CS5010 Algorithm Assignment 4

Q1 Diameter of Tree

The diameter of the tree T is defined as the length of the longest path of the tree. Write an efficient linear time algorithm to find the diameter of the tree.

Hint: You can apply DFS as part of your solution.

Answer

The question requires us to find the longest path in a tree.

The longest path that go over the root, has a length of the height of the left tree + height of the right tree + 1, which means that the longest path would start from the lowest level of leaf node from the left tree, to the lowest level of leaf node to the right tree. The longest path that pass the root should have this rule.

However, it is not correct to say that the longest path in the whole tree would definitely pass the root. However, we can find the longest path in one-way traversal.

The following method solve the problem with O(n) time complexity, where the n is the number of the nodes in the tree.

Maintain a global variable ans, when we traverse to a node, we get the height of its left subtree and right subtree, add them together and plus 1, compare the number to the ans, set the ans to be the bigger value, then return the current height of the node.

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

class Solution(object):
    ans = 0
    def findDiameter(self, root):
        self.ans = 0
        def findHeight(root):
            if not root:
                return 0
            L = findHeight(root.left)
            R = findHeight(root.right)
            self.ans = max(self.ans, L + R + 1)
            return 1 + max(L, R)
        h = findHeight(root)
        return self.ans
```

The Algorithm should have a time complexity of O(n), whic is linear time complexity, and O(1) space complexity.

Q2 Topological Sort

One interesting way to implement topological sort in a directed acyclic graph G is to repeatedly finding nodes of in-degree 0. Describe how you would implement this idea to implement a fully functional topological sort. Write the pseudo-code of the algorithm. Please make sure the runtime of the algorithm is at least O(V+E).

Answer

The graph G must be a DAG otherwise the problem has no point.

When we do not have the check whether there's loop, all we have to do is the following steps:

- Go over each edge, maintain a dictionary(key-value set), the key is every node in the graph, the value is a set, which contains every to node that starts from the key. Put and search operations are O(1) in dictionary(Hash related functions), and go over the edges has O(E).
- Now we go over the dictionary's keys, which we have O(V), find out the key with an empty value, means that this node's in-degree is 0. Push all nodes with 0 in-degree into a stack, and remove them from the dictinary.
- While the stack is not empty, pop out a node each time, for each of the nodes that are still in the dictionary, remove the current node from its value list, if the list is empty, push the node into the stack. We now add the popped node into the result list as part result of topological sort. This will have a time complexity of O(V).

The input is **List<List<int,int>> edges.**

In actual python code, we use collection's deque to implement fast search and pop from the left.

The actual python code are written in the below:

```
import collections
class Soluion(object):

    def topologicalSort(self, edges):
        pair = {}
        for i in edges:
            if i[0] not in pair:
                pair[i[0]].add(i[1])
            if i[1] not in pair:
                pair[i[1]] = set([])
        #Now we have the dictionary, next step is to
        #find the 0 in-degree nodes.
        stack = collections.deque([])
        for i in pair:
            if not pair[i]:
```

```
                stack.append(i)

        while stack:
            currentNode = stack.popleft()
            pair.remove(currentNode)
            for i in pair:
                pair[i].discard(currentNode)
            #discard in set would remove the element if exist.
            #If not exist, ignore and go on.
                if not pair[i]:
                    stack.append(i)
            rst.append(currentNode)

        return rst
```

The time complexity of the algorithm is O(E+V).

Q3 Network Construction Problem

Gracie is a civil engineer in charge of constructing an underground fiber network in the state of Michisota. Michisota has N cities and M roads and you can always travel from one city to any other city in the state through these M roads. Each one of the M roads connects two town and has a distance marked for each road. The cost of network construction is directly proportional to the total length of the wires laid down under the road. Gracie has to figure out a unique scheme to connect all cities and lay out the fiber with the least possible cost.

Given the list of towns T and roads R connecting them along with their distances D, write an algorithm to design an efficient network construction construction scheme for Gracie that minimizes the overall construction cost.

Answer

Standard MST problem.

There are R roads, each road has its own distance d, since the wire cost is the same when the distance is the same, we only need to choose the roads that connect all the cities with minimum path distance sum.

Then the problem is simplified as how do we choose the path, with minimum cost, but connect all the cities. To solve this we use Greedy approach, for each time we choose the shortest path remained in the paths. Determine if the two cities that are connected by the path are both visited, if so, skip this path; If not, add the cities which are not visited to visited, and add the path to selected path.

However, only conduct such operations may not connect all cities, the may form seperate unions that are not connected to each other. So as we go through each edge, we need to record the union of the city, if the connected cities are in two different unions, we change them into one same unions.

The terminate condition has changed, once all cities are connected and there's only one union

The procedures are simply demonstrated:

- Sort the paths according to the distances.
- Maintain a set that contains all the cities that have been visited before.
- Go over the paths from the lowest cost to highest cost, add the path to the record if the connected cities are not visited. Update their's unions and check for single union.
- Return the collections of the paths.

Make the paths to **List<List<int,int,int>>**, first is from city, second is to city, the third is the distance of the path.

The python code is written below:

