Name: Le Thanh Cong

Student ID: 20245998

Programming – TP3

Exercise 1:

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center). For example, this binary tree [1,2,2,3,4,4,3] is symmetric.

Idea: Use recursion technique to traversal through the binary tree. Check the similarities between the left of root1 and right of root2 and vice versa.

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

def checkSymmetricTree(root1: Node, root2: Node):
    # Purpose of 2 check variables is to break the recursion as soon as possible
    check1 = True
    check2 = True
    # If value of the 2 root is not equal then return False
    if root1.val != root2.val:
        return False

    # If the left of root1 and right of root2, one is None and one is not the
return False
    if (root1.left is None and root2.right is not None) or (root1.left is not
None and root2.right is None):
        return False
    # Same for right of root 1 and left of root2
    if (root1.right is None and root2.left is not None) or (root1.right is not
None and root2.left is None):
        return False

    if root1.left is not None and root2.right is not None:
        check1 = checkSymmetricTree(root1.left, root2.right)
    if root1.right is not None and root2.left is not None:
        check2 = checkSymmetricTree(root1.right, root2.left)

    return check1 and check2
```

```python
def symmetric_tree(root):
    if root.left is None:
        # if this tree contain only the root and no children then return True
        if root.right is None:
            return True
        else:
            return False
    if root.right is None:
        return False

    return checkSymmetricTree(root.left, root.right)

n1 = Node(1)
n2 = Node(2)
n3 = Node(2)
n4 = Node(3)
n5 = Node(4)
n6 = Node(4)
n7 = Node(3)

n1.left = n2
n1.right = n3
n2.left = n4
n2.right = n5
n3.left = n6
n3.right = n7

print(symmetric_tree(n1))
```

Exercise 2:

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Idea: Use the Inorder traversal technique and store the value in the list, if the list is not in accending order, then this is not a binary search tree.

```python
# Idea: we use the technique Inorder traversal to get a sequence.
# If this is a valid binary search tree then the obtained sequence must be in
accendding order
# Return False and break if the next value is smaller than the last value in the
list (to safe time)
class Node:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def is_valid_BST(root: Node, pre_traversal_list: list):
    # Purpose of check1 and check2 is to stop the recursion as soon as it meets a
value
    # that doesn't fit in the accending list, dont need to check further, return
False

    # Check the left child
    if root.left is not None:
        check1 = is_valid_BST(root.left, pre_traversal_list)
        if check1 == False:
            return False

    # if the value of current node is not greater than the last value in
pre_traversal_list,
    # return False
    if len(pre_traversal_list) > 0:
        if root.val <= pre_traversal_list[-1]:
            return False
    pre_traversal_list.append(root.val)

    # check the right child
    if root.right is not None:
        check2 = is_valid_BST(root.right, pre_traversal_list)
        if check2 == False:
            return False

def check_valid_BST(root: Node):
    if root is None:
        return False

    if is_valid_BST(root, []) == False:
        return False
    return True
```

```
n1 = Node(4)
n2 = Node(2)
n3 = Node(6)
n4 = Node(1)
n5 = Node(3)
n6 = Node(5)
n7 = Node(7)

n1.left = n2
n1.right = n3
n2.left = n4
n2.right = n5
n3.left = n6
n3.right = n7

print(check_valid_BST(n1))
```

Exercise 3:

There are N network nodes, labeled 1 to N.

Given times, a list of travel times as directed edges times[i] = (u, v, w), where u is the source node, v is the target node, and w is the time it takes for a signal to travel from source to target.

Now, we send a signal from a certain node K. How long will it take for all nodes to receive the signal? If it is impossible, return -1.

