all the remaining triples:
    - We check if there is only one triple:
      * If the triple is the only one that is occurring at that time, we will check to see if one or both the computers are infected. If they are infected, we will mark the other as infected.
      * Now, if none or both of the computers are infected, do nothing
    - More than one triple:
      * Take all the triple, and we will create a graph with the computers as nodes, and the triple that records the communication of two computers as edges.
  + We will apply BFS to this created graph, and while exploring the triples at the same time with BFS, if any computer is part of a component connected to an infected machine, we will classify the whole component as infected.
  + After looping through all the triples, we go through them again and make a set of all infected computers.
  + After all the triples have been covered, we will use our hashmap to check if the computer is infected or not.
* Proof:
  + If a computer was infected before the algorithm returned:
    - There must be a series of triples which would have helped predict this, and since all triples are assessed in a chronological order based on their sorting, this is a contradiction.
  + If a computer was infected after the algorithm returned:
    - Then the algorithm might have at some point encountered a series of triples that made the specific computer infected. This is contradictory to the fact that the computer was infected at a later time.
* Time Complexity:
  + To create the hashmap of m computers: O(m)
  + BFS to process triples it encounters: O(n)
  + Entire algorithm run time: O(n+m)

**Question 5**

* Algorithm:
  + Creating the Graph: For every individual , make two nodes: for their birth and for their death, connected by a directed edge.
    - Make a directed edge from to for every type 1 fact where passed away before was born.
    - Make a directed edge from to for each type 2 fact where passed away before .
    - Make a directed edge from to and another from to for every fact of type 3 in which and were alive concurrently.
  + Consistency Check:
    - Sort the generated graph topologically.
    - The facts hold if a topological ordering is feasible. Assign dates of birth and death according to the topological order.
    - If a topological ordering is not possible (i.e., the graph contains a cycle), the facts are not consistent.
* Proof:
  + As we add new edges for type 2 facts, given that the facts are true, it should be possible to arrange the births and deaths (the graph's nodes) in a way that respects each directed edge, or temporal link.
  + A topological sort is possible iff the graph is DAG. The ability to sort the nodes topologically (assigning birth/death dates without contradiction) directly corresponds to the facts' consistency. If we are not able to topologically sort the values, then we have a cycle which is contradictory as in Point 3
  + A cycle in the directed graph indicates an irregularity in the temporal relationships among the people. This is due to the fact that a cycle would suggest a contradiction, such as someone existing both before and after another person, which is simply not possible
* Time Complexity:
  + To construct the graph, we have O(E) as we need to check facts to add each edge.
  + The complexity for creating nodes is O(V) since two nodes are created per person.
  + The sorting complexity for topological sort is O(V + E)
  + Thus the total complexity is O(V + E)

**Question 6**

* Algorithm:
  + Initialize an array min\_jumps of size N, where min\_jumps[i] represents the minimum number of jumps required to reach index i from index 0 and set min\_jumps[0] to 0, as no jumps are needed to reach the first index from itself.
  + Iterate through the array from index 1 to N-1, and for each index i:
    - Initialize min\_jumps[i] to a large value (e.g., infinity).
    - For each index j from 0 to i-1, check if arr[i] is equal to arr[j] + 2:
  + If the condition is met, calculate the number of jumps required from j to i.
  + Update min\_jumps[i] with the minimum of its current value and min\_jumps[j] + 1.
  + Check if a jump from the previous index (i-1) would require fewer jumps than the current min\_jumps[i]. If so, update min\_jumps[i] to min\_jumps[i-1] + 1.
  + The minimum number of jumps to reach the last index will be stored in min\_jumps[N-1].
* Proof of Correctness:
  + The algorithm maintains an array min\_jumps, where min\_jumps[i] represents the minimum number of jumps needed to reach index i. It initializes the first element to 0 since no jumps are needed to reach the first index from itself. The algorithm then iteratively computes min\_jumps for each index i. It considers two cases:
    - If there exists an index j such that arr[i] is equal to arr[j] + 2, it calculates the minimum number of jumps required to reach index i from index j and updates min\_jumps[i] accordingly.
    - It also considers the case of jumping from the previous index (i-1) to index i and checks whether it would require fewer jumps. If so, it updates min\_jumps[i] with the minimum of its current value and min\_jumps[i-1] + 1.
  + The algorithm iterates through the entire array, and min\_jumps[N-1] will contain the minimum number of jumps required to reach the last index.
* Time Complexity:
  + The algorithm uses a nested loop to calculate the minimum jumps for each index. The outer loop runs N times, and for each index i, the inner loop may iterate over a subset of the previous indices. Therefore, the time complexity is O(N^2), where N is the size of the input array.