Arthur Zhou

Dis 1A, 10am - 11:50am

Parshan

+ also initialize a vector v for the final topo sorted output

1) n nodes, m edges

GOAL: A topo sort if the graph is a DAG, a cycle if it isn't

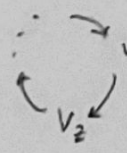
- Create a hashmap containing all n vertices, with the key being each node and value being the # of in-degrees into the particular node
  - Traverse each edge in the graph, update the in-degree value of the node the edge points to in the hashmap
- Traverse the hashmap again, put all vertices w/ indegree of 0 into a stack in any order
- Pop the top of the stack
  - Place it in vector v
  - For every edge that points from the vertex, decrement the in-degree of the vertex it points to
    - If any of the vertexes that the edge points to has its in-degree become 0 after this process, push it to the top of the stack
  - Repeat this popping process until ALL vertices are put into the topo sort vector
    - When done, return the vector
  - IF at any point there are no vertices with 0 in-degree AND the topo sort vector isn't full,
    - Pop the top of the stack
    - Choose an edge that points to the vertex, walk back that edge arriving to the previous vertex
    - Mark all edges and vertices walked through
    - Continue doing this until you reach a node that's already been visited
    - Return the cycle to prove the graph isn't a DAG

PROOF: We know the existence of a topo sort of a graph means the graph is a DAG, and vice versa

CLAIM: If graph G can be toposorted then G is a DAG

BWOC: Assume G has both a cycle and topo order.

Let's assume v_z is the lowest node in a cycle, and v_y is right before it.

Giving: 

But if (v_y, v_z) is an edge, but v_z leads to v_y, it contradicts $y > z$.

∴ If G has a toposort then it must be a DAG bc no cycles will exist!

CLAIM: If G is a DAG it must contain a toposort

We made an algorithm for this in class:

- Find a source
  - Output the source
  - Remove the source and all edges from it

repeat until we've outputted all nodes

The time complexity is $\boxed{O(m+n)}$

$T(n) = O(m+n) = 3n + 2m$

- Traversing all nodes: n
- Traversing all edges: m
- Traversing hashmap: n

Popping the stack and following the path of edges will at max. only visit each node and edge once, giving us an extra m and n

2) 2 species: A & B

2 subjects (i, j) can be labled "same" or "diff"

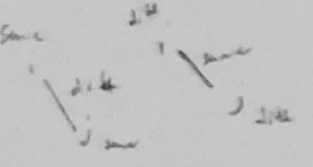
n specimens, m comparisons

GOAL: Is the data consistent?

Time complexity: $\boxed{O(m+n)}$ bc we perform BFS twice, one for vertices and one for edges.

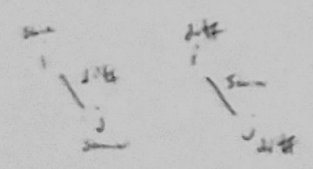
PROOF: 2 cases:

CASE 1: Algorithm labled a set of judgements that's inconsistent consistent



can't be possible since the 2nd BFS will assign a proper consistency judgement

CASE 2: Algorithm labels a set of judgements that's consistent inconsistent

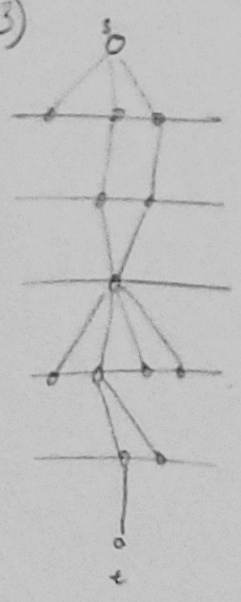


2nd BFS with edges will make sure this doesn't happen

- Create an undirected graph w/ each specimen as a node and each "same"/"diff" as an undirected edge
- Start from the vertex (arbitrary node), then perform BFS fr it
- Start BFS: queue: {vertex v}, mark v as "visited"
  - For each of vertex v's neighbors, add them to the queue and mark all of them as "visited"
    - If the judgement from v to its neighbor is same, labe v as the same as its neighbor
    - Otherwise, mark v and its neighbor as different
  - Remove the top element of queue and continue
- Iterate through all the edges too
  - For each edge (i, j),
    - If judgement is "same" AND both i and j are labled differen we have an inconsistency
    - If judgement is "diff" AND both i and j are labled same, we also have an inconsistency

$\therefore$ we will know if we have an inconsistency

3)



Perform BFS with node s as the starting node

If at ANY level there is only 1 node, remember it!

