CSCI B505 – Fall 2019

Written assignment 6

Dong Liang

11/28/2019

Question 1

A forest is a collection of trees. Given n nodes of a forest and their edges, describe and prove an algorithm (i.e., show correctness) that finds the number of trees in the forest.

Solution:

The number of trees contained in a given forest can be obtained by the following algorithm. The algorithm first builds an adjacency list to store all the nodes in the forest. Then, it takes a node from the adjacency list and use the DFS method to traverse the tree starting with this node. After the traversal is completed, all nodes in this tree are taken from the adjacency list, and the counter (the default is 0) is incremented by one. Repeat the above steps until the adjacency list is empty. The algorithm finally returns the count of the counter. Because every time a tree is traversed, the whole tree is removed from the adjacency list, and the DFS algorithm can traverse the entire tree, this algorithm can count the number of trees in the forest. This algorithm involves constructing an adjacency list and traversing with DFS, both of which take O (N + E). In total, it runs in O (N + E) time.

```
class Node(object):
    def __init__(self, key):
        self.key = key
        self.neighbors = set()

class Graph(object):
    def __init__(self, forest):
        self.forest = forest
        self.existing_nodes = dict()
        self.adjList = set()
        self._build_adj_list()


    @property
    def graph(self):
        return [(node.key, [nbs.key for nbs in node.neighbors]) for node in self.adjList]

    @property
    def tree_number(self):

        count = 0
        while self.adjList:
            first_node = self.adjList.pop()
            visited = self.tree_traversal(first_node)
            self.adjList = self.adjList - visited
            count += 1

        return count
```

```
    def tree_traversal(self, first_node): # DFS
        stack = [first_node]
        visited = set()

        while stack:
            node = stack.pop()
            if node not in visited:
                visited.add(node)
                stack.extend(node.neighbors - visited)

        return visited


    def _build_individual_node(self, key):

        if key not in self.existing_nodes.keys():
            node = Node(key)
            self.existing_nodes[key] = node
        else:
            node = self.existing_nodes[key]

        return node


    def _build_adj_list(self):
        for pair in self.forest:
            # Build individual nodes
            node1 = self._build_individual_node(pair[0])
            node2 = self._build_individual_node(pair[1])

            # Add edges
            node1.neighbors.add(node2)
            node2.neighbors.add(node1)

            # Add nodes to adj list
            self.adjList.add(node1)
            self.adjList.add(node2)

#####################################
forest = [[0, 1], [0, 2], [3, 4], [13, 15], [12, 4]]
g = Graph(forest)
g.tree_number
```

```
## 3
```

Question 2

You are given a tree T with directed tree edges (each edge points from the parent to the child). How can you topologically sort the vertices of T without doing DFS? Analyze the running time of your algorithm.

Solution:

The topoloical sort can be implemented using Kanh's algorithm instead of DFS as follows. This implementation involves constructing an adjacency list that takes O(N + E), and performing topoloical sort using the Kanh's algorithm. The Kanh's algorithm starts by placing a node without any in_neighbors in a stack, and iteratively clears its next level neighbor's in_neighbors connecting to it. If all in_neighbors of this next-level neighbor

are empty at this time, it is put in the stack. Repeat until all nodes and their out neighbors are explored The entire process takes also O (N + E) time. In total, this implementation thus takes O(N + E) time.

```
class Node(object):
    def __init__(self, key):
        self.key = key
        self.in_neighbors = set()
        self.out_neighbors = set()


class Graph(object):
    def __init__(self, forest):
        self.forest = forest
        self.existing_nodes = dict()
        self.adjList = set()
        self._build_adj_list()


    @property
    def graph(self):
        return [(node.key, [nbs.key for nbs in node.in_neighbors]) for node in self.adjList]


    def topological_sort_Kanh(self):
        # Topological sort implemented using Kanh's algorithm
        stack = [node for node in self.adjList if not node.in_neighbors]
        sorted = []

        while stack:
            node = stack.pop()
            sorted.append(node.key)

            for n in node.out_neighbors:
                n.in_neighbors.remove(node)
                if not n.in_neighbors:
                    stack.append(n)

        return sorted



    def _build_individual_node(self, key):

        if key not in self.existing_nodes.keys():
            node = Node(key)
            self.existing_nodes[key] = node
        else:
            node = self.existing_nodes[key]

        return node


    def _build_adj_list(self):
        for pair in self.forest:
```

```
            # Build individual nodes
            node1 = self._build_individual_node(pair[0])
            node2 = self._build_individual_node(pair[1])

            # Add edges
            node1.out_neighbors.add(node2)
            node2.in_neighbors.add(node1)

            # Add nodes to adj list
            self.adjList.add(node1)
            self.adjList.add(node2)

##################################
forest = [[0, 1], [0, 2], [2, 3], [1, 3]]
g = Graph(forest)
g.topological_sort_Kanh()
```

```
## [0, 1, 2, 3]
```

Question 3

Given a directed graph G, give an algorithm that tests if G is a DAG. Analyze the running time of your algorithm.

Solution:

If during the DFS traversal of a tree a node under exploration is encountered, this means that this node has been visted a second time before its successors have been visited, or the graph has a back edge forming a cycle. It can thus be inferred whether a given graph is a directed acyclic graph. The entire implementation traverses the graph in a DFS manner, which requires initial access to each node, taking O(N) time, and all neighbors taking O(E) time, totaling O(N + E) time. In addition, the implementation also takes O(N + E) time to build the adjacency list. Adding the two together results in a total of O(N + E) time.

```
class Node(object):
    def __init__(self, key):
        self.key = key
        self.in_neighbors = set()
        self.out_neighbors = set()
        self.time = list()
        self.visited = False
        self.completed = False
        self.under_exploration = False
        self.status = None

class Graph(object):
    def __init__(self, forest):
        self.forest = forest
        self.existing_nodes = dict()
        self.visited = set()
        self.time = 0
        self.adjList = set()
        self.acyclic = None
        self._build_adj_list()


    @property
```

```
    def graph(self):
        return [(node.key, [nbs.key for nbs in node.in_neighbors]) for node in self.adjList]


    def DFS_rec(self, node):
        self.time += 1

        node.visited = True
        node.under_exploration = True
        node.time.append(self.time)

        for nbs in node.out_neighbors:
            if nbs.under_exploration:
                self.acyclic = False
            elif not nbs.visited:
                self.DFS_rec(nbs)


        node.under_exploration = False
        node.completed = True
        self.time += 1
        node.time.append(self.time)




    def is_DAG(self):
        self.acyclic = True
        for node in self.adjList:
            if not node.visited:
                self.DFS_rec(node)

        if self.acyclic:
            return 'This is a DAG'
        else:
            return 'This is not a DAG.'

    def _build_individual_node(self, key):

        if key not in self.existing_nodes.keys():
            node = Node(key)
            self.existing_nodes[key] = node
        else:
            node = self.existing_nodes[key]

        return node


    def _build_adj_list(self):
        for pair in self.forest:
            # Build individual nodes
            node1 = self._build_individual_node(pair[0])
            node2 = self._build_individual_node(pair[1])
```

```
            # Add edges
            node1.out_neighbors.add(node2)
            node2.in_neighbors.add(node1)

            # Add nodes to adj list
            self.adjList.add(node1)
            self.adjList.add(node2)


#####################################################
forest = [[1, 2], [1, 3], [2, 3], [1, 4],
          [4, 5], [6, 4], [5, 6]]
g = Graph(forest)
g.is_DAG()
```

```
## 'This is not a DAG.'
```