Assignment 05

B403

May 2021

Clare Tidmarsh

PROBLEM 1

Given a 2D array flights, and flights[i]= [u, v, w], where u is the start city, v is the end city, and w is the price to travel from u to v.

For Example:

Input: flights = [[0,1,100], [1,2,100], [0,2,500]], source = 0, destination = 2, k = 1

Output: 200

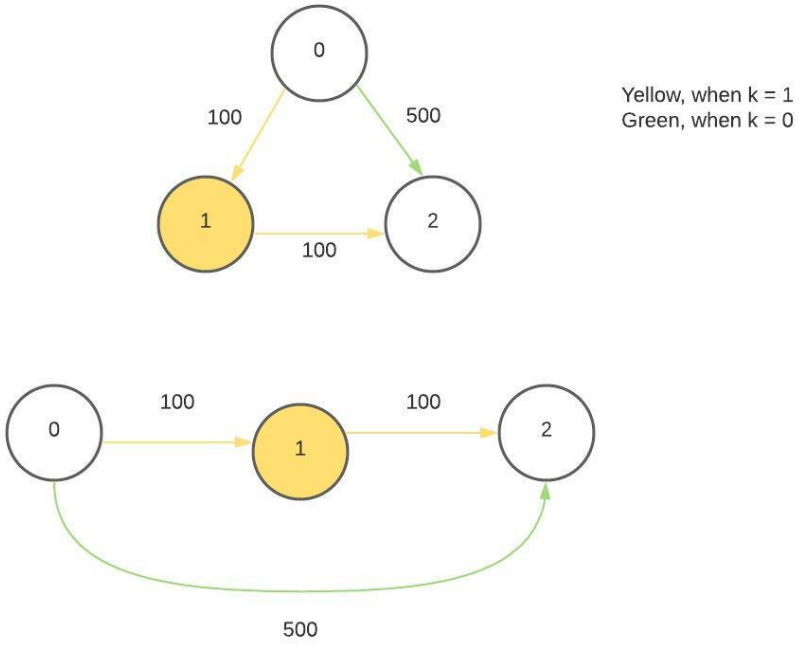
Explanation: The cheapest price from city 0 to city 2 with at most 1 stop costs 200.

Input: edges = [[0,1,100], [1,2,100], [0,2,500]], source = 0, destination = 2, k = 0

Output: 500

Explanation: The cheapest price from city 0 to city 2 with at most 0 stops costs 500.

(a) Describe and analyze an algorithm to find the cheapest price to travel from source city to destination city with up to k stops and if there is no such route, the output should be -1. (Pseudo code is expected with explanation) (10)



To begin, I made a visualization of the two examples given in Problem 1. The yellow arrows and node correspond to the first example where the starting position is city 0 (source = 0), only one stop is made (k = 1), and the output is 200 (cost = 200). The green arrow corresponds to example two where the starting position is city 0 (source = 0), zero stops are made (k = 0), and the output is 500 (cost = 500). The question asks to find the cheapest flights with up to an unknown k stops. The key word in this problem is “cheapest” which can translate into the words “minimum cost”. Therefore, in order to find the minimum cost, or cheapest, of a flight, we will have to create an algorithm similar to the Uniform Cost Search algorithm, where the values of the edges of the nodes matter the most. However, we will need to consider all possible paths in the graph. A slightly modified Depth First Search will be the best algorithm to use while keeping a sum of the costs of each path taken. Since this is a directed graph, Depth First Search will traverse through the graph as path 1 = node 0, node 1, node 2. Then recursively backtrack to the root, for path 2 = node 0, node 2. Each edge sum of each path will be stored and the lowest sum of edges in a path will determine which path is returned. By the end, all possible paths will be explored as well as all nodes and edges will be visited. By visiting all nodes and edges, we can find the best path with the lowest cost. Assuming that we already have a Node and Edges class,

```
Initialize Graph, G, where G hold all nodes in the tree
Initialize a stack, S, with the root as its first element
Initialize the Sum as an integer 0
Initialize the desired destination, v
Initialize Visited, visited[u], where visited is set to false
Initialize CountNumStops as 0
Flights(G, u):
        Starting from the root, for each node in G
                If a node is already visited,
                        Move to the next neighbor
                When a node is visited for the first time, set visited as true for that node
                Add the visited node to S
                Add the edge[w] from each node to the sum until path is complete to v
                Set the sum = sum + edge[w]
`               CountNumStops++
                Recursively go through Depth First Seach in the G at d (DFS(G, v))
                If the current sum is less than the previous sum
                        Set the current sum as the sum
                Return the path of the final sum from S
        If CountNumStops is greater than k
                Return -1
        End
```

