Joyce Chen Dis 1F 2:00PM Parshan Teimouri

CS180 Homework #2

1. **Exercise 3 Page 107**

Problem:

- Run a topological sort on a graph G that may or may not be a DAG.
- If no valid topological ordering exists, output any cycle in G. Conclude that G isn't a DAG.
- If a topological ordering exists, output a valid topological ordering. Conclude that G is a DAG.

Algorithm:

- Create a hashmap H that maps the nodes to their incoming degrees.
- If all nodes have incoming edges:
  - Conclude that G is not a DAG
  - Pick a node from the remaining nodes with incoming edges
  - Do a DFS on this node until we find a path that goes back to itself.
  - Output this cycle and return.
- Otherwise:
  - Find a node v with incoming degree of 0 and order it first
  - For each existing neighbor of v, decrement their incoming degrees by 1
  - Delete v from G + all its outgoing edges
  - Recursively compute a topological ordering of G − {v}
  - Append this order after v
- Return the topological ordering of G

Runtime Analysis:

- This algorithm produces a topological ordering if G is a DAG; otherwise, it establishes that G is not a DAG.
- Identifying a node v with no incoming edges and deleting it from G takes $O(1)$
- If a topological ordering exists, we would have to go through the neighbors of all nodes in G, which takes $O(2m)$ time, or $O(m)$.
- If a cycle exists, the final DFS to output the cycle takes $O(m+n)$ time.
- Total runtime would therefore be $O(m+n)$ time.

Algorithm Proof:

- Suppose the algorithm doesn't detect that G isn't a DAG:
  - If G is not a DAG, it would have to contain a cycle such that a topological ordering is impossible.
  - Case 1: G is a cycle
    - Right from the start, no nodes have an incoming degree of 0 because each node has an in-degree 1 and out-degree 1.
    - The algorithm will enter the first condition, which concludes that G is not a DAG.
    - ><
  - Case 2: G contains cycles
    - The only way for G to return an ordering is if the algorithm is able to delete an edge in the cycle(s).

        - To delete an edge in the cycle, that edge would need an in-degree of 0. However, all edges in a cycle have an in-degree of at least 1.
        - At one point, all edges have an in-degree of at least 1 (only cycles are left, and all other nodes are gone).
        - Algorithm concludes that G isn't a DAG due to the first condition.
        - ><

2. **Exercise 4 on Page 107**

Problem:

- Figure out whether *m* judgments are consistent or not. Consistency is defined as being able to label each specimen pair conforming to the judgments, where (A, B) = "different" and (A, A)/(B, B) = "same".
- In other words, if i and j are labeled different, they must have opposite labels (ie. i = A, j = B). If another specimen k is said to be different from i, it must have the label B, or else the judgments are inconsistent.

Algorithm:

- Create a graph where each node is a specimen, and each edge is a judgment for the 2 specimens it connects. Each edge should have a weight associated with it that specifies whether the 2 specimens are the "same" or "different."
- Check whether the graph is bipartite by running BFS on the graph.
- Create a hashmap that maps each node to its label, either A or B. The starting node gets assigned an arbitrary label.
- At each node V, check each of its neighboring nodes.
    - If neighbor N was visited before:
        - If edge (V, N) is "same," then V and N should have the same label. Otherwise, conclude inconsistent.
        - If edge (V, N) is "different," then V and N should have opposite labels. Otherwise, conclude inconsistent.
    - If the neighboring node N was NOT visited before:
        - If edge (V, N) is "same," assign N the same label as V.
        - If edge (V, N) is "different," assign N the opposite label from V.
- If we successfully reach the end of BFS and traverse all the nodes, this means that the *m* judgments are consistent.

