Problem 1: Road Trip

Answer:

Algorithm description:

1. For each day, loop through the hotels which come after the hotel in which you stayed last night.

2. If a hotel **'h'** is found which is at more than **'d'** distance away from last stayed hotel, then the hotel previous of **'h'** is chosen to stay for that night. This is the greedy step and you stay in this hotel.

3. Repeat step 1 and 2 till we have reached the last hotel x_n.

Running time:

Notice that the worst case occurs if each hotel is at a distance of successive multiples of **'d'**. In such a case, we calculate the distance to each hotel two times in the whole computation.
So, the total running time in worst case is O(2n) = O(n). This is linear time.

Problem 2: CLRS 16-1-2 Activity Selection Last-to-Start

Answer:

The proposed approach – selecting the last activity to start that is compatible with all previously selected activities – is really the greedy algorithm but starting from the end rather than the beginning.

Another way to look at it is as follows. We are given a set S = $\{a_1, a_2, \dots , a_n\}$ of activities, where a_i = $[s_i , f_i)$, and we propose to find an optimal solution by selecting the last activity to start that is compatible with all previously selected activities. Instead, let us create a set S' = $\{ a'_1, a'_2, \dots a'_n\}$, where a a'_i = $[f_i, s_i)$. That is, a'_i is a_i in reverse. Clearly, a subset of $\{a_1, a_2, \cdots, a_n\} \subset$ S is mutually compatible if and only if the corresponding subset $\{ a'_1, a'_2, \dots a'_n\} \subset S'$ is also mutually compatible. Thus, an optimal solution for S maps directly to an optimal solution for S' and vice versa.

The proposed approach of selecting the last activity to start that is compatible with all previously selected activities, when run on S, gives the same answer as the greedy algorithm from the text – selecting the first activity to finish that is compatible with all previously selected activities – when run on S'. The solution that the proposed approach finds for S corresponds to the solution that the text's greedy algorithm finds for S', and so it is optimal.

Problem 3. MST

Answer:

(a)The minimum spanning tree of the tree will not change.

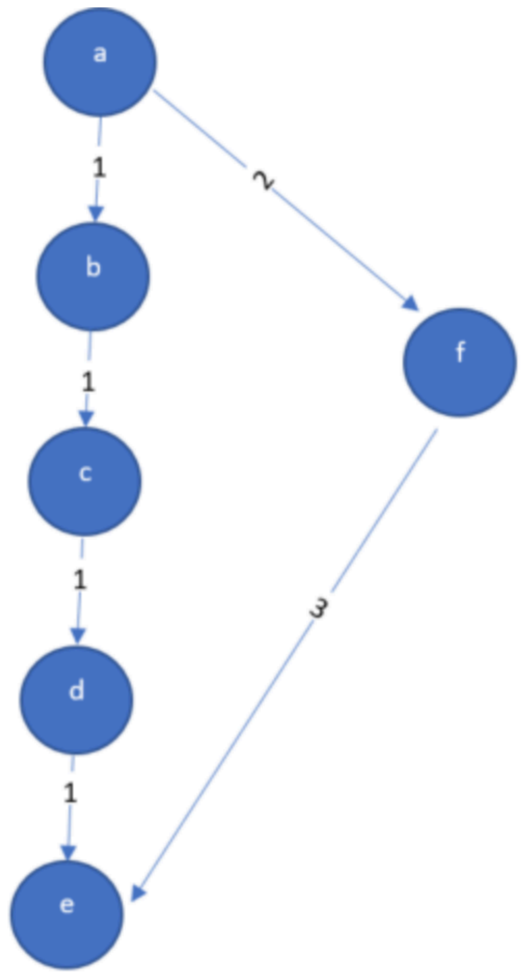
Explanation:

- Each edge has a distinct weight.
- Consider two edges e1 and e2 of the graph with weights w1 and w2, where w1 < w2.
- Consider that e1 is a part of the MST whereas e2 is not.
- 1 is added to the weights of all the edges of the graph.

- The edge that would be picked would still be e1 as the weight of e1 is still less than the weight of the edge e2.
- Similarly, the edges that were picked to form the original MST would still have less weight than the edges that weren't picked.

Therefore, the MST would remain the same.

