Homework 3

Tejas Kamtam

305749402

Discussion 1B - Friday, 10 am

TA: Parshan

Problem 1

> Page 110, Exercise 10

Use a modified BFS and keep track of all paths and path lengths of $v \rightarrow w$.

Algorithm

- Create a queue, a visited array, and attach the level 0 to v (the root/starting node) and append the node v to the queue
- Create a hashmap to map path lengths of $v \rightarrow w$ to the frequency they occur
- Pop the queue and add the resulting node n to the visited array if it's not there already
- If $n == w$ increment the frequency value in the hashmap that corresponds to the key of the level of n (i.e. the hashmap maps path lengths to frequency so we might have something like `{4:0+1, 3:2, 7:4}` which just incremented the frequency of the path length of 4)
- Now, grab all of the immediate children of n that are not in the visited array

- If any of the children are w, increment the frequency of the path length that corresponds to 1+ the level of n (i.e. w is 1 more edge away from the current node n)
- For the remaining child nodes (because we do not attach levels to w nor add it to the visited array) that are not w, attach the level of $\text{level}(n) + 1$ to each of them and add them to the queue and the visited array
- Repeat this while the queue is not empty
- After this loop, we will have a hashmap of `{path length : frequency}`, we can find the minimum path length and return its frequency as the number of shortest paths from v to w

Time Complexity

- The graph exploration is a modified BFS with modifications that occur in constant time (hashmap insert/update) so the run time is still $O(n + e)$ where we at worst search all nodes n and edges e once
- Finding the minimum in the hashmap is an operation linear in $O(n)$
- So, the total time complexity is:

$$O(n + e)$$

Proof

- We only have to run BFS from the node v as we only want to see paths from v to b $\rightarrow$ any disconnected graphs/nodes will never be considered and will never have to be considered since the social network is guaranteed to be connected to the start node by its definition
- Out of the possible algorithms for graph traversal, BFS is the optimal choice because it explores nodes in

"level order" as discussed in class, so every time the algo visits a node, it would have been when that node was the shortest distance away from the start node v, thus we are always only ever storing (accounting for) the shortest paths to the hashmap

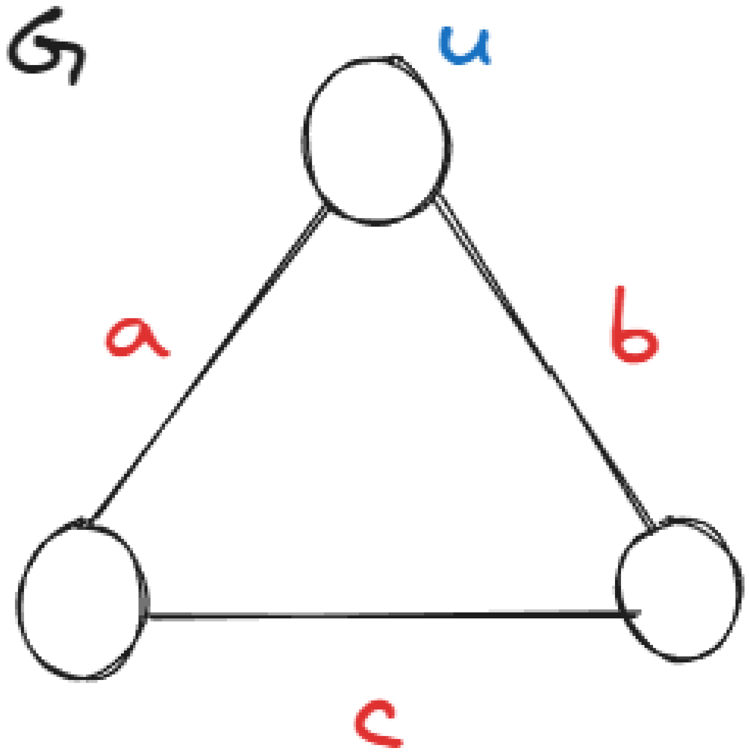
- Because we want to find all paths to w from v we should not add w to the visited array and instead allow every node (that can) to "discover" w
- We only ever make constant time modification to BFS by attaching levels to nodes and accessing/modifying values in a hashmap, so our algo is indeed optimal in linear time

Problem 2

> Page 108, Exercise 6

Proof

- Suppose the graph G is NOT a tree, then there must exist a cycle in G
- A BFS tree will not have edges connecting two nodes of the same level as we discussed in class by the BFS Tree property because BFS operate in level order
- A DFS tree, on the other hand, when encountering a cycle MUST include the edge connecting two nodes of the same level as DFS explores all the children of a node depth order
- The following visual describes the situation (where DFS happens to choose the left most branch WLOG):



