CSC3100_Homework4_Report

Question_1

What algorithm do in program

In problem 1 we generate a binary tree first. And we use DFS algorithm to search for the target node. First we set a `Node` class to represent the node of the binary tree, the object has three characters of key, index of left child, and index of right child. We transform the weights into a list of numbers and create a binary tree according to the input information, if the node has no child, index of child will be `-1`.

The algorithm mainly use recursion method to retrieve the binary tree, and the main function of the programe is `is_symmetric` function to judge whether the subtree at the node is a symmetric binary tree. If it is, it returns `True`, and if not, it returns `False`. If the subtree is a symmetric tree, then the function counts the number of subtree. Finally count the tree node by node, we get the max number of subtree.

Why use this way to solve the problem

The most time spending process in this program is searching all nodes. Using knowledges in lectures, we can use dis method to solve the problem. We can also use recursion method with memory method, however, the memory will be extremly large for this problem if the binary tree has so many nodes. So here we use this way to solve the problem.

the code is as below:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
cnt = 0
n = int(input())
ans = input().split(' ')
nodes = []
for i in ans:
    nodes.append(Node(int(i)))
for n in nodes:
    idx_c = input().split(' ')
    if idx_c[0] == '-1':
        n.left = None
    else:
        n.left = (nodes[int(idx_c[0])-1])
    if idx_c[1] == '-1':
        n.right = None
    else:
        n.right = (nodes[int(idx_c[1])-1])
def is_symmetric(idx_l, idx_r):
    global cnt
    if idx_l == None and idx_r != None:
        return False
    elif idx_l != None and idx_r == None:
        return False
    else:
        if idx_l == idx_r:
            return True
        elif idx_l.key == idx_r.key:
            cnt += 1
            return is_symmetric(idx_l.left, idx_r.right) == True and 
is_symmetric(idx_l.right, idx_r.left) == True
        else:
            return False
ans = 0
for i in nodes:
    cnt = 1
    is_symmetric(i.left, i.right)
    if is_symmetric(i.left, i.right) == False:
        cnt = 0
    ans = max(ans, cnt)
print(ans)
```

Question 2

What algorithm do in program

We use hash in this problem. In the algorithm, in order to delete the repeated numbers, we have to tranverse the input number list. Here we use a dictionary `D` to store the number which already appeared in the previous text. When transverse the number list, check if the number is in the dictionary. If yes, then we ignore the number, if no, then we add the number into both dictionary and the output list `text`. Finally, when the input is totally transversed, output the list is the non-repeating number list.

Why use this way to solve the problem

To save the running time of this program, we use a dictionary to keep as a memory unit. This avoid the condition that we have to transverse the number list so many times. Using hash table, we can get the output more quickly than searching directly

Time Complexity

$T = O(n)$

The code is as below:

```
def main(t):
    tmpL = []
    D = {}
    D = D.fromkeys(t)
    for i in range(len(t)):
        if D[t[i]] == None:
            D[t[i]] = 1
            tmpL.append(t[i])
    for i in range(0,len(tmpL)-1):
        print(tmpL[i],end=" ")
    print(tmpL[-1])

text = list()
num = int(input())
for i in range(num):
    total = int(input())
    data = input().split(' ')
    text.append([int(i) for i in data])

for i in range(len(text)):
    main(text[i])
```

Question 3

What algorithm do in program

We use Bellman algorithm for this problem. In this method first we set an object `Edge` to keep its weight and connection. And we have a list `edges` to store the two vertices connected by the edge and its weight. And we set Infiniti as `0x3f3f3f3f`, also use list of `first_p` and `second_p` to keep the rank information for taking the second shortest path. For input part, we input the edges into edges list. And we implemented a queue `Q` to be store the vertices in the process. Append the new neighbors of the current source node, and compare using Bellman method to solve the problem, when the Queue is empty, the nodes are transversed, and we can find our second shortest path.

Why use this way to solve the problem

For the reason that queue can be changed by length, so it would be better than array. And we choose list to represent the graph, otherwise if the edges are sparse, calculation would be complex. Bellman-Ford algorithm can relax the vertices for every edge. This is to find the shortest path. What's more, we save the data before updating the shortest path to be the second path.

The code is as below:

```
max_n = int(1e6)
inf = 0x3f3f3f3f
edge_num = 0


class Edge:
    def __init__(self):
        self.next = 0
        self.point = 0
        self.weight = 0

edge = [Edge() for _ in range(max_n) ]
dir = [0] * max_n
```

```
first_p = [inf] * max_n
second_p = [inf] * max_n
col = [False] * max_n

def append(x, y, z):
    global edge_num
    edge_num += 1
    edge[edge_num].next = dir[x]
    edge[edge_num].point = y
    edge[edge_num].weight = z
    dir[x] = edge_num

def main(edges):
    global edge, first_p, second_p, dir
    for e in edges:
        x, y, z = e
        append(x, y, z)
        append(y, x, z)

    Q = [1] # the Queue to impelemt the
    col[1] = True
    first_p[1] = 0
    while len(Q) != 0:
        u = Q[0]
        del Q[0]
        col[u] = False
        i = dir[u]
        while i != 0:
            flag = False
            v = edge[i].point
            weight = edge[i].weight
            if first_p[v] > first_p[u] + weight:
                second_p[v] = first_p[v]
                first_p[v] = first_p[u] + weight
                flag = True
            if first_p[v] == first_p[u] + weight and second_p[v] > second_p[u] +
weight:
                second_p[v] = second_p[u] + weight
                flag = True
            if first_p[v] < first_p[u] + weight < second_p[v]:
                second_p[v] = first_p[u] + weight
                flag = True
            if flag and not col[v]:
                col[v] = True
                Q.append(v)
            i = edge[i].next
    return second_p[N]

t = [int(i) for i in input().split()]
```

```
N, R = t[0], t[1]
edges = []
for r in range(R):
    edges.append(int(i) for i in input().split())
print(main(edges))
```

Question 4

We choose Krusal algorithm to solve this problem. First sort the edge in edges list by their weights, and give different vertices with different marks. Then transverse the N edges to choose the smallest total weight. Starting finding the vertices, if the position is already exsisting and marks different, then it cannot be a correct set, mark that edge to be the part of the final path, and change the mark of new marks to be the same. If the difference of vertices and edges is smaller than 1, the answer is the sum of weight. To minimize the cost of a network of the planets, so simplify the spanning tree, we know Kruskal's Algorithm can find minimal spanning tree, so we choose Kruskal's Algorithm.

Time Complexity

$T = O(nlogn)$ For Krusal algorithm

$T = O(n^2)$ For generate the numbers

So the time somplexity for this problem is $O(n^2)$

And the code is as below:

```
def Kruskal(N, edges):
    num = 0
    min_cost = 0
    A = [i for i in range(N)]
    edges.sort()
    for edge in edges:
        v1 = edge[1] - 1
        v2 = edge[2] - 1
        if A[v1] != A[v2]:
            min_cost += edge[0]
            num += 1
            element = A[v2]
            for r in range(N):
                if A[r] == element:
                    A[r]= A[v1]
            if num == N - 1:
                return min_cost
    return -1

N, C = (int(x) for x in input().split())
nodes = []
edges = []
for t in range(N):
    nodes.append([int(x) for x in input().split()])
```

```
for i in range(N):
    for j in  range(N):
        if i < j:
            weight = (nodes[i][0] - nodes[j][0])**2 + (nodes[i][1] - nodes[j][1])**2
            if weight >= C:
                edges.append([weight, i+1, j+1])

print(Kruskal(N, edges))
```