Time Complexity: If we are using BFS the time complexity is $\boxed{O(v+e)}$ where v is # of vertices and e is # of edges

PROOF: If the distance between nodes s and t is greater than $\frac{n}{2}$ it means that the # of levels the tree has must be over $\frac{n}{2}$.

- To say that a node that cuts all paths from s to t doesn't exist implies the existence of at least 2 FULLY UNIQUE paths of length $\frac{n}{2}$
  - But if it took over $\frac{n}{2}$ nodes to create the first path, there's not enough nodes to recreate another one that's size $\frac{n}{2}$ do, a CONTRADICTION!

$\therefore$ there's at least 1 node that will be on a level alone and will be the sole node that connects s & t

4) n computers, m triplets

C_1
C_2
⋮
C_n

- Create a hashmap with the key being the computer and value being a boolean marking if the computer is infected
  - Initially, the infected boolean for ALL computers should be false
- Ignore ALL triplets before time x and after time y
- Iterate through all the triplets
  - If there's only ONE triplet at time t, check if EXACTLY one computer is infected
    - If yes, set the infected boolean of the other computer do TRUE
    - Else, continue
  - If there's MULTIPLE computers at this time, create a graph with the nodes being each computer involved at the time and edges being 2 computers being connected via triplet
    - BFS the graph, explore it
      - If a SINGULAR computer is infected within the entire web, mark all of them as infected
- To check if computer C was infected, just search it in the hashmap

PROOF:

CASE 1: If a computer was infected earlier and the algorithm returned it as healthy

But all triples are sorted by time order, so this is a contradiction!

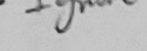
CASE 2: a computer was infected after time y yet our algorithm still returned that it got infected

Then compute C' must've infected C,

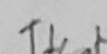
Time Complexity: $\boxed{O(m+n)}$, at worst we iterate through EVERY computer and triplet. Even BFS is O(m+n) too if we must do that

5) 3 possibilities.

P_i died before P_j born



P_i dies before P_j dies



P_i and P_j's lives overlap partially

- For each person P_i create nodes P_{bi} and P_{di} representing their birth & death dates
  - Create directed edge from P_{bi} to P_{di}
- If p_i dies before p_j was born, draw edge from P_{di} to P_{bj}
- If p_i dies before p_j dies OR if their lives aligned partially, draw 2 edges: one from P_{bi} to P_{dj} and P_{bj} to P_{di}
- Do a topo sort of the graph
  - If not possible, details are INCONSISTENT!
  - Else, details ARE consistent and assign dates in ascending order from left to right in the topo order

PROOF: An inconsistency must contain p_i was born before p_i, thus creating a cycle. Or if both p_i and p_j are alive at the same time denotes either p_i was born before p_j died or vice versa. Regardless, there must be an edge connecting the cycle when p_i and p_j were alive at the same time

But this is a contradiction bc topo sort ONLY works on DAGs! So the algorithm couldn't have possibly labled any cycles.

Time Complexity:

We make 1 or 2 edges per detail and 2V nodes where V is the # of people. So we have O(E+V) time complexity for processing people and edges.

Topo sort is O(e+v) too but O(e+v) + O(e+v) is still $\boxed{O(v+e)}$

6) • Iterate through the array, create nodes w/ each one storing its value and index
  ◦ Draw edges from each node to their neighboring indices
  ◦ Draw an edge from a node to another if the node's value is equal to the other node's value +2
• Perform BFS with node at index 0 as the starting node
  ◦ Return the level or the # of edges it took to reach the last node in the array index-wise

PROVE: Our graph represents all possible jumps from any element to another.

CLAIM: There's a faster route to reach the end, or that the last node is not equal to the distance from the starting node

↑ but this is a CONTRADICTION to BFS by itself!

Time complexity: Worst case scenario of BFS is if the graph is COMPLETELY CONNECTED resulting in an $O(N^2)$ time, though it's rare but we don't know the # of edges.