Homework 2

Tejas Kamtam

305749402

Discussion 1B

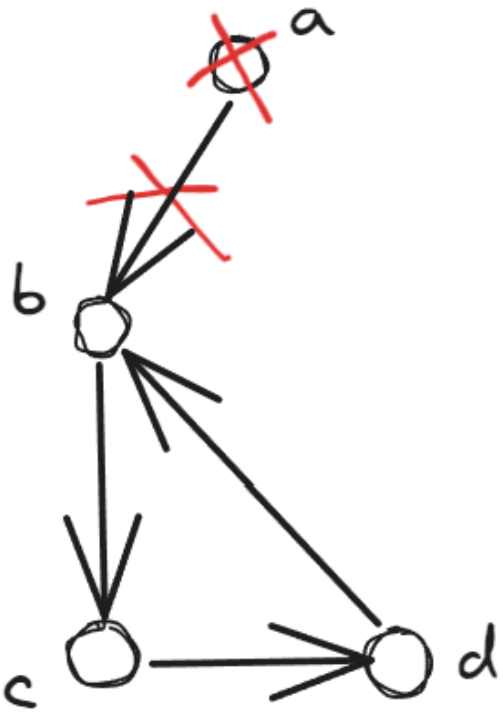
Friday, 10 am

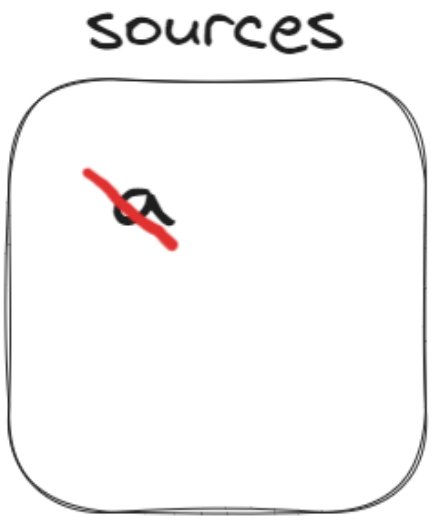
TA: Parshan

Problem 1

- Exercise 3, Pg. 107

Reasoning
We can check for cycles by simply modifying the return value of the topo sort we learned in class (Kahn's algo) using a constant time check that runs only once per sort (regardless of the number of iterations).
All we have to do is observe that the topo sort we learned in class has a `while` loop while the queue/list/pool of sources is not empty. Now, suppose we do have a cycle. Then, **no node s in the cycle would ever be added to the list of sources** because there will always be some other node pointing to s making it a sink. For example, consider that in the following graph, the algorithm's loop will end right after appending `a` and decrementing the in-degree of `b` to the output list/linked-list:





So, the loop exits after adding `a` to the output list, because there are no nodes left with in-degree of 0 (sources).
Thus, all that we have to do is, once the output list `ans` is returned, check if the size of `ans` is `<` `n`, the number of vertices . If so, we can say that there exists a cycle, if not, then we just return the output list.
Because all we do is add a single constant time check to the end of the algorithm (once, not per iteration), the time complexity is unchanged and still linear in $O(n + e)$ as the original topo sort.

Algorithm

- Given a list of n nodes and e edges, find the in-degree of each node
- Get a list of all the sources in the graph
- From the sources list, arbitrarily pop one node a (access then delete) from the list, add it to the output visited array
- Check all of the children of a and decrement their in-degree by 1 and if their in-degree is 0 => add them to the "source list"
- do this until the source list is empty
- at the very end, compare the size of the output array to n (the number of nodes): if it is less than n => there exists a cycle, if not => return the output

array as the sorted list of nodes

Time Complexity Analysis

- Constructing the directed graph from n nodes and e edges is $O(n+e)$
- Running the Topological Sort is $O(n+e)$
- Modifying any necessary lists is $O(n)$
- Total time complexity is linear in $O(n+e)$