```
# Idea: Dijkstra algorithm, we find the shortest path from node K to all other
Node.
# Return the largest distance, if it is inf (cant reach some nodes) then return -
1
# First we need to create a relative dictionary to store all the route of one
node its neighbor
# This is one side route

# Use PriorityQueue to reduce time to find the smallest distance
from queue import PriorityQueue

# Create the neighbor dictionary from the input times
def create_neighbor_dict(times, N):
    neighbor_dict = {}
    for i in range(N):
        neighbor_dict[i] = []
```

```python
    for [source, dest, weigh] in times:
        neighbor_dict[source-1].append([dest-1, weigh])
    return neighbor_dict

def find_signal_tranmission_time(times, N, source):
    # Transform to 0-index
    source -= 1

    neighbor_dict = create_neighbor_dict(times, N)

    visited_nodes = set()

    # Initialize all distance as infinite except the source
    distances = {node: float('inf') for node in range(N)}
    distances[source] = 0

    # push source to priority queue with distance = 0
    pri_queue = PriorityQueue()
    pri_queue.put([0, source])

    while pri_queue.qsize():
        distance, node = pri_queue.get()

        if node in visited_nodes:
            continue

        visited_nodes.add(node)

        for neighbor, distance_neighbor in neighbor_dict[node]:
            if neighbor in visited_nodes:
                continue
            tmp = distance + distance_neighbor
            if (tmp < distances[neighbor]):
                distances[neighbor] = tmp
                pri_queue.put([distances[neighbor], neighbor])

    # After getting the shortest distances from source to all other node
    # Find the largest, inf means this node can't be reach from source
    _, max_distance = max(distances.items())
    if (max_distance == float('inf')):
        return -1
    return max_distance

times = [[2,1,1],[2,3,1],[3,4,1]]
N = 4
```

```
K = 2

print(find_signal_tranmission_time(times, N, K))
```

Exercise 4:

```
# Idea: find the MST of the full graph MST1,
# remove the one edge at a time, then find the MST,
# Which one make the MST total weight increase and appear in MST1, that one is
the critical one
# If we remove an edge and still find the same weight as MST1, that's another MST
for the full graph

class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n)]
        self.rank = [1] * n

    def find(self,  v1):
        while v1 != self.parent[v1]:
            self.parent[v1] = self.parent[self.parent[v1]]
            v1 = self.parent[v1]
        return v1

    def union(self, v1, v2):
        p1, p2 = self.find(v1), self.find(v2)
        if p1 == p2:
            return False
        if self.rank[p1] > self.rank[p2]:
            self.parent[p2] = p1
            self.rank[p1] += self.rank[p2]
        else:
            self.parent[p1] = p2
            self.rank[p2] += self.rank[p1]
        return True

def kruskal_mst(edges, n):
    uf = UnionFind(n)
    mst_weight = 0
    result = []
    for a, b, weight, i in edges:
        if uf.union(a, b):
            result.append([a, b, weight, i])
            mst_weight += weight
    # if all node is not connected
    if max(uf.rank) != n:
        return [], -1
    return result, mst_weight

def find_critical_and_pesudo_critical_edges(edges, n):
    # keep the original index of the edges
    for i, edge in enumerate(edges):
        edge.append(i)

    # sort edges by the weight
    edges.sort(key=lambda e: e[2])

    # find the mst with the full graph
    list_edges, min_mst = kruskal_mst(edges, n)

    # for all edge in list_edge, try remove them and calculate the mst again
    # if it cannot connect to all node or new MST > min_mst then it is a critical
edge
    # if it can create a new MST = min_mst then it is a pseudo_critical edge
    # add edge in new found mst in the list_edge

    criticals = []
    pseudo_critical = []

    observed_edge = [0 for i in range(len(edges))]
    while(len(list_edges)>0):
        edge = list_edges.pop()
        # mark as observed
        observed_edge[edge[3]] = 1
        # create a new list except this edge
        tmp_edges = []
        for element in edges:
            if element[3] != edge[3]:
                tmp_edges.append(element)

        tmp_list_edge, tmp_min_mst = kruskal_mst(tmp_edges, n)
        if tmp_min_mst == -1 or tmp_min_mst > min_mst:
            criticals.append(edge[3])

        elif tmp_min_mst == min_mst:
            pseudo_critical.append(edge[3])
            # Join tmp_list_edge and list_edge, just add the edge that doesnt
appear on observed list
```

```
            set1 = set(tuple(x) for x in list_edges)
            tmp_list = [x for x in tmp_list_edge if observed_edge[x[3]] == 0]
            set2 = set(tuple(x) for x in tmp_list)
            list_edges = list(set1.union(set2))

    return criticals, pseudo_critical

edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]
n = 5
print(find_critical_and_pesudo_critical_edges(edges, n))
```