Social Network Analysis – Homework 1
George Choumos

This document just includes a really brief overview of how the 2 problems were solved. In fact, the scripts that are submitted are sufficiently commented and a read through them will be enough to give you a clear picture of the implementation details.

Problem 1

Implementation of the functions:

- ***has_all_vertices_even***
  Iterates through the graph nodes and checks if their degree is divisible by 2. If it finds one that has odd degree it return False. Otherwise, if all nodes are checked and none is odd, it returns True.

- ***get_odd_vertices***
  Iterates through the graph nodes and add the odd-degree ones in a set which it eventually returns. If none is found, then it just returns an empty set.

- ***is_connected***
  This is just a wrapper function that only calls snap's corresponding function and then returns the value that comes back.

- ***has_euler_path***
  Decides whether or not the given graph has an euler path. Firstly it checks if the graph is connected. Then it calls *get_odd_vertices* in order to get the nodes that have odd degree. If the graph is connected and the odd vertices are exactly 2 then the graph does have an Euler path. Otherwise, if any of the above checks fail, it has not.

- ***has_euler_circuit***
  Similar to the previous one except for the fact that after the connected check, it then checks whether or not the nodes (all of them) have even degree.

Test cases preparation:

- ***test_has_euler_path_but_not_circuit***:
  A graph is manually created (not through a generator) with 6 nodes and 9 edges. 2 of the nodes have odd degree.

- ***test_does_not_have_euler_path***::
  A graph is manually created again with 6 nodes and 6 edges. 4 of the nodes have odd degree.

- ***test_has_euler_circuit***:
  Since we want a sufficiently large graph here, we used a generator. A circular graph seemed to be the easiest pick in order to satisfy the even degree prerequisite for all nodes.

- ***test_does_not_have_euler_circuit***:
  We created a smaller version of the previous graph (as we don't have size limiations here) and we just deleted an edge!

Problem 2

First part

The script is implemented so that it gets the number of nodes and edges from the user upon execution. There exist default values as well, 50 for the number of nodes and 10 for the number of edges.

Input checks are provided so that if the user enters no value or something that is not a number or not in the acceptable range, the defaults are being used instead.

The graph in the first part is generated using snap's *GenPrefAttach* generator.

Max outdegree is calculated and printed along with the node id that holds it In fact, as I mention in the comments, it is quite possible that the maximum outdegree is going to be shared by many nodes. We could either print all of the tied outdegree nodes or just keep and print 1 of them. I used the second approach as it seems sufficient for the purpose of this exercise.

Then the PageRank max value is calculated in the same manner as with the outdegree. The probability of a tie in this case is quite smaller though.

The next step is to measure the time that is required for the Girvan-Newman community detection algorithm. I am using the time module to achieve this. Right after, similar functionality is triggered in order to measure the time required for the Clauser-Newman-Moore community detection algorithm. Printing the communities is commented out as I don't think it has much to offer and because of the fact that we are actually interested in the execution time.

Second Part

Here is what happens now. We are starting with 50 nodes and 15 as the edge parameter. The edge parameter remains stable but the nodes are incremented. We actually add 10% of the nodes for each iteration. This means that for the first iterations the nodes will be 50, 55, 60, 66, 72, 79, 86 … and so on.

There are 2 cases for termination of the execution. We either receive a memory error or we are at a time point which is more than 10 minutes from the moment that the script's execution started.

In each iteration, you will see output for the number of the nodes that are in the generated graph, as well as the number of the edges parameter. When the execution of the iteration ends, you will also see in the output how much time it took for this iteration to finish.

I have added a handler for the MemoryError exception so that the script doesn't fail, but terminate grraciously instead.