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).

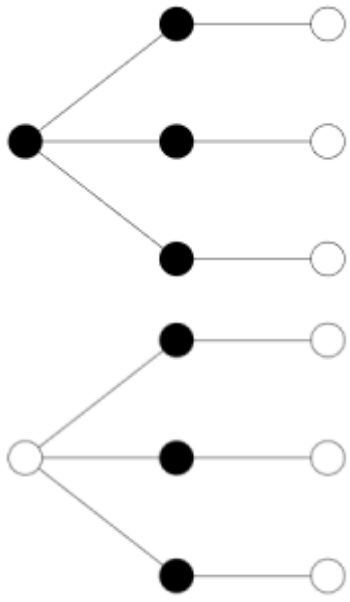
Complexity

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

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 in the first image.
Our algorithm could provide the solution of 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) approximate.
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

Our algorithm is a function called vertex_cover in the **vertex_cover.py** module, it takes in input a graph istance of the class Graph present in the module graph.py module. It evaluates the vertex cover on the graph and labels the vertices as **True** or **False**, if a vertex is True it is in the vertex cover otherwise not.
In order to test the function our implementation of the graph has to be used. Our implementation has a method that dumps a graphcal representation of the graph. If you want to use this method, you have to import networkx 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.

The test.py

In this module, we just call a function that we made in order to evaluate the performances of different solutions the **evaluate_performances** present in the **test_functions.py** module.
If you want to use it,you need Python >= 3.7.

This function takes as input:

1. An array of functions, that contains the functions for which the performances must be evaluated.
2. The number of graphs on which the performances must be evaluated (They are made of 100 vertices for default).
3. The probability that a vertex has and edge linking it with any other vertices.

This function creates as random graphs as specified in the second parameter using the function **build_random_graph** in the **test_functions.py** that takes in input the number of vertices and the probability specified in the third parameter.
Then each function is runned on each graph, calculating the time it takes and the value of the vertex

cover that it returns.
For each graph the correcteness of the result is tested using the **is_vertex_cover_correct**.
At the end a mean of this values is calculated and returned.

If you dont want to use this function, all you have to do is creating a graph instance of the Graph class, with the default values of elements for vertices and edges, and pass it to the vertex_cover.