(b)The shortest paths of the graph may change.
Example:
Case1: Consider the graph G to be as follows:

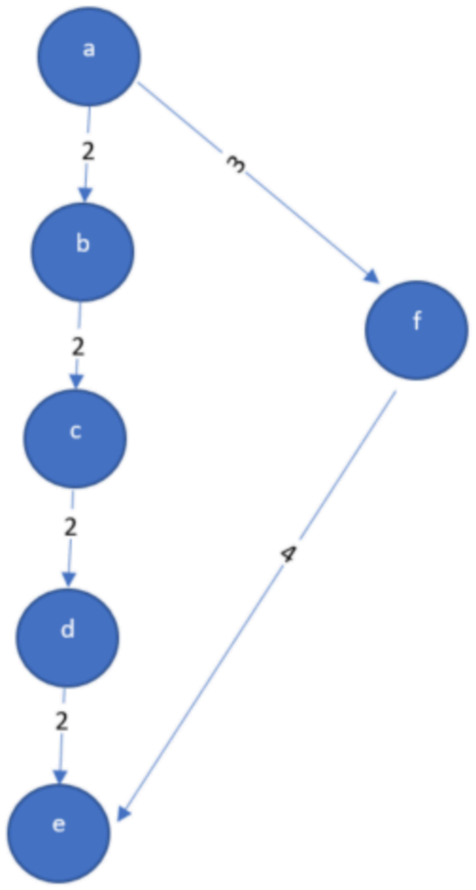


The shortest path in the graph G from a to e is as follows:

- **a -> b -> c -> d -> e**
- **This path has a total weight of 4 units.**

Now, add 1 unit to the weight of all the edges of the graph G.

The graph G is now as follows:



The shortest path from a to e in the graph G is now as follows:

- a -> f -> e
- This path has a total weight of 7 units.
- The path a -> b -> c -> d -> e now has a weight of 8 units.

Since, the shortest path from a to e changed after adding 1 to the weight of all the edges of the graph G. Therefore, the shortest paths of a graph may change after adding 1 to the weight of all the edges of the graph.

Case 2: same graph, assume: the weight of a->b->c->d->e are 1, and the weight of a->f is 200, the weight of f->e is 300. When we add weight value 1, the shortest path will not change.

Overall, the shortest path maybe change.

Problem 4: Euclidean MST Implementation

Answer:

Code: In teach

- A verbal description of your algorithm and data structures

Use prim algorithm to solve the problem.

```
create a list visited to check vertex is visited or not
Assign distance of every node as infinite
Assign distance of first node as 0
Repeat below step until any node is unvisited
    pick a vertex which is not visited and distance is minimum and add it to a visited list
        for all adjacent vertices which are not visited to the current selected vertex
            update the distance
```

data structures:

I use prim algorithm to solve the problem. In the algorithm, I'll use two-D array to represent a point in my program, which stores the coordinate x and y of the point. I also construct a class of GRAPH to store the list things:

```
int size;// total node size
int **ary;// distance d array
Node *node;// node list
int ind;// current index
// calculate distance
```

int distance{}//nearest the distance: $\sqrt{(x1-x2)^2+(y1-y2)^2}$

- The pseudo-code for your algorithm.

#Let G[V][V] contains the weights of all the edges.
#visited[V] is initialized to false and keeps track of all the vertices if they are visited or not.

#min_cost[V] is initialized to INF for every vertex , it keeps the minimum distance of a point not currently in the tree from all the points which are currently in the tree.

```
EMST():
  root = 0 #any random node
  min_cost[root]=0
  for j in range(n):
    selected_vertex = -1
    for i in range(n):
      if( !used[i] && (selected_vertex==-1 || min_cost[i]<min_cost[selected_vertex]) ):
        selected_vertex = i

    if(min_cost[selected_vertex]==INF):
      return "No EMST"
    visited[selected_vertex]=true

    #update the min_cost array for rest of the vertices
    for i in range(n):
      if G[selected_vertex][i]<min_cost[i] :
        min_cost[i]=G[selected_vertex][i]
```

- Analysis of the theoretical running time of your algorithm.

Time complexity of my program is O(n^2). Where n is number of vertices in Graph. In Euclidean MST problem all the 'n' points are connected to each other, and becomes the maximally dense graph. I use prim's algorithm to solve it. A simple implementation of Prim's, using an adjacency matrix or an adjacency list graph representation and linearly searching an array of weights to find the minimum weight edge to add, requires O(|V|^2) running time. V is the number of vertex.