CS 580 Algorithm Design, Analysis, And Implementation
Fall 2021 Vassilis Zikas HW 2

Problem 1

Collaborators: List students that you have discussed problem 1 with: Vishnu Teja Narapareddy, Tulika Sureka

(a) (i) This statement is false. Here is the counter example.

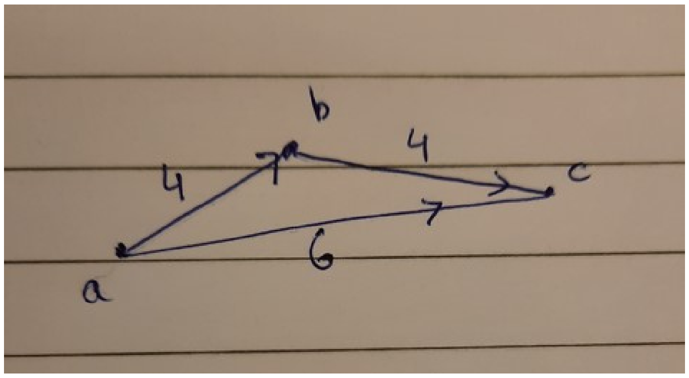

Figure 1: Counter Example

Shortest path from a to c is 6 currently.
After squaring the edges, the shortest path becomes 32 (including vertex b) as it is less than squared distance of 36 between a and c.

(ii) This is true because upon using the Kruskal algorithm to find the MST, the sorted order of edges would still remain the same and hence the same edges would be used in creating the MST for a given graph.

(iii) This is false because when we square the interval duration by extending the finish time for the jobs, there might be case which leads to overlapping between two jobs which were not overlapping earlier.
Let say there were 3 jobs to schedule.
J_1 = [0,3]
J_2 = [3,4]
J_3 = [4,8]
In the current scenario, one can do all the three jobs.
When we square the interval duration and increase the finish time for the jobs, then the new problem looks like this:
J_1 = [0,9]
J_2 = [3,4]
J_3 = [4,20]
Now the optimal schedule would be to do jobs J_2 and J_3. So this shows that order changes.

(b) In this problem, our aim is to find a path p whit the highest probability of not disappearing from the entrance of the maze to it's exit.

Let's assume that s be the entrance vertex and t be the terminal vertex. We need to find path from s to t in which the product of probability of not disappearing is maximised. These given probabilities of not disappearing can be thought of as weight for that edge.

Let the path is represented by p = $\{v_0, v_1, ..., v_m\}$. So we need to maximize this:
p = arg max $\prod_{i=i}^{m} prob(v_{i-1}, v_i)$
However, if we change the weights of these edges by taking the logarithm of these probabilities, we can reduce the problem to the single source shortest path problem (Dijkstra). We will replace the edge weights by the following quantity:
e(u,v) = - log (prob(u,v))
Log is a monotonic function and a negative log would change the minimisation problem into a maximisation problem. So, now our problem reduces to find this:
p = arg min $\prod_{i=i}^{m} e(v_{i-1}, v_i)$
To solve this reduced problem, we can easily use Dijkstra's algorithm to get the shortest path. This path would actually give us the path with the highest probability of not disappearing on the original graph.

Analyzing time complexity :
Initialization of weights : O(E)
Dijkstra's TC : O((E+V)log V)   (using priority queues)
$\implies$ Total TC = O((E+V)log V)

Analyzing space complexity :
Dijkstra's SC : O(E+V)   (using adjacency list to store edges)
$\implies$ Total TC = O(E+V)

Problem 2

Collaborators: List students that you have discussed problem 2 with: Vishnu Teja Narapareddy, Tulika Sureka

(a) Algorithm to minimize the delivery cost of cakes
In this algorithms, I keep a list of stacks and maintain a separate array of their top element. For each cake in the input list, I check if it can be stacked on top of current cakes in any of the stacks made till now. We would place the cake in the first compatible stack. This step can be done using binary search as the array containing the top element of stacks would always be sorted. So, using binary search, we can calculate the first element which is larger than the the given cake size in the array.
If I find such a stack, then I update the top most element for that stack in the array and continue.
Else, I create a new stack and place that cake in the stack and add a new entry in the array while updating the top most element for that new stack in the array.

