Joyce Chen Dis 1F 2:00PM Parshan Teimouri

CS180 Homework #3

1. **Exercise 10 Page 110**

Problem:

Suppose we are given an undirected graph G = (V, E), and we identify two nodes v and w in G. Give an algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be O(m + n) for a graph with n nodes and m edges.

Algorithm:

- Run BFS on G starting from node v.
  - Use a hashmap to keep track of each node's level.
  - The level of w will be the length of all shortest v-w paths in G.
- Run DFS on G starting from node v. Include a count variable.
  - For each neighbor S of the top node N in the DFS stack:
    - If S = w, increment the count and continue.
    - If **S's level = N's level + 1 and S's level < w's level**, add S to the stack.
    - Otherwise, disregard and continue.
  - Repeat until the DFS stack is empty.
- Return the count, or the # of shortest v-w paths in G.

Proof of Correctness:

- Suppose the algorithm doesn't return the # of shortest v-w paths in G.
- Proof by induction.
  - Base case: Suppose v is the source node on level 0, and w is on level 1. The shortest path from v to w is therefore of length 1, and the # of shortest paths = the # of edges connecting v to w.
- Inductive hypothesis:
  - Assume that our algorithm will return the # of shortest v-a paths in G, where node a lies on the level i-1.
  - The only way our algorithm wouldn't return the shortest # of v-a paths in G for level i would be if it didn't count all of the nodes with level i that are connected to existing explored paths.
  - However, at each node we look at all its neighbors that are exactly 1 level after the node, meaning that we explore all connected shortest paths from v-a.
  - Therefore, it would count all the nodes in level i that are connected to pre-existing paths from v.
- Our algorithm will count all shortest paths leading from node v to node w.

Time Complexity:

- Running BFS on G takes O(V+E) time.
- Running a modified version of DFS on G takes O(V+E) time.
- Therefore, total runtime is O(V+E).

2. **Exercise 6 on page 108**

   Problem:

   We have a connected graph G = (V, E), and a specific vertex u ∈ V. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that G = T. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

   Proof by Contradiction:

   - Assume there is an edge (u, v) in the original graph G that is not in tree T.
   - (u, v) means that v is a neighbor of u, and v is reachable from u in G.
   - Since T is a DFS tree rooted at u, it contains all nodes reachable from u in G using DFS. This includes node v, since v is reachable from u.
   - Since T is a BFS tree rooted at u, it contains all nodes reachable from u in G using BFS. Likewise, this includes node v, since v is reachable from u.
   - Consider the path from u to v in both the DFS and BFS trees.
     - DFS tree: this path is the unique path from the root u to v, following the DFS traversal order.
     - BFS tree: this path is the unique path from the root u to v, following the BFS traversal order.
     - Since both trees are the same (T), these paths must be the same.
   - However, this contradicts the assumption that the edge (u, v) is not in tree T, since both the DFS and BFS trees contain a path from u to v
     - v is connected to u in T
     - Edge (u, v) must also be in tree T
     - ><

3. **Exercise 12 on page 193**

   (a) Consider the following claim:

   > Claim: There exists a valid schedule if and only if each stream i satisfies $b_i \leq rt_i$.

   Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

- This statement is false. To provide a counter-example, suppose there are 2 streams (b-1, t-1) = (20, 1) and (b-2, t-2) = (10, 2) where r = 10.
- b-1 is greater than r*t-1 = 10*1 = 10, which doesn't satisfy the condition.
- b-2 is equal to r*t-2 = 10*2 = 20, which satisfies the condition.
- Based on the claim, there will be no schedule that exists with this combination of strings.
- However, the schedule where b-2 runs first is valid. When t = 3, r*t = 30 which is equal to the total number of bits sent across the stream, 10+20.
- Therefore, there exists a valid schedule even if a stream i doesn't satisfy b-i <= r*t-i.

(b) Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n.

Algorithm:

- Sort the streams by their rates, or # of bits / time passed.
- Send the streams in increasing order of rates.
- Create a variable to accumulate the number of bits that have passed through the stream, B.
- Create a variable to count the total time that has passed so far, T.
- Iterate through the increasing rates:
    - Calculate the # of bits in the stream: b-i
    - Add the number of bits to the total count: B += b-i
    - Add t-i to the total running time: T += t-i
    - Check whether B exceeds r*T.
        - If it does, then there is no valid schedule.
- If we have reached this point without declaring an invalid schedule, there is a valid scheduling.

Proof of Correctness:

- Claim: By sorting in increasing rate, every new stream we add will be B <= constant * T if there is a valid schedule.
- Proof by Induction:
    - Base case: there is only 1 stream.
        - If the stream produces a total # of bits > constant * T, then there is no valid schedule.
        - Our claim holds true.
    - Inductive hypothesis: say that we have n streams.

