

Name: Huayue Hua
USC ID: 9817961224
Email: huayuehu@usc.edu

Lab10

Mandatory Part

HEADS UP: All files in the mandatory part of this lab should be in the SAME folder

Commands to be run for each part:

Part I: `python3 HuayueHua_PI_graph_generator.py`

Part II: `python3 HuayueHua_PII_top_execute.py`

Part III: `python3 HuayueHua_PIII_clique_checker.py`

Part I Graph Valid Tree Checker

There are two files for this part:

`HuayueHua_PI_tree_checker.py`

`HuayueHua_PI_graph_generator.py`

(1) To run this part:

Simply run the following command:

`python3 HuayueHua_PI_graph_generator.py`

Result like this should be displayed in the terminal interface:

```
Part I — insane@niuuhulujiahua:~/lab10/Part I — zsh — 85x47
Last login: Thu Apr 16 20:30:48 on ttys000
Part I git:(master) x python3 HuayueHua_PI_graph_generator.py
----- case 1: 0 node, 0 edge -----
Graph 1
This graph is not a tree. There is no node and edge.
----- case 2: fully connected (with 5 nodes) -----
Graph 2
This graph is not a tree. A loop in this graph: 0 1 2 0
----- case 3: 6 nodes, 2 edges -----
Graph 3
This graph is not a tree. Node 2 is not in connected to the rest of graph.
----- case 4: other scenarios -----
Graph 4
This graph is not a tree. There is no node and edge.
Graph 5
This graph is not a tree. Node 0 is not in connected to the rest of graph.
Graph 6
This graph is not a tree. Node 0 is not in connected to the rest of graph.
Graph 7
This graph is not a tree. Node 0 is not in connected to the rest of graph.
Graph 8
This graph is not a tree. A loop in this graph: 0 1 2 0
Graph 9
This graph is not a tree. Node 0 is not in connected to the rest of graph.
Graph 10
This graph is not a tree. A loop in this graph: 0 2 7 6 0
Graph 11
This graph is not a tree. Node 0 is not in connected to the rest of graph.
Graph 12
This graph is not a tree. A loop in this graph: 0 2 3 1 2
Graph 13
This graph is not a tree. A loop in this graph: 0 1 2 4 3 5 2
Graph 14
This graph is a tree, i.e., it does not have a loop.
Graph 15
This graph is not a tree. A loop in this graph: 0 3 4 1 2 3
Graph 16
This graph is not a tree. A loop in this graph: 0 4 2 1 3 4
Graph 17
This graph is not a tree. A loop in this graph: 0 1 5 4 3 0
Graph 18
This graph is not a tree. A loop in this graph: 0 2 3 1 0
Graph 19
This graph is a tree, i.e., it does not have a loop.
Graph 20
This graph is not a tree. A loop in this graph: 0 1 3 0
Part I git:(master) x
```

- (2) In **HuayueHua_PI_graph_generator.py**, I import **HuayueHua_PI_tree_checker** as **tc**. Every time I generate a new graph, **input.txt** would be rewrite, then **tc.main()** would be called to check whether this graph is a tree or not.
- (3) To write cycle checker function in **HuayueHua_PI_tree_checker.py**, I take <https://www.geeksforgeeks.org/check-given-graph-tree/> as a reference.

The idea for **tree checker** is to read **node_num**, **edge_num** and edges connection information from the **input.txt**, then I check if the graph has cycle. If there is a cycle, print out corresponding information. If there is no cycle, check the connectivity to see if the graph is a tree.

Pseudocode for **cycle_checker()** function:

```
1. def cycle_checker(node, parent, visited):
2.     visited[node] = True
3.
4.     for i in graph[node]:
5.         if i not visited yet:
6.             recursively check other node connected using DFS fashion:
7.                 if detect cycle => return True
8.         elif node i is visited and node i is not the parent node => cycle detected
9.             return True
10.
11.     return False
```

Pseudocode for **connectivity_checker()** function:

```
1. def connectivity_checker(graph, node_num):
2.     initially set visited array to False
3.
4.     for each node:
5.         for i in graph[node]:
6.             set visited[i] to True
7.
8.     for each node:
9.         check if visited[node] is False,:
10.            return False if there is a False
11.
12.     return True
```