Problem 2

- Exercise 4, Pg. 107

Reasoning

We can model each of the n butterflies as a node in a graph and draw an undirected edge to another butterfly IF the scientists judge that they are **different** (in the worst case this could be all m judgements) (for the task, we don't care about "same" judgements as they are implicit in determining consistency). If their judgements are indeed consistent, the resulting graph should be **bipartite** such that all nodes on one "side" of the graph are either group A or B and the other "side" is the other group. Now, we want an algorithm that determines if such a graph is bipartite or "2-colorable" which we have learned in class must ensure that there are no "odd-cycles" by the odd-cycle lemma we proved in class using a modified BFS.

Algorithm

- given a list of n nodes of butterflies and e edges of "different" judgements, arbitrarily select one node a and add it to a queue and assign it a classification A and add it to a visited list
- pop from the queue and check all the children/adjacencies and assign these nodes to class B and add them to the queue and visited list
- repeat until the queue is empty such that when adding a node to the visited list at position $i+1$, assign the node to class A if $i+1$ is odd, else assign to class B

- if at any time in the loop, we try to **re-assign** a node to a class that is not consistent with its previously assigned class, we can determine that the scientists' judgements are inconsistent, otherwise the judgement graph is consistent

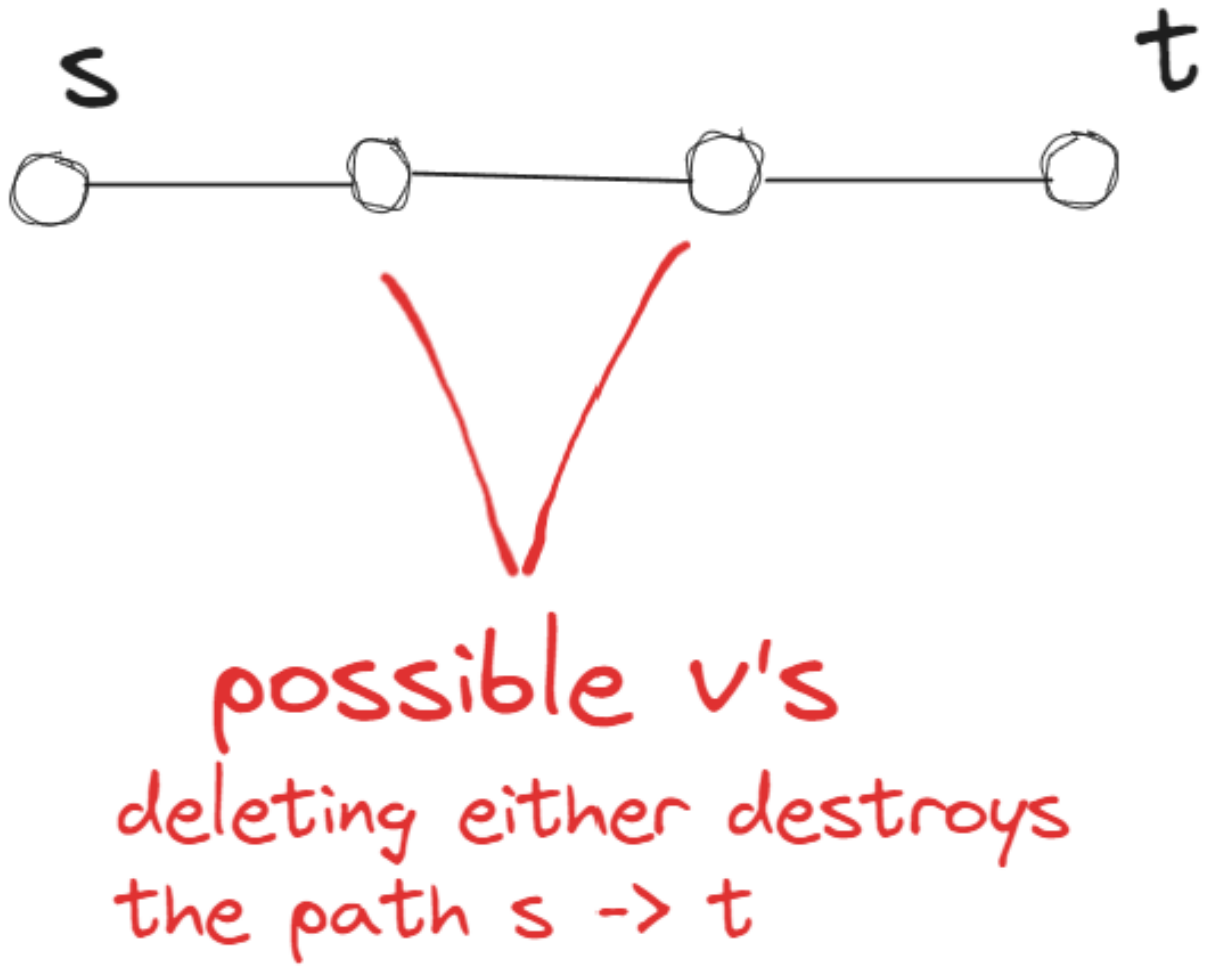
Time Complexity Analysis

- Constructing the graph is $O(n+e)$
- Running the modified BFS is $O(n+e)$
- Modifying any necessary lists/queues is $O(n)$
- Total time complexity is linear in $O(n+e)$