        - Assume that our claim holds true for n-1 streams: every stream we've added so far produces a valid schedule, that is, the cumulative number of bits sent so far doesn't exceed constant * total time passed.
        - In order for our claim to not hold true for the nth stream, it would need to have a rate r and time duration t such that r * t + B > constant * (Tn-1 + t), where Tn-1 is the cumulative time passed for n-1 streams.
        - Because B is at most constant * Tn-1, we can generalize the equality to r * t > constant * t.
        - This would mean that the rate is greater than the constant, or r > constant, and all future rates after n will be greater than the constant as well.
        - Therefore, the statement r * t > constant * t will hold true for all streams after n. This means a valid schedule is impossible, as the number of bits will always exceed constant * T.
        - Thus, if we encounter a stream that creates an invalid schedule, there would be no valid schedule ordering overall.
        - Our claim is true.

Time Complexity Analysis:

- Sorting the streams by rates takes O(NlogN) time, N = # of streams.
- Iterating through the streams takes O(N) time.
- Total time: O(NlogN) time.

4. **Exercise 3 on page 189**

Problem:

- Each truck has a fixed limit W (max weight it can carry)
- Each box i has weight w-i.
- Each box must be processed in order of arrival.
- Let S be the set of trucks that our algorithm produces. Suppose there exists some optimal solution S* such that the # of trucks is minimized.
- Let f(i, S) be the truck that the ith box boards in S. Let f(i, S*) be the truck that the ith box boards in S*.

Claim: For all boxes r…k where r <= k, we have f(i-r, S) <= f(i-r, S*).

Proof by induction:

- Base case

    - S places the first box i-1 into the first truck that can hold its weight.
    - This will always be 1 truck, which is the minimum number of trucks that can be sent off. f(i-r, S) = f(i-r, S*) = 1.
    - Our claim is true.
- Inductive Hypothesis
    - Let r > 1. Assume that our claim is true for r-1, and prove it for r.
    - Therefore, assume that f(i-r-1, S) <= f(i-r-1, S*). In order for this statement to be false on the r-th box (more trucks are sent off for r boxes in our algorithm compared to the optimal solution), we would need to place the r-th box on a new truck for send-off.
    - In other words, our algorithm would need to "fall behind."
    - However, this would not happen, because our algorithm always places the next box into the first truck available. S* must have it such that this box is placed into a new truck or the original truck. Therefore, our solution's truck # will either be less than or equal to S*'s truck.
    - f(i-r, S) <= f(i-r, S*); therefore, our claim is true.

<u>Claim</u>: The greedy algorithm for placing boxes in the first truck that has enough space (not reached limit W) is optimal.
<u>Proof by contradiction:</u>

- Assume that the number of trucks in our solution |S| is greater than the number of trucks in the optimal solution |S*|, such that |S| > |S*|.
- Because |S| > |S*| there is some additional truck k in S at the arrival of package p such that the number of trucks in S is greater than that in S*.
- At package p, f(p, S) <= f(p, S*) due to the last proof, so the number of trucks for S must be less than or equal to the number of trucks in S*.
- ><

5. **Exercise 6 on page 191**
   <u>Problem:</u>
   What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

   <u>Algorithm</u>:
   - Let each contestant i have their projected times of (s-i, b-i, r-i) where s-i is their projected swimming time, b-i is their projected biking time, and r-i is their projected running time in the triathlon.
   - For each contestant, add their biking and running times b-i + r-i

- Sort the contestant's projected times in decreasing order of their biking + running times, so b-i1 + r-i1 > b-i2 + r-i2 > … > b-iN + riN
- This will give the most optimal schedule (smallest completion time).

<u>Proof of Correctness:</u>

- The best order for sending people out is by sending those with longer biking + running times first, because other contestants are able to start their triathlons in parallel to the biking + running.
- Because the b+r time cancels out the swim time in terms of calculating when the next contestant will run, by having longer b+r times run first will cancel out the longer swim times as they appear.
- Let i-1 … i-N be the set of contestants in the triathlon.
- Let S be the contestant schedule that our algorithm produces, and S* be the optimal contestant schedule.
- Let f(i, S) be the completion time if we consider just the first i contestants in our solution S. Let f(i, S*) represent the same for optimal solution S*.