Runtime Analysis:

- This is BFS but with an extra hashmap data structure to track the labels for each node + an extra check before adding neighboring nodes to the queue.
- Setting up the graph would require us to go through all *m* judgments, which takes O(m) time.
- Running BFS on the graph will take O(m+n) time, where m = # of edges (judgments) and n = # of nodes (specimen).
- Total time = O(m+n)

Algorithm Proof:

- Suppose that for specimens A and B, the algorithm doesn't detect the inconsistency in the judgments: "A and B are different," "A and B are the same"

- This means we traversed A and B successfully, and considered the edges (A, B) twice for both judgments.
- In the first judgment encountered where “A and B are different,” A is assigned the opposite label from B.
- In the second judgment encountered where “A and B are the same,” A is assigned the same label from B. However, because A was already visited, it would have gone through the first check.
- Since A and B were already labeled in the same category from the first judgment, the algorithm would have concluded an inconsistency.
- ><

3. **Exercise 9 on page 110**

Proof by contradiction:

- Suppose there are no nodes in G such that deleting the node would destroy all s-t paths.
- All s-t paths would not have overlapping nodes. That is, all paths have a distinct, disjoint set of nodes they visit.
- Because distance = # of nodes in path - 1 = at least n/2+1, we can infer that all paths from s to t must contain at least n/2+2 nodes.
- Out of n-2 nodes to choose from, we must choose n/2 nodes.
- We can now divide the problem into 2 cases:
  - There is only 1 path leading from s to t.
    - Because a path doesn’t contain cycles, deleting 1 node from the path would remove all paths from s to t.
  - There are >1 paths leading from s to t.
    - Each path will have at least n/2+2 nodes, including s and t.
    - Total # of nodes visited by P_1 and P_2 is at least 2*(n/2+2) = n+4.
    - Since n+4 > n (the total # of nodes in G), at least 1 node was visited in *both* P_1 and P_2.
    - P_1 and P_2 have overlapping nodes. Contradiction ><

Algorithm:

- Create a hashmap that maps each node to its level. The starting node should have a level of 0. Also create a variable L that tracks the current level.
- Run BFS on the graph:
  - Create a queue Q and push the starting node “s.”
  - Create a set V to track all the visited nodes.
  - While Q is not empty:
    - Pop the top of Q to get the current node “n”
    - Increment L (level += 1)
    - For each of n’s neighbors:
      - If the neighbor is not in V:
        - Add to Q.
        - Set its level to L.
        - Continue.
      - Otherwise, if the neighbor is in V (already visited):
        - Ignore and continue.

- Iterate through the hashmap to find the node v that is the singular occupant of some level.
- Return v.

Runtime Analysis:
- Running BFS on the graph takes O(m+n) time. The additional step to add a node-level pair to the hashmap takes constant O(1) time.
- The final iteration through the hashmap takes O(n) time, because we're going through all the nodes and checking their levels.
- Total time would therefore be O(m+n).

Algorithm Proof:
- Suppose the algorithm returned a vertex v that didn't destroy all s-t paths.
    - An s-t path would have nodes on all layers of the path from L_0 to L_n, where n = the distance.
    - Say vertex v lies on L_i. Then there must be more vertices on L_i such that the s-t path includes edges connecting the layers L_{i-1} and L_i, and edges connecting the layers L_i and L_{i+1}.
    - Otherwise, the layers L_{i-1} and L_{i+1} would be disconnected, and there would be no path from s to t.
    - However, in the algorithm we found that the node v was the singular occupant of the level it resides in, or L_i.
    - ><

4. **Exercise 11 on page 111**

Problem:
- Constraint:
    - Triples are presented in sorted order of time
    - Each pair of computers communicates at most once during the interval
- Variables:
    - *m* – total # of triples, *n* – total # of computers in the system
    - C_a – the computer that the virus was inserted into
    - *x* – the time the virus was inserted
    - C_b – the final computer
    - *y* – the final time

Algorithm:
- Find the first time a trace data point includes C_a and t_k is greater than or equal to x. Call this first point (C_i, C_j, t_x).
- Create a graph G by iterating the trace data starting from (C_i, C_j, t_x) *up to but not including* the point where t_k is first greater than y. Call the last point (C_i, C_j, t_y).
    - Each node represents a computer C_k.
    - Each edge is a connection between 2 computers.
    - Each edge will have an associated "weight" that corresponds to the time that C_i and C_j communicated, t_k.
- Run BFS on G starting from C_a to determine whether some computer C_b would be infected.
    - If C_b is reachable by BFS, then it'll be infected.