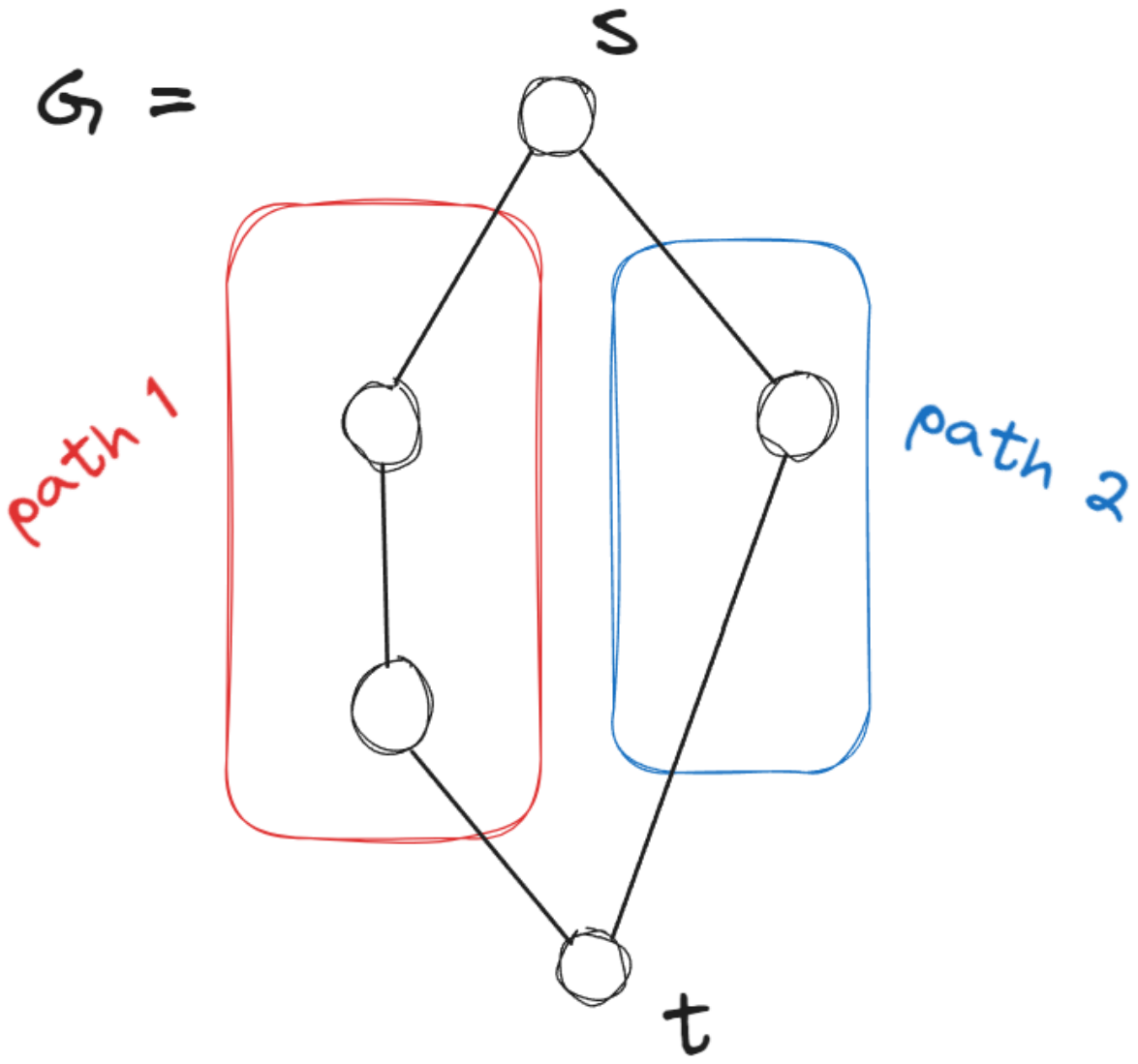
Problem 3

- Exercise 9, Pg. 110

Proof: BWOC
We can consider the situation when there is no such node v such that all possible paths $s \to t$ intersect. First considering the situation with only 1 path $s \to t$: there must be some node v in $s \to t$ to actually be a valid path, such that removing v destroys all possible paths (in this case, only 1), like so:



Now, if we consider the case with multiple paths $s \to t$, then if there is no such node v, there must be **unique paths using completely unique nodes** in each path between $s \to t$. WLOG, something like this:



However, in this scenario, no such unique paths are possible under the constraint that the length of path $s \to t$ must be strictly $> \frac{n}{2}$ (path 1: length 3, path 2: length 2, $n = 5 \implies \frac{n}{2} = 2.5$). We can make an inductive assumption that this impossibility will hold for all possible variations of unique paths as there is no possible path that increases the length without increasing the number of nodes proportionally.
So, the only possible way to satisfy the constraint must be using some intermediary node v such that deleting v destroys the paths $s \to t$ WHEN the length of the path must be $> \frac{n}{2}$.
We can find such a node v using a BFS starting from both ends of the path s and t simultaneously.

Algorithm

- begin BFS from s and begin BFS from t simultaneously with independent queues and visited arrays
- at each iteration of the BFS, check the visited array to see if there are any shared nodes
- once there is a shared node, return this node as node v and exit the loop

Time Complexity Analysis
We've already proven that there must be a shared node v, and because this program runs simultaneously on both ends of the path at each iteration, the worst case time complexity is going through the entire graph

- Running BFS is $O(n+e)$
- Running BFS on two nodes at the same time is a constant multiple of the run time of a single BFS
- Total time complexity is $O(n+e)$

Problem 4

- Exercise 11, Pg. 111

Reasoning
We can consider using a graph to model the networking events. We can model each computer as a node and include edges (bidirectional/undirected) whenever 2 computers (nodes) communicate. But, to track time, we can give each edge an associated weight that represents the time of the communication event. We can then run a slightly modified BFS on this graph to find if node b has been infected by time y given a virus injected in node a at time x.

