Preyasi Gaur Disc 1A

Time: 8:00AM - 9:50 AM

TA: Vincent Li

Computer Science 180

Homework 3

**Question 1**

We will prove this using contradiction.

* Suppose there is an edge (a, b) in G but it is not present in T. Then the following is possible:
  + Case 1: a and b are exactly one layer apart
    - We know the T is a Depth First Search Tree.
    - For the sake of the proof, assume that DFS picks to explore a before b.
    - That means a is the ancestor of b
    - But, a and b are one layer apart
    - This means that a is the parent of b and e exists in T
  + Case 2: a and b are more than one layer apart in T
    - We know that T is a Breadth First Search Tree.
    - In BFS, when we explore a node, the other will be added to the subsequent layer
  + Case 3: a and b are siblings
    - We know the T is a Depth First Search Tree.
    - Assume, a is explored first. b should be in the subtree of a, as T is a DFS tree
    - Thus, a and b cant be siblings.
* As all the cases lead to a contradiction, T must be equal to G.

**Question 2**

We can adapt this problem to be a graph problem. Suppose

* There is an undirected graph G(V, E)
* Suppose we are given two node a and b
* We want to find the number of all the shortest paths from a to b

Algorithm:

* We run BFS starting at a.
* We modify BFS to store a counter of every node starting from 0, except for root a, for which the counter starts from 1.
* Store a different variable for each node, to keep track of the layer the nodes are in, and this variable we initialize to -1, except for the root, for which it is initialized to 0.
* This variable also gets updated as we keep discovering nodes.
* Now, suppose that we discover a node c, when as we are exploring the neighbors of another node d, then we can implement counter[c] = counter[c] + counter[d]. However, if c is a node in the upper layer of d or its sibling, we don’t update counter[c].
* Now, while running BFS, we check if we have found our target node, b. When a node at b’s layer is picked to explore around, the algorithm terminates.

Runtime:

* Increasing the counter and checking the layer: O(1)
* BFS: O(V+E)
* Overall runtime: O(V+E)

Proof:

* We will prove this mathematical induction.
* Hypothesis: The algorithm correctly counts the number of paths for all the nodes in layer 1 to k.
* Base Case: for k = 1 → number of all paths from vertex a to a is 1
* Induction step: Suppose we know the number of paths from root a to layer (k-1)
* Now for all the nodes b in the subsequent layer: In all the shortest paths from a and b, the vertices before b must be in the layer (k-1). Thus we get the number of paths ending in b = sum of all the number of paths ending in every subsequent edge in the layer.

**Question 3**

Lemma:

To prove that the greedy algorithm always stays ahead.

Proof:

We will use proof by contradiction.

* Let’s assume that the greedy algorithm is not optimal and another algorithm works better than it.
* Then let’s prove this using induction
* The greedy algorithm sends greater than or the same number of boxes in the first n trucks.
* Base Case: In the base case of the proof, we consider the situation where we are dealing with the first truck (n = 1). The base case is trivially true because there are no previous trucks to compare with. Therefore, there are no other algorithms to outperform the greedy algorithm in this scenario.
* Inductive Step: Suppose that our claim is true for n-1 trucks.
  + The greedy algorithm has sent and the other where
  + The other algorithm sends . And we know that the sum of the weights of ( ) < W as .
  + So we can put our packages of the other algorithm has sent in the truck and then use the remaining space to load the rest of the packages.
  + Thus our algorithm is ahead after n steps.
* This concludes the proof and we can have proved that the greedy algorithm uses less than or equal to the number of trucks than any of the other algorithms.

**Question 4**

Algorithm:

* Suppose if are the swimming, biking, and running time for the contestant .
* Now we sort all of the contestants on the basis of in decreasing order
* Now, we claim that this is the optimal order

Proof:

* We will prove this using contradiction. Suppose that the optimal order is something different than what we suggested.
* Thus, the optimal algorithm does not follow decreasing order in terms of , which means that we can find two contestants , where contestant starts immediately after . Thus,
* Now our claim is that if we swap the position, the order that we get is better.
* Notice, that the pool is in use of swimming time for both the cases.
* The finishing time of the pool usage of contestants would be modified with the time they started using it. Thus, the finishing time is the maximum of ) & ) , which was originally .
* Now, as , the finishing time of the modified algorithm is better, and this is a contradiction.
* Thus, the algorithm with an inversion is not optimal and our initial algorithm and claim is correct.

Runtime:

* Calculating running and biking time for all the contestants: O(n)
* Sorting: O
* Total runtime: O

**Question 5**

**Part (a)**

* Suppose link parameter is r = 4000
* We have and
* While does not hold here, and running these two streams in the order (1, 2) is valid.

**Part (b)**

Algorithm:

* Sort the streams by the ratio in decreasing order. This means we will prioritize streams that have the highest number of bits per unit time.
* After sorting, go through the streams sequentially and keep track of the total number of bits sent (B) and the total time elapsed (T).
* At each step, check if B r × T. If this condition is ever violated, the schedule is not valid according to the constraint, and the algorithm should return false.
* If the end of the list is reached without violating the constraint, then the schedule is valid, and the algorithm returns true.
* Psuedocode:

IsValidSchedule (streams, r):

Sort streams by in non-increasing order

B = 0 // Total bits sent  
 T = 0 // Total time elapsed

for each stream i in streams:

B = B +  
 T = T +

if B > r\*T:

return false

return true

Proof:

* Suppose there is a valid schedule S that does not follow the order of .
* This means there are two consecutive streams in S where <
* If we swap streams , the total bits sent by time remains the same, and thus the schedule remains valid up to this point. However, the swap can potentially make the schedule valid for later times if it was invalid before because we are sending da "heavier" stream earlier, which can prevent the total sent bits from exceeding r ×T at any point.
* By repeatedly applying this exchange, we can transform any valid schedule into one that follows the order of without invalidating it.

Runtime:

* Sorting the streams: O
* The for loop: O(n)
* Total runtime: O

**Question 6**

Algorithm:

* Initialize a queue to keep track of rotten oranges.
* Count the number of fresh oranges.
* For each rotten orange in the matrix, add its position to the queue.
* Process the queue in a loop simulating each day:
* For each orange in the queue, rot its neighboring fresh oranges.
* After processing all oranges for the day, increment the day counter.
* At the end of the day, convert all dirty oranges to rotten and add them to the queue for the next day's processing.
* If there are still fresh oranges left after the queue is empty, return -1, indicating it's impossible to rot all oranges. Otherwise, return the day counter.

Proof:

* We need to prove that if there is a shortest path a (length k) from a rotten to fresh orange, then the fresh orange will definitely rot, and it will rot at time k.
* This claim is proved using the fundamental properties of BFS (level search and traversal).
* It is also evident that if there is no connecting edge between a rotten and fresh orange, then our algorithm will give -1.

Runtime:

* If we take each index as a vertex, there are MN vertices.
* We select each vertex for exploration at most once and explore 4 directions.
* Total runtime: O(MN)