**Link for Code:**

[**https://repl.it/@Sk\_adibadib/miniproject**](https://repl.it/@Sk_adibadib/miniproject)

(It takes some times to run the code)

**Task 1:**

1) For directed type data set: I chose an ‘Epinions’ Dataset <https://snap.stanford.edu/data/soc-Epinions1.html>

It has 75879 Nodes and 508837 Edges. This dataset represents a who-trust-whom online social network of a general consumer review site. Members of the site can decide whether to ''trust'' each other. All the trust relationships interact and form the Web of Trust which is then combined with review ratings to determine which reviews are shown to the user. Each line is an edge A --> B, means user A trust user B.

2) For undirected type data set: I chose Gemsec Deezer Dataset <https://snap.stanford.edu/data/gemsec-Deezer.html>

These datasets represent friendship networks of users from 3 European countries: Romania, Croatia, Hungary. Nodes represent the users and edges are the mutual friendships. The csv files contain the edges: meaning that users A and B are mutual friends, also there exists a json files which contains the genre preferences of users -- each key is a user id, the genres loved are given as lists.

**Task 2:**

For a big graph it takes too much time for repl.it to read the file and sometimes it freezes. That’s why I tried to manage with a smaller dataset.

Dataset files has a different format of representing the graph. Some has tab delimiter to separate two nodes, in other space or semicolon are used, sometimes there is a commentary at the beginning of the file. I tried to handle all this case and split the nodes depending on delimiters and skip comments.

**Tasks 4-5:**

1. During recursively calling Explore(G) function got error “Runtime Error: maximum recursion depth 992 exceeded”. The problem was solved by increasing the standard maximum recursion depth value

import sys

sys.setrecursionlimit(100000)

1. The execution time of DFS is too big with graph representation as the list of edges.

Changed the representation of the graph from task 2. Previously I just read the file and stored the graph as list of edges. Such graph representation takes too much time to handle each node in DFS() and Explore() functions.

It’s better to store the graph as the adjacency list. I decided to use the dictionary data structure, where dictionary’s key is a node U and its value is a list of nodes outgoing from the node U.

E.g. for graph

      A->B

      A->C

      A->D

      B->C

The dictionary representation is

G={A:[B,C,D], B:[C]}

1. We can implement the idea of connected components even to directed graph. Everything what is needed is to consider the directed graph as undirected one and find connected components for that.
2. Sometimes the DFS algorithm is freeze for undirected graph, the reason in a big number of nodes. One way for solving the problem is to reduce the file by half.
3. To find the strongly connected components the hardest thing is to Handle

vertices in decreasing order of their post numbers in the DFS loop.

The solution is to use the standard python’s function sorted() with lambda function as key. The solution to sort my graph which is a dictionary data structure were founded in Web.

<https://www.saltycrane.com/blog/2007/09/how-to-sort-python-dictionary-by-keys/>

<https://stackoverflow.com/questions/613183/how-do-i-sort-a-dictionary-by-value>

**Results:**

For Task 4. “Write the code that will find and report these smaller components of the network”. the implemented algorithm gives on the directed graph Epinions1 (<https://snap.stanford.edu/data/soc-Epinions1.html>), gives only 2 ‘fake’ nodes (74845, 71749) it’s true by checking this node in dataset.txt file. I figured that there is only 1 edge 74845, 71749 and there are no more input/output nodes connected with these ones.

For Task 5 the implemented algorithm gives a lot of strongly connected components and it’s hard to understand if these components are strongly connected. To make sure that the function works in a proper way, I tested it on a small graph and the results are right.

**Algorithm’s description:**

For this project, it is important to think about the way how to represent the graph in python.

I took the adjacency list graph representation as the way too store a graph, because matrix representation takes lot of time and memory. In python I realized it as a dictionary where the key of dictionary is a vertex and value of this key is a list of outgoing vertices.

V1 🡪 [V2, V5, V7]

means that V1🡪V2, V1 🡪 V5, V1 🡪 V7.

Such dictionary is a convenient way to insert, delete, search any vertex in the graph, python ‘s library provides plenty of methods to operate with dictionary.

The central place of the algorithm is the modified depth first search algorithm DFS(G), taking as its argument a graph G. During DFS we collect some data about the graph G like number of connected components in the graph, the node’s connected components id, node’s pre-visit and post-visit time.

We store these data as a dictionary graph\_stat, where dict.key is a node and the value are a list of 3 elements.

node 🡪[a,b,c]

where a – nodes’s connected component id,

b – node’s previsit time

c – postvisit time

The DFS(G) goes through the graph and collects the above data for each node in the graph. Having such info about each node we can find out many interesting things about the graph G.

For instance, we can find all connected components of the graph G. Moreover, in task 4 we are interested in detecting nodes which are either isolated or creates a small group not connected to the main tree. We will call such nodes fake. The below pseudocode does this task

def FindSmallerComponents(G, is\_directed, min\_node\_numb):

If G is directed then make G undirected

Run DFS(G) #to get data of each node

For each connected\_component\_id: DO

count the number of nodes its included

The connected\_comonents\_id’s with small number of nodes is a fake

This is just a general idea, in the code the user can determine the parameter min\_node\_numb which checks if the number of nodes in some connected component is less then this parameter then it’s a fake one. The hardest part of this algorithm is to find how many nodes does the connected component contain? It’s realized in the function

def CC\_nodes\_number(graph\_stats):

nb\_cc\_arr[nb\_cc] = 0

for cc\_id in graph\_stats

nb\_cc\_arr[cc\_id] ++

It takes the graph\_stats as the argument which is the dictionary { node 🡪 [cc\_id, previsit, postvisit]} which we obtain by running DFS.

So, knowing the total number of connected components which is a global value(initialized by DFS), we create a list nb\_cc\_arr of length nb\_cc with zeros.

The i-th position of the nb\_cc\_arr is a cc\_id, while nb\_cc\_arr[i] stores the number of nodes in i-th connected component. (connected components id start from 0).

The function is just looping through the graph\_stats dictionary takes the node’s connected component id(cc\_id) and increase the nb\_cc\_arr[cc\_id].

Also using our DFS algorithm we can determine the strongly connected components.

def FindStronglyConectedComponents(G):

1. Run DFS(G)

2. Construct reverse graph G\_R

3. Run DFS on G\_R, but process the vertices in decreasing order

of their post numbers from step 1

4. Strongly connected components are nodes with the same

connected component value