Algorithm

- given the list of all triples with at max n computers (nodes) and e communication events (edges), we can construct an undirected, weighted graph representing communication events and times between computers
- we begin with node a and run BFS, but only add children nodes if the edge between them has a weight (time) greater than the time of insertion x and

greater than the weight of the edge before it (which we can store along with each node: the incoming node and weight)
- we continue to run BFS until we observe node y OR until all further nodes (in the queue) only have incoming edges with a weight greater than y (the time we are checking by)
- if we find the node by time y return True, else return False

Time Complexity Analysis

- Constructing the graph is $O(n+e)$
- Running BFS is $O(n+e)$
- Modifying the queue is $O(n)$ or $O(e)$ depending on how densely connected the graph is since we are also storing edge weights
- Total time complexity is $O(n+e)$

Problem 5

- Exercise 12, Pg. 112

Reasoning

We can consider modeling this problem using a graph of birth and death dates instead of specific people. We make each person's birth (e.g., for Person A, A_b) AND death (A_d) dates nodes and draw the following **directed** edges: (1) An edge $A_b \to A_d$, (2) if $A_d < B_b$ then an edge $A_d \to B_b$, (3) if $A_d \geq B_b$ then an edge $B_b \to A_d$.

We can then determine that the graph (the peoples' given testaments) are consistent if and only if there are no cycles. From the spirit of Problem 1, we can use a Topological Sorting (proofs for cycle detection are in Problem 1).

Algorithm

- Given a list of birth and death dates for each person as n nodes and e edges based on the given rules above, find the in-degree of each node
- Find a list of all the sources in the graph
- Construct the directed graph

- Arbitrarily pop a source from the source list and add it to an output array (visited array)
- Decrement each of the node's children's in-degree by 1 and if their in-degree reaches 0 => append them to the source list and delete the parent node from the source list
- Continue doing this until the source list is empty
- At the end of the loop, if the size of the output array is less than the number of nodes, there must be a loop (proven in problem 1) => output that the graph is inconsistent
- If not, return the output array; this output array is all the birth and death dates of each person: assign arbitrary values to the nodes such that the output array is in non-decreasing order and return this as a possible set of birth and death dates of each person that adheres to the facts given by word-of-mouth
    - this list must ensure a consistent graph because there are no loops (As proven by the length of the output array being equal to the number of nodes)
    - so, based on the edge rules, there are only edges to nodes if they occur before the node they point to, so it is impossible for the array to not be a valid proposed set of dates (because the array is sorted non-decreasingly)

Time Complexity Analysis

- Finding the in-degree of each node is $O(e)$
- Finding a list of sources in $O(n)$
- Constructing the graph is $O(n+e)$
- Running the topo sort is $O(n+e)$
- Assigning proposed values to the births/deaths is $O(n)$
- Total time complexity is $O(n+e)$

Problem 6

> Given an array `arr[]` of size `N`, the task is to 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)`.
> **Note:** You can not jump outside of the array at any time.
> **Example:**
> Input: `arr = {100, -23, -23, 404, 100, 23, 23, 23, 3, 404}`
> Output: `3`
> Explanation: Valid jump indices are `0 → 4 → 3 → 9`

Reasoning

We can consider modeling this problem using a directed graph of indices. Taking a couple examples, WLOG, we can see that we want to find a path in the graph from the starting index to the ending index. Minimizing this path length is congruent to minimizing the "height/levels" of the path. BFS is the perfect algorithm for this problem.

Algorithm

- create a graph of N nodes and e edges (the max number of edges per node is $N-1$) using a hashmap by going through the array and storing each index as a key and adding all accessible indices to its keys (e.g., `map = {0: (1,4), 1:(0,2), 2:(1,3), ... , 9:(8)}`)
- we can now add the starting index, `a`, to the queue
- pop `a` from the queue and add it to a visited array, check its children and add them to the queue and increment the `min_steps` counter by 1
- check if the ending index is in the queue, if so, return the `min_steps`, otherwise repeat the previous steps until the queue is empty
- if we reach the end of the loop (i.e,. nothing has been returned), then return that there is no such valid jumps that will take us from index `0` to the last index.

Time Complexity Analysis

- Constructing the graph is $O(N+e)$
- Running the modified BFS is $O(N+e)$
- Modifying any necessary lists/queues is $O(N)$

- Total time complexity is $O(N + e)$