Analysing TC:
As I would have to iterate through the array maintaining topmost stack element for each cake to find its suitable position for placing it in the stack, binary search has been used for this purpose as this array is in sorted order. So this would lead to a complexity of O(nlogn) where n is the number of elements in the input list because we have to perform binary search (O(logn)) for every element in the input list of size n.
Analysing SC:
Space would be utilised to store cakes in stack and for maintaining the array containing the top most stack

element. This leads to a space complexity of O(n) where n is the number of elements in the input list.

Algorithm 1: Find order of packaging cakes to minimize the delivery cost

Input: Ordered list of cake sizes (L)
Output: List of valid stacks of cakes

```
Function MinBoxes($L$):
    stackList ← null
    stackTop ← null
    for each cake c in L do
        if c can be placed in any of earlier stacks then
            get the first compatible stack s
            add c to that stack s
            update top element in the array corresponding to the stack
        else
            create a new stack s
            put c in s
            add a new element in the array stackTop
            update top element in the array corresponding to the stack s
            add s to stackList
    return stackList
```

(b) Let us assume that by using the greedy strategy, we require t stacks to package all the cakes. So, if we can show that any optimal strategy would also require atleast t stacks for packaging the cakes, then we can say that greedy is optimal.
Proof:
With any given input list of cakes, we can say that we would find atleast one increasing subsequence of t cakes. To show this, we can add pointers from the current element being added to the top most element of previous stack and as we assumed that greedy required t stacks, so we would atleast get one increasing subsequence of length t. The subsequence is strictly increasing because the pointer goes from a larger cake to a smaller cake. So, let say the sequence which we got is like this: $c_1 < c_2 < . . . . < c_t$.
Under the given constraints in the problem that a larger cake cannot be placed on top of smaller cake, we can say that in any optimal strategy, each of these c_i would be placed to the right of the stack containing c_{i-1}. So this shows that optimal strategy requires atleast t stacks for packaging these cakes.
We have shown that optimal strategy requires atleast that many number of stacks as required by greedy. So, this proves that greedy is optimal.

Problem 3

Collaborators: List students that you have discussed problem 3 with: Vishnu Teja Narapareddy, Tulika Sureka

In this problem, using the hint provided, I decided to follow this approach.
We make three copies of each vertex given in the graph G and label them as v_w for white, v_b for black and v_p for pink color. Let say this new graph is G'. Now to connect these vertices in G', we follow this scheme:
If nodes u and v were connected by black edge in graph G, we connect u_w with v_b in graph G'. Similarly, if nodes u and v were connected by pink edge in graph G, we connect u_b with v_p in graph G' and if nodes u and v were connected by white edge in graph G, we connect u_p with v_w in graph G'.

So, in the graph G', we have covered all the edges which were there in graph G without any loss of information. We can say that all the valid walks which were possible in graph G become possible paths in graph G'. This can be shown in the way we have constructed the graph G'. All the valid walks in G would follow the given color order and while incorporating that colored edge in G', we are connecting it the respective colored vertices. Similarly, all the possible paths in G' reduce to valid walk in G.
The graph given to us has non-negative edge weights, so this gives us the hint that we might have to use the Dijkstra algorithm.
So, now we can directly apply Dijkstra algorithm on the graph G' with our starting vertex as s_w, s_b and s_p for three different iterations. For each of the iteration, we take the minimum path between the given s_i and t_w, t_b and t_p.Then return the minimum path cost obtained from the three iterations for s_i. We have to take all the three instances of starting vertex s because walk can start from any colored edge but they have to maintain their order only. Hence, the graph G' ensures that the order of color would be maintained as described by the edge scheme discussed above.
To solve this reduced problem, I have used Dijkstra's algorithm to get the shortest path. This path would actually give us the shortest valid walk in the original graph.

Analysing TC:
For constructing the graph, we need to triple the number of vertices and iterate through all the edges. So, it would take O(E+V) time.
We are making use of Dijkstra algorithm mainly, so the time complexity for this problem is same as that of Dijkstra algorithm.
$\implies$ Total TC = O(3V + Elog3V), where V is the number of vertices and E is the number of edges in the graph.
Analysing SC:
We have tripled the number of vertices and kept the number of edges as same.
$\implies$ Total SC = O(3V + E) where V is the number of vertices and E is the number of edges in the graph.

Problem 5

Collaborators: List students that you have discussed problem 5 with: None

Yes