```
class Connection(object):
        node1 = ""
        node2 = ""
        cost = 0

        def __init__(self,node1,node2,cost):
                self.node1 = node1
                self.node2 = node2
                self.cost = cost

def MST(inputs):
        current_group_num = 0
        path = []
        group_dict = {}
        sorted_inputs = sorted(inputs,key=lambda x:x.cost)

        for i in sorted_inputs:
                node1 = i.node1
                node2 = i.node2

                if node1 not in group_dict and node2 not in group_dict:
                        current_group_num += 1
                        group_dict[node1] = current_group_num
                        group_dict[node2] = current_group_num

                elif node1 in group_dict and node2 not in group_dict:
                        group_dict[node2] = group_dict[node1]

                elif node1 not in group_dict and node2 in group_dict:
                        group_dict[node1] = group_dict[node2]

                else:
                        if group_dict[node1] == group_dict[node2]:
                                continue
                        else:
                                for k in group_dict:
                                        if group_dict[k] ==
group_dict[node2]:
                                                group_dict[k] =
```

```
group_dict[node1]

                path.append(i)
        group_first = group_dict[sorted_inputs[0].node1]
        for i in group_dict:
                if group_dict[i] != group_first:
                        return []
        return path
```

The algorithm would have a runtime complexity of O(E + V), where E is the number of the paths.

Q4 Single Points of Failure

While designing a robust cellular network, Alice was worried that a single point of failure may completely break the network. A single point of failure is defined as a node in the network graph, which if removed (along with its edges), will cause the number of connected components in the graph to increase. Let us see this with an example:

The graph below is a single connected component. If you remove node 2and edges (2-1, 2-3, 2-4), the resulting graph will still be a single connected component. However, if you remove node 1 and edges (1-0, 1-2, 1-3), you will have two connected components: one with nodes (0,5) and other with nodes (2,3,4). Hence node 1 is a single point of failure.

Write an algorithm to find out all nodes that are a single point of failure in a given graph, as there can be more than one. In the example below, these nodes are: 0 & 1.

Answer

We can use Tarjan Algorithm to solve this problem. We chose any node to be the root of the tree, and use DFS as the base of the algorithm. Maintain three lists in order for future useage. First is DFN, which records the sequence of the node has been visited, or "level" of the node; The second is LOW, which records the current node can access not going through parent node. The third one is visited, records all the nodes that has been visited before.

The Procedures of the algorithm are listed below:

- Start a DFS search from any node, now for simplicity, we start from the 5. Go through every node, set the corresbonding DFN to the depth of the node, firstly set the LOW to be the same as DFN. So the initial graph is:

```
        1(3,3)
      /   |    \
   0(2,2)|     3(6,6)
     |    |  /  |
  5(1,1) 2(4,4)-4(5,5)
```

- When we start from 5, it's still the same (1,1), go to 0, still (2,2), cause no other node can be accessed by 0 other than the parent node. 1 is still the same. When we first went 2 and 4, still the same. But when it comes to 3, the parent node is 4, but there's another edge connected 1. The 1 node has a smaller LOW value, for the connected node is not the parent node, then there is a ancestor node that has a smaller LOW value, we first update the node 3's LOW value.

```
        1(3,3)
      /   |    \
   0(2,2)|     3(6,3)
     |    |  /  |
  5(1,1) 2(4,4)-4(5,5)
```

- Now we go backtrack to the parent node, compare the LOW value all back through, and update the values.

```
        1(3,3)
      /   |    \
   0(2,2)|     3(6,3)
     |    |  /  |
  5(1,1) 2(4,3)-4(5,3)
```

- Now we have finished the traversal, we then go over again the nodes, compare each node's DFN value and LOW value, if the LOW value is smaller than DFN value, it means that the node is connected to the previous ancestor bypass the parent node, which makes the current node not a single point of failure. If the LOW value is bigger or equal to DFN value, then there's no previous ancestor connected to current node, that indicates this is a single point of failure.

- The root is rather special, since the root has no parent node. Then the visited list can be useful to determine whether the root is a single point of failure. If after backtrack to the root, there are still some nodes that has not been visited before, which means that we need to start another DFS on the neighbors of the root, that makes the root a single point of failure.

The python code of the algorithm is written below;

```
class Graph(object):
    self.time = 0
    self.graph = edges #Edges are in List<List<int,int>> form.

    def findSingleFailure(u, LOW, DFN, st, stackMember):
        DFN[u] = self.time
        LOW[u] = self.time
        stackMember[u] = True
        self.time += 1

        for v in self.graph[u]:
            if DFN[v] == -1:
                self.findSingleFailure(v, LOW, DNF, st, stackMember)
```

```
                LOW[u] = min(LOW[u], LOW[v])
            elif stackMember[v] == True:
                LOW[u] = min(LOW[u], DFN[v])

        w = -1
        if LOW[u] == DFN[u]:
            while w != u:
                w = st.pop()
                print w,
                stackMember[w] = False
```

Once the DFN[i] > LOW[i], i is the single point of failure.