README

Problem 5

Introduction

The problem is an example of the **vertex cover problem** on a graph. So ,for simplicity, in this and in all the documents and files related to the problem number 5, we will refer to the vertex cover problem and not to the bacefook problem.

The algorithm

The algorithm that we have designed is greedy and consists of the following steps:

- for each vertex of the graph that has not been already added to the vertex cover, call it v, consider all its neighbour vertices that are not in the vertex cover.
- for v and for each neighbour, call it u, compute the degree. The degree is the score assigned to each vertex and it is the number of adjacent vertices not in the vertex cover.
- for each couple (v, u), add to the vertex cover the vertex with the highest degree. If v is added to the vertex cover, stop the iteration for the remaining couples (all the couples (v, u) are satisfied).

Complexity

The complexity of this algorithm is O(n + m) where n is the number of vertices and m is the number of edges.

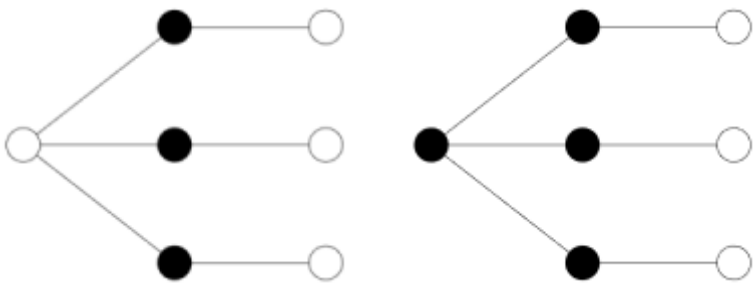
Performance evaluation

We tested the function using 100 graphs with 100 nodes. Each graph is randomly generated (with the probability of taking an edge of 0.5).
The average performance achieved is 92 nodes inserted in the vertex cover.
The average time is 3.5 ms.

Does the algorithm return an optimal solution?

The algorithm does not return always an optimal solution.

For example the optimal solution for this graph is the one displayed in the first image.
Our algorith could provide the solution displayed in the second image, that is not optimal.



Approximation ratio

We are not able to find a precise approximation ratio for our algorithm. We notice that it's similar to an another algorithm, that we implemented and tested too. This algorithm, at each iteration, insert in the vertex cover the vertex with the higest degree (number of adjacent vertices not in the vertex cover). If a vertex has degree zero (is isolated), don't insert it. It is demonstrated that this algorithm, in the worst case, is log(n) worse respects the optimal one. So our algorithm is probably log(n) approximatde.
We don't use this algorithm because is very slow compared to our algorithm (that is linear) and the performance is more or less the same.

This is not the algorithm with the best approximation ratio.
The following simple algorithm (that we implemented and tested too) achieve a 2 approximation ratio:
The algorithm simply takes ad edge and add each vertex of this edge to the vertex cover (removing or setting as removed these vertices and all the incident edges of these vertices). So, in this way, in the worst case, the solution will contain twice the number of vertices respects the optimal solution.
We decide to not use this algorithm because in practice, using the graphs randomly generated, it performs worse than our algorithm.

How to test the script

The function is located in **vertex_cover.py**. In order to test the function our implementation of the graph has to be used (the implementation is located in graph.py). Our implementation has a method that dumps a graphcal representation of the graph. If you want to use this method, you have to import network and matplotlib.pyplot, otherwise you don't need to import any modules for testing the function. When building a new graph, you have not to pass a value, for the element of the vertex and for the element of the edge, different from the default one. In fact, the algorithm uses these values during the computation and assumes that at the begging they are the default ones, assigned as default in the graph.

There are also two additional files that we use for testing the function. In **test_functions.py** there are a function for evaluating the correctness of the vertex cover, a function for building a random graph and a function for evaluating the performance of the algorithm. If you want to use the last function, you need Python >= 3.7.
In test.py the functiion for evaluating the performance is used.