(b) What is the time and space complexity? Justify the same. (5)

Since this is supposed to be a modified version of Depth First Search, the time complexity will be O(n^m) because each node is only visited once and the edges are also only visited once. The n represents the number of nodes and the m represents the edges in the graph. Therefore, time complexity depends solely on how many nodes and edges there are in the tree. The space complexity is O(n) since each node and edge is only visited and stored once. The space complexity is only useful when the graph is small otherwise the algorithm will require a lot of storage if it were to keep to the O(n) space complexity.

PROBLEM 2

Cameras are to be installed on the nodes of a binary tree. Each camera at a node can

monitor its parent, itself, and its immediate children.

For Example:

Output: 1

Explanation: One camera is enough to monitor all nodes if placed as shown.

Output: 2

Explanation: At least two cameras are needed to monitor all nodes of the tree. The

above image shows one of the valid configurations of camera placement

(a) Describe and analyze an algorithm to find the minimum number of cameras needed

to monitor all nodes of the tree. (Pseudo code is expected with explanation) (10)

Assuming that a class that defines the nodes, edges, and the graph is already created, we can start at the root of the tree and look for its children. Start by creating an empty queue to store the nodes that will be cameras. Then identify the left child node of the root if it exists, then the right child node of the root if it exists. If the left child of the root exists, set the left child to be a parent, then identify if it has children. First check if it has a left child, if it does, then check the if the child has a neighbor. Since the parent has at least one child, add the parent node to the queue. After getting the neighbor of the parent's left child (the right child), visit the neighbor of that child (the neighbor of the right child). If a child exists, then add the parent of that child to the queue of cameras. This algorithm is similar to Breadth First Search as every node on each level is visited quicker than the Depth First Search algorithm as used in Problem 1. Breadth First Search is more efficient than depth first search as we only need to check if at least one child of each parent exists for the parent to be considered a camera. Unlike Problem 1, the cost of edges will not and cannot be considered in figuring out how many cameras there are. The program will then return the number of cameras in the graph as well as the list of nodes that are cameras.

```
Initialize Graph, G, where G hold all nodes in the tree
Initialize an empty queue for nodes visited, V
Initialize an empty queue for cameras, C
Initialize Count for the number of cameras existing
Initialized visited to be false
CamerasInTree(G, root)
        Initialize the parent to be the root
        Mark the parent as visited
        For each node in G,
                If the parent has a left.child
                        Visit the left child
                        Mark the parent's left child as visited
                        Add left child to queue, V
                        If the left.child has a neighbor,
                                Visit the neighbor (right.child)
                                Mark the right.child as visited
                                Add left child's neighbor (right.child) to queue, V
                                Set left.child as parent
                                Add parent to C
                                If right.child has neighbor,
                                        set the root's right child as the parent
                                        Add parent to C
                                        Visit the neighbor and mark as visited
                Return the list of cameras
        For each i in C,
        count++
return the number of cameras (count)
```

(b) What is the time and space complexity? Justify the same. (5)

The time complexity will also be O(n^m) + O(n), where n is the nodes and the m are the edges. Plus, after the program iterates through the entire graph, it will then iterate through all of the nodes that where added in the camera queue. It will then count how many are in the queue and then return the final count. That way both the list of nodes who are cameras (in order) will be returned O(n^m) as well as how many cameras there are will also be returned O(n). The space complexity will also depend on how many nodes there are in the graph. As it should be O(n^m). With it being O(n^m), you will have more space and will also be better equipped for larger graphs. Because the purpose is to determine whether a node is a camera or not and be needing to follow the given rules, the algorithm will need to be able to store at least all nodes and edges even if all nodes are not cameras, the algorithm will still need to visit all nodes and take the edges into consideration. Looking back, I should have implemented for loop to check how many edges a parent has. That would then tell you if it has any children and then if it has at least one child, add the parent to the queue of cameras. For the algorithm to be more efficient, I should have implemented that instead of checking whether there is it a left and right neighbor and then adding that C. However, the space complexity should be the same, if not, close to O(n^m).