DFS Tree = $\{a,c\}$

BFS Tree = $\{a,b\}$

-
- In the above example (WLOG) we see that if the graph G is NOT a tree, then

$$\exists e \in E(DFS) \text{ s.t. } e \notin E(BFS)$$

- But this contradicts the fact that the DFS Tree $==$ BFS Tree since a tree is equal only if it has the same root (which is true since they are both rooted at u by the problem statement) AND the same edges (by the definition of a tree)

- So, if and only if the DFS Tree and BFS Tree are the same, then there must not exist such an edge e as described above, so there must not exist any cycles in G
- And, because G has been told to be connected (in the problem statement) and has no cycles, it must be a Tree QED

Problem 3

> Page 193, Exercise 12

Part a

It is not the case that "there exists a valid schedule **if and only if** each stream i satisfies $b_i \leq rt_i$. We can prove this with a counter example, consider:

$$n_1, n_2, n_3 = (1000, 1), (1000, 1), (11000, 2)$$

We can give an example with the ordering 1,2,3 where we have the following network link calculations/validations:
$t = 1 : \text{1st stream sent} \implies 1000 \leq 5000 \cdot 1$
$t = 2 : \text{1st+2nd streams sent} \implies 1000 + 1000 \leq 5000 \cdot 2$
$t = 3 : \text{1st+2nd+half of 3rd sent} \implies 1000 + 1000 + 5500 \leq 5000 \cdot 3$
$t = 4 : \text{all streams sent} \implies 1000 + 1000 + 11000 \leq 5000 \cdot 4$
This shows that this schedule ordering is valid; however, n_3 does not meet the claim proposed which can be seen by the calculation:

$$n_3 : 11000 \nleq 5000 \cdot 2$$

Therefore, this claim does not hold for all schedules by this counterexample.

Part b

Algorithm

- for the first stream, select the stream with the largest $b_i \cdot t_i \leq 5000 \cdot t_i$ and append it to our ordering of selected streams and remove it from the pool unchosen streams
- for the next jth stream, select the stream with the largest b_j such that

$$\sum_{i=0}^{j} b_i \leq 5000 \cdot \sum_{i=0}^{j} t_i$$

  from the pool of unchosen streams and append it to our ordering and remove it from the pool
- continue to do this for all the remaining n streams until we can no longer do so
- if we can no longer select new streams, and our selected ordering of streams contains all n streams, then we can say a valid schedule exists, else we can say a valid schedule does not exist

Time Complexity

- For each stream, we are looping through the entire pool/list of streams again to select the next one which is of $n - i$ element for each ith selection
- So, overall this algorithm has run time

$$O(n^2)$$

Proof

- at each selection, we are making the locally optimal choice (largest interval) and checking that selecting that stream at each iteration does not invalidate the schedule $\rightarrow$ doing so ensures that the overall schedule will is valid since at every selection we only pick streams that allow the schedule to be valid

- because we are making locally optimal choices, the selection of the largest bits-per-interval ensures that any future selections are at most as "expensive" as our previous choice so any one selection of a stream will not invalidate the stream
- Because there is the possibility that we get a set of streams that has no possible valid permutations of a schedule, we return that there is no valid schedule since it is not a part of our chosen schedule
- To prove our algorithm, consider someone else's "optimal" solution. The only case where ours is not the same is if the optimal solution contains a stream ours does not. However, this contradicts the method of our algorithm as we check for each stream that adding it to our schedule, it does not violate the constraint (*). So if the optimal solution has a stream, then our algo is guaranteed to have that stream as well.
- If the optimal solution does not have a stream ours does have, their solution is not optimal as our solution will contain all streams that guarantee a valid stream and thus must include all streams that make a valid schedule.

Problem 4

> Page 189, Exercise 3

Proof

- let us say that our algorithm is the original algorithm that packs as many boxes to trucks as possible in the order they arrive in and the other solution, the proposed "optimal" solution tries to pack at least 1 less box to optimize weight usage in the next truck to make deliveries faster

- consider the most basic case WLOG: the optimal solution packs the first truck T_1 with $N-1$ boxes such that the total weight of their first truck's load is $W - w_i$. Our solution would pack the first truck with as much will fit so our first truck will contain N boxes with a total weight of N
- For our base case, we can see that our solution will be 1 box ahead of the optimal solution.
- Now, inductively, the optimal solution has 2 choices for the next truck T_2: (1) pack the truck to the full W weight capacity or (2) once again leave at least one box out so that the weight is $W - w_i$.
  - 1st case: by the time the second truck departs, our solution will be ahead by 1 box
  - 2nd case: by the time the second truck leaves, our solution will be cumulatively ahead by 2 boxes
- Generalizing this for any truck T_i for which our solution packs up to W weight and the optimal solution packs $W - w_i$ weight, for truck T_{i+1} our solution will be ahead by at least 1 box.
- Thus, for any truck, our solution will always be ahead by at least 1 box.

