author = '8175858, Braun'

Analysis:

The program is developed according to the EVA design pattern. Input and output are done in the console.

Output: Output is done in the console.

Description of the program:

The program does not require any further libraries other than random, it needs the ue05_modul.py and spielkarten.py module. The program is written and tested in Python 3.9.6. A Python interpreter must be installed. The program can be started with the command "python3 ue06.py" in the terminal. The Output is done in the console. No bugs are known.

The programm uses functions from ue06_functions.py which uses graph.py to depict the graph.

get_graph() returns a object of the Graph class from graph.py. It uses ast.literal_eval() to convert the string from the file to a dictionary. This method is from the ast (Abstract Syntax Trees) module. The dictionary is then inserted into the items attribute of the Graph class. The graph is then returned.

is_tree() returns a bool. It uses the is_vertex_isolted() method to check if the graph is connected. Then it creates a undirected graph of the given directed graph. The method is using copy.deepcopy(), so that the original graph is not changed later on. It then checks if the number of edges is equal to the number of vertices minus 1. Then it checks if the graph is cyclic. This works using DFS. We go through every vertex and its neighbours. The idea is to use a seperate visited dict with the vertices as keys and initially False as value for all of them. First we initialize the current vertex as True and then go through every neighbour and its neighbours recursively. If we visit a node, we set its value to True. If we visit a node that is already True, we have a cycle except the former vertex, the one from which we have visited the current edge, is a parent. If we have a cycle, we return True, else False. This procceedure is done for every vertex. This is done in the is_cyclic() and the is_cyclic_utillity() method. if is_cyclic() returns True, we have a cycle and return False, else True.

get_leafs() goes through every vertex and checks if the length of is edges list is 0, meaning if it has no children. Since we call the function if we know we have a tree, if it has no children, it is a leaf and is added to the leafs list. The leafs set is then returned.

has_root() determines to find a root if given using in_degree_counters, which are given initially to every vertex. We make a new dict using the originial dict

and just add counters. The in-degree of a graph is the number of edges that point to a vertex. Since the root (or roots) is the only vertex with a in-degree value of 0 (no edges pointing to it), we can just check if the in-degree of a vertex is 0. If it is, we add it to the roots list. If the length of the roots list is 0, we return False, else True. To initialize the in-degree counters, we go through every vertex and its edges and add 1 to the in-degree counter of the every node in its edges list.

How to use:

The program can be started with the command "python3 ue06.py" in the terminal. The programm then asks for a graph in adjacency list format. The graph must be given in the following format:

```
{1: [2, 3], 2: [4, 5], 3: [6, 7], 4: [], 5: [], 6: [], 7: []}
```