- <u>Claim</u>: For the number of contestants r…n where r <= n, we have f(i-r, S) <= f(i-r, S*). In other words, the completion time for "r" contestants in our schedule will be less than or equal to the completion time of the optimal solution's schedule.
- <u>Proof by Induction</u>:
  - Base case: 1 contestant in the triathlon
    - S will send out that contestant, which produces the schedule with minimum possible completion time.
    - Claim is true.
  - Inductive Hypothesis:
    - Let i > 1. Assume that our claim is true for i-1, and prove it for i. So, assume that f(i-1, S) <= f(i-1, S*).
    - For f(i, S) > f(i, S*) to hold true, the swim time of S's chosen contestant must induce a greater completion time than S*'s. We don't consider the b+r times, because they are already the most optimal (minimum) based on our algorithm of choosing the shortest b+r times first.
    - Suppose the time it takes for the nth contestant of S to finish swimming = s-1 + s-2 + … = s-n.
    - Because the contestants are sorted in decreasing order of b+r times, the projected swim time for the ith contestant of S will be at least as long as the projected swim time for the i-1th contestant.

            - For S, the total time it takes for all i contestants to finish swimming will be at most s-1 + s-2 + … s-i-1 + s-i = s-i + s-i = 2s-i.
        - For S*, the completion time is s-i + C, where C is the minimum possible completion time for i-1 contestants.
        - Because s-i >= s-(i-1) and s-(i-1) + C >= 2s-(i-1), we have s-i + C >= 2s-i, or s-i + C >= C.
        - Therefore, our algorithm S produces a completion time of at most the completion time of S*.

<u>Time Complexity Analysis:</u>

- Sorting the contestants by the sum of their bike + run times takes O(NlogN) time.
- Processing the contestants (ie. computing the total time of the schedule returned by the sort) takes O(N) time.
- Total: O(NlogN) time.

6. **Rotten Oranges Problem**

<u>Algorithm</u>:

- Represent the matrix as a graph G, where each node is an orange (either rotten, fresh, or empty) and each edge connects an orange to an adjacent orange. If there is an edge between a rotten and fresh orange, the fresh orange will be turned rotten when the edge is explored.
- Run a BFS on G. Create a BFS queue and call it Q.
- Add all rotten oranges to Q. Then, add an indicator of the current time frame into the queue.
- Keep a variable T that tracks the total # of time frames passed.
- While Q is not empty:
    - Pop Q.
        - If the element is a time indicator:
            - Increment T by 1.
            - Append a new time indicator to the end of the queue.
            - Continue.
        - Otherwise:
            - Let the element be orange O.
    - Check if O is fresh:
        - If yes:
            - Mark it rotten and append it to the queue.
        - If no (it is either empty or already rotten):

        - Ignore and continue.
- Run a BFS on G again to check whether there are any fresh oranges left.
    - If there are, then return -1 as it is impossible to rot every orange.
    - If not, then return the time T, or the minimum time required to rot all the oranges.

Proofs of Correctness:

- Claim: Suppose that our algorithm doesn't rot all possible oranges.
- Then, at least one fresh orange exists that has a rotten orange adjacent to it by the end of the algorithm.
- In that case, the rotten orange must have not been added to the BFS queue, because we would have explored all the fresh oranges adjacent to the rotten orange by our algorithm if it were in the queue at one point.
- For an orange to be rotten, it would have either started off rotten or been turned rotten by an adjacent rotten orange.
    - If it started off rotten, our algorithm would've added it to the queue at the beginning. ><
    - If it were turned rotten, it would've started off as a fresh orange, been marked rotten, and appended to the queue. ><
- Therefore, all possible fresh oranges will be marked rotten by the end of our algorithm.

- Claim: Suppose that our algorithm doesn't find the minimum time required so that all oranges become rotten.
- Then, our algorithm returns a longer time to rot all oranges.
- This means that we do not rot all possible oranges in 1 single time frame.
    - In other words, there exists at least 1 fresh orange adjacent a rotten orange at time frame t-i that did not get marked rotten.
- For time t = 1:
    - We start by appending all existing rotten oranges to the queue.
    - We explore all fresh oranges adjacent to these rotten oranges, before incrementing t.
    - We mark these fresh oranges rotten. Therefore, those are all the fresh oranges that can possibly rot at t = 1.
- For time t > 1:
    - We explore all neighboring oranges of all newly rotten oranges from the previous round; no need to explore those of previously rotten oranges because we've already explored all their neighbors in the round before.

    - We mark all neighboring fresh oranges rotten before incrementing t.
- Therefore, at any point in time, we are rotting all adjacent fresh oranges to rotten oranges; the time t is minimized that way.
- Our algorithm will find the minimum time required such that all oranges become rotten.

Time Complexity Analysis:

- Let N be the # of oranges (nodes) and M be the # of connections between oranges (edges).
- We run an initial BFS on G to find all the rotten oranges and add them to the queue, which takes O(M * N) time.
- We do another BFS on G to find all the fresh oranges adjacent to the rotten oranges, which takes O(M * N) time.
- Since there is a possibility that a fresh orange was not encountered by a rotten orange (perhaps it was surrounded by empty cells), we run a final BFS check to see if it is impossible to rot all fresh oranges – O(M * N).
- Total time = O(M * N).