Problem 5

> Page 191, Exercise 6

Algorithm

- for N contestants with swimming, biking, and running times S_i, B_i, R_i respectively
- we can order the contestants by their $B_i + R_i$ times in descending (non-increasing) order: greatest biking+running time first to least last

Time complexity

- finding the biking+running times of each contestant is $O(N)$
- BUT, the best algorithm we know to sort is using a minheap which is of $O(N \log N)$ time (discussed in class)
- So, the overall time complexity is of

$$O(N \log N)$$

Proof

- WLOG, we can consider the most basic example of contestant a and b with times S_a, B_a, R_a and S_b, B_b, R_b respectively
- We can make the claim that, arbitrarily, $B_a + R_a < B_b + R_b$ (if they are equivalent, their positions on the schedule are interchangeable as we'll see)
- Then, we have 2 cases of possible schedules (swimming time is cumulative by position because we cannot parallelize swimming, each contestant must swim sequentially so we have to wait for each earlier contestant to finish swimming):
  Case 1:

Position	Contestant	Total Time
1	a	$S_a + B_a + R_a$
2	b	$S_a + S_b + B_b + R_b$

Case 2:

Position	Contestant	Total Time
1	b	$S_b + B_b + R_b$
2	a	$S_b + S_a + B_a + R_a$

- Now, assuming non-trivial values (only positive values for any times), we can see that in Case 1, contestant b finishes last thus the total time of the contest is the time contestant b takes; whereas, in Case 2, the total time of the contest is the time contestant a takes
- Comparing these two values we see:

$$(S_a + S_b) + (B_b + R_b) > (S_b + S_a) + (B_a + R_a) \implies B_b + R_b > B_a + R_a$$

- Which is true by the claim we made above. Therefore, the ordering with the slowest biking+running contestants first and fastest last (descending, non-increasing order) is the most optimal and will result in the least overall contest time.

Problem 6

> Given a matrix of dimension M * N, where each cell in the matrix can have values 0, 1 or 2 which has the following meaning:
> 0: Empty Cell
> 1: Fresh Oranges
> 2: Rotten Oranges
> A rotten orange at index (i,j) can rot other fresh oranges which are its neighbors (up,down,left,right).
>
> **TASK:** Find the minimum time required so that all the oranges become rotten or return -1 if not possible
>
> **Example:**
> Input: `[ [2, 1, 0, 2, 1], [1, 0, 1, 2, 1], [1, 0, 0, 2, 1] ]`
> Output: `2`
> Explanation:
> At 0th time frame

```
[2, 1, 0, 2, 1]
[1, 0, 1, 2, 1]
[1, 0, 0, 2, 1]
```

> At 1st time frame

```
[2, 2, 0, 2, 2]
[2, 0, 2, 2, 2]
[1, 0, 0, 2, 2]
```

> At 2nd time frame

```
[2, 2, 0, 2, 2]
[2, 0, 2, 2, 2]
[2, 0, 0, 2, 2]
```

We can run a multi-source BFS to track the spread of rotten oranges

Algorithm

- Traverse the grid and add the locations of rotten oranges (2) to a queue and record the count of fresh oranges and a time step counter set to 0
- for each rotten orange in the queue, travel to its adjacent neighbors (level-wise because it's a BFS) and change any values of 1 (ripe oranges) to 2 (rotten) and decrease the ripe orange count by 1 for each orange we change to rotten and add these visited children (oranges converted to 2) to the queue
- increment the time step counter
- repeat the step above until the queue is empty
- if there are still ripe oranges remaining, return -1 since we are unable to rot all oranges

- otherwise, return the time counter

Time Complexity Analysis

- although we are running a BFS, because we are using an adjacency matrix, the worst operation we do is traverse the grid to grab all rotten oranges, which is $O(M \cdot N)$
- Thus, the overall time complexity is

$$O(M \cdot N)$$

Proof

- We are guaranteed to get the minimal solution because we are using a BFS which does a level-order search and will only make valid moves because it checks only adjacent squares in the grid
- Because the BFS explores all available paths, we are also guaranteed to reach all accessible ripe oranges; thus, if there are any ripe oranges remaining, we are guaranteed to know there is no valid path to it from any rotten orange
- the multi-source implementation of BFS ensures that the timer is only incremented once across a BFS iteration across all rotten oranges, so it checks all possible changes to ripe oranges per time step correctly