- Otherwise, C_b is part of a different component in the G that doesn't include C_a, meaning that it doesn't get infected.

Runtime Analysis:

- To create the graph, we'd need to go through each triple and choose the triples that have a time between x and y. This takes O(m) time, where m = # of triples.
- Worst case the triples given are all between the times x and y, so we'd have to run DFS on all the triples given, which takes O(m+n) time.
  - m = # of triples and n = # of computers.
  - In this problem, the # of edges corresponds to the # of triples because each triple specifies an edge between 2 computers.
- Total runtime = O(m+n).

Algorithm Proof:

- Suppose that C_b could be infected, but the algorithm returned that C_b is virus-free within the x-y time interval.
  - If C_b is counted virus-free, it wasn't reachable by the BFS traversal on G (starting from C_a).
  - Because we were given the triples in sorted order, each node will be added in chronological order, so each successive edge we encounter will have a greater time.
  - In other words, an infection would occur from node A → node B if a path exists between them.
  - Therefore, if C_b wasn't seen during BFS, then there is no path from C_a to C_b, indicating that an infection cannot occur.
  - ><
- Suppose that C_b is virus-free in the x-y time interval, but the algorithm returned that C_b could be infected.
  - If C_b could be infected, it will have to exist in the same connected graph component as C_a by definition of a path.
  - To be virus free, the algorithm states that C_b must be in a different graph component from C_b.
  - ><

5. Exercise 12 on page 112

Problem:

- Given a set of people and a set of facts, output a valid ordering of their birth and death dates if the facts are consistent. Otherwise, conclude inconsistent.
- Facts are of the following format:
  - Person A died before Person B.
  - Person A's lifespan overlapped with Person B's.
- Examples of inconsistencies:
  - Person A died before B, but its lifespan overlapped with B.
  - Person A died before B, and B died before C, but C died before A.

Algorithm:

- For each person, create a birth (b_i) node and death (d_i) node (total 2n nodes) for the birth date and death date, respectively.

    - Have each node track its corresponding person via some internal variable
    - Create an edge pointing from the birth node to death node: ($b_i \rightarrow d_i$)
- Iterate through the facts to create a directed graph G:
    - Say Person A has edge ($b_a \rightarrow d_a$) and Person B has edge ($b_b \rightarrow d_b$)
    - If Person A died before B:
        - Create an edge pointing from d_a to b_b, indicating that B comes after A in the timeline: ($d_a \rightarrow b_b$)
    - If Person A overlaps with B:
        - Create an edge pointing from b_a to d_b and b_b to d_a such that Person A is born before B's death and Person B is born before A's death: ($b_a \rightarrow d_b$), (b_b, d_a)
- Run a topological sorting algorithm on G:
    - If there is a cycle in G:
        - There is some inconsistency with the facts given
    - If we are able to produce a valid topological ordering:
        - Return this ordering of birth and death nodes

<u>Runtime Analysis:</u>

- Creating the directed graph takes O(M+N) time
    - First iterating through all the people and adding directed edges between their birth and death nodes – O(N), where N = # of people.
    - Then iterating through all the facts to add edges between these birth and death nodes – O(M), where M = # of facts.
- Running topological sort on the graph to obtain the birth and death dates takes O(M+N) time.
- Total runtime is therefore O(M+N).