Part II Miscellaneous

There are two files for this part:

HuayueHua_PII_top_execute.py

HuayueHua_PII.cpp

- (1) To run this part:

Simply run the following command:

python3 HuayueHua_PII_top_execute.py

Result like this should be displayed in the terminal interface:

```
lab10 — insane@niuholuediannao — ..595/lab/lab10 — -zsh — 94x47
[→ lab10 git:(master) * python3 HuayueHua_PII_top_execute.py
This graph is a tree, i.e., it does not have a loop.
The maximum degree is 3 for node number 0.
There is at least one bridge in this graph.
After topological sort: 3 0 1 2 4
[→ lab10 git:(master) * █
```

- (2) In **HuayueHua_PII.cpp**, I do the following three things:

- Calculate the maximum degree
- Check whether there is a bridge in the graph
- Perform topological sorting

To calculate the maximum degree, I simply find the degree of every node in the graph and return the maximum one.

To write bridge checker part, I take <https://www.geeksforgeeks.org/bridge-in-a-graph/> as a reference. In `bridge_checker()` function, `visited` array is to track whether a node has been visited or not. `Visited_time` and `less_time` stores the discovery time of the visited node and less time that the visited tree can be visited by other path if exists. `Parent` stores the parent node of current node. We initial `visited` array to false and all `parent` to -1. Iterate all the nodes, if not visited yet, perform DFS recursively to update `visited_time` and `less_time` information to decide whether there is a bridge in the graph or not.

To write topo sort part, I take <https://songlee24.github.io/2015/05/07/topological-sorting/> as a reference. When adding edges in the previous procedure, I use `in_degree` to check the direction of each edge, so if a node with `in_degree` equals to 0, this node is a root node. Push the root node to queue, for each node in the queue, print out the current node in order and also push all the adjacent nodes to the current node to queue. In this way, node will be sorted.

- (3) In **HuayueHua_PII_top_execute.py**, I firstly call tree checker in part I to check whether the graph represented by `input.txt` is a tree or not. If it is a tree, then using `subprocess.call()` to help run **HuayueHua_PII.cpp**.

Part III Find the maximum clique of a graph

There are two files for this part:

`HuayueHua_PIII_clique_checker.py`

`HuayueHua_PIII_max_clique_calc.py`

- (1) To run this part:

Simply run the following command:

`python3 HuayueHua_PIII_clique_checker.py`

Result like this should be displayed in the terminal interface:

```
lab10 — insane@niuholuediannao — ..595/lab/lab10 — -zsh — 94x47
→ lab10 git:(master) * python3 HuayueHua_PIII_clique_checker.py
----- case 1: 0 node, 0 edge -----
Graph 1
No clique found.
----- case 2: fully connected (with 5 nodes) -----
Graph 2
Clique number (5): 0 1 2 3 4
----- case 3: 6 nodes, 2 edges -----
Graph 3
No clique found.
----- case 4: other scenarios -----
Graph 4
No clique found.
Graph 5
No clique found.
Graph 6
No clique found.
Graph 7
No clique found.
Graph 8
No clique found.
Graph 9
No clique found.
Graph 10
No clique found.
Graph 11
No clique found.
Graph 12
No clique found.
Graph 13
No clique found.
Graph 14
Clique number (3): 0 1 2
Graph 15
Clique number (4): 0 1 3 4
Graph 16
No clique found.
Graph 17
No clique found.
Graph 18
Clique number (3): 0 1 2
Graph 19
No clique found.
Graph 20
No clique found.
→ lab10 git:(master) *
```

- (2) In **HuayueHua_PIII_max_clique_calc.py**, for each edge, when an edge is added to the present nodes list, check if the present nodes list forms a clique, keeping adding nodes until the list does not form a clique. In **HuayueHua_PIII_clique_checker.py**, it's the same as the part I graph generator, except that it imports **HuayueHua_PIII_max_clique_calc**.

To write this part, I take <https://www.geeksforgeeks.org/maximal-clique-problem-recursive-solution/> as a reference