<u>Algorithm Proof:</u>

- Suppose the algorithm returns an ordering where Person A died before Person B, but the fact says otherwise.
    - If Person A died before B, then A's birth and death nodes would have to both come before B's birth node.
    - The topological ordering must be: A birth, A death, B birth, B death
    - According to the algorithm, we establish that if Person A dies before B, we create a directed edge pointing from A's death to B's birth node.
    - Then, when we run the topological ordering, B's birth node is guaranteed to come after A's death node.
    - Because there is a directed edge from A's birth node to A's death node already, and B's birth node to B's death node, there is no possibility that A's birth node will come after B's birth node, or B's death node will come before A's death node.
    - Therefore, the fact must have been "A died before B." ><
- Suppose the algorithm returns an ordering where Person A and Person B overlap in their lifespans, but the fact says otherwise.
    - If Person A overlapped with B, then one of the following orderings should exist:
        - A birth, B birth, A death, B death
        - A birth, B birth, B death, A death

        - B birth, A birth, B death, A death
        - B birth, A birth, A death, B death
    - In other words, A's birth must come before B's death, and B's birth must come before A's death.
    - Our algorithm establishes this by adding a directed edge from A's birth node to B's death node and B's birth node to A's death node.
    - The original fact must have been "A overlapped with B."

6. **Given an array arr[] of size N, find the minimum number of jumps to reach the last index of the array starting from index 0. In one jump you can move from current index i to index j, if arr[i] = arr[j] and i != j or you can jump to (i + 1) or (i – 1).**

   Problem:
   - We can use the idea that if we do BFS on a graph, the layer that any vertex is on would be equivalent to its minimum distance from the source vertex.
   - Therefore, if we arrange the array into a graph structure and perform BFS starting from the element at arr[0], the layer that element arr[len(arr)] is at would be "the minimum number of jumps to reach the last index of the array starting from index 0."
   - To arrange the array into a graph, we observe that an edge would represent a jumping path between 2 elements in the array
     - If arr[i] were a vertex and arr[j] were a different vertex, they will be connected by an edge if arr[i] = arr[j].
     - An edge would also exist between elements arr[i] and arr[i+1], as well as arr[i] and arr[i-1].

   Algorithm:
   - Create a hashmap H that maps a node (value) to a list of indices it can jump to.
   - Traverse arr from index i = 0 → N-1 to fill H.
     - Append i to the list mapped to by the value arr[i] in H.
   - (Do BFS on arr, starting from the first element)
   - Create a queue and push the index 0 (starting node).
   - Create a set V that tracks all visited nodes.
   - Create an array L to track the levels of each node. Initialize all slots to 1 (level 1).
   - While queue is not empty:
     - Pop queue to obtain the index i.
     - If i-1 > 0 and i-1 is not in V:
       - Add i-1 to the queue and the set V (visited).
       - Set L[i-1] to be L[i] + 1 (incrementing the level)
     - if i+1 < N-1 and i+1 is not in V:
       - Add i+1 to the queue and the set V (visited).
       - Set L[i+1] to be L[i] + 1 (incrementing the level)
     - Iterate through arr[i]'s list of indices (using H) from j = 0 → k:
       - If the j is equal to N-1:
         - We've reached the end!
         - Return L[i] + 1 (level # of parent+1 = min # of jumps).
       - If j is not in V:
         - Add j to the queue and the set V (visited).
         - Set L[j] to be L[i] + 1.

Runtime Analysis:

- Traversing the array to create the hashmap takes O(N) time.
- Running BFS on the array takes O(M+N) time.
- Creating the auxiliary array to store the levels of each node takes O(N) time.
- Total runtime = O(M+N).

Algorithm Proof:

- Suppose that the algorithm doesn't give the minimum number of jumps possible.
    - This means that we returned a level number greater than the minimum number of jumps. We reached node len(arr)-1 later than expected.
    - This means that there is no direct path in G for our solution (ie. if solution was 0-4-5-9 then there's no path in G with the path 0-4-5-9) because BFS will always give the distance between source and end vertex (ie. 0 and 9).
    - This is only possible if G was not connected properly (not all the valid jumps for each index were mapped).
    - However, in the algorithm we are adding the indices that each node can jump to (indices that have the same value as the node, index + 1, index - 1). These are all the valid jumping indices for a node as established by the problem.
    - ><