First AA Assignment - Problem #18 - Minimum edge cover

Artur Romão, 98470

Resumo **- Este trabalho tem como objetivo explorar dois algoritmos de pesquisa para resolver um problema de grafos, o "minimum edge cover" ou, em português, menor cobertura de arestas. Um edge cover é um conjunto de arestas tal que cada vértice de um dado grafo seja incidente a pelo menos uma aresta do conjunto. Um min edge cover é um edge cover de menor tamanho possível, isto é, com menor número de arestas. De maneira a solucionar este problema, foram seguidas duas abordagens: pesquisa exaustiva e pesquisa gulosa. Neste relatório será explicado como foi feito o ataque ao problema, a construção dos dois algoritmos e uma comparação entre os dois.**

Abstract **- This work aims to explore two search algorithms to solve a graph coverage problem: "min edge cover". An edge cover is a set of edges such that each vertex of a given graph is incident to at least one edge of the set. A min edge cover is an edge cover of the smallest possible size, that is, with the least number of edges. In order to solve this problem, two approaches were followed: exhaustive search and greedy search. In this report it will be explained how the problem was solved, the construction of the two algorithms and a comparison between both of them.**

Keywords **- graph, edge, vertex, min edge cover, exhaustive search, greedy search.**

I. Introduction

This report will give an insight of a graph problem, the minimum edge cover. It consists in finding out the smallest set of edges so that each vertex of the graph is incident to at least one of the edges in the set. For that, two different algorithms were designed and tested: an exhaustive search or brute force approach, where we take a look at every possible solution and discard the impossible ones; and another method using greedy heuristics. Afterwards, a formal analysis of both algorithms will take place, comparing computational complexities, the execution time, the number of basic operations and solutions and so on.

The programs can be run by executing the following commands:

```
$ python3 exhaustive_search.py
$ python3 greedy_search.py
```

The output files will be created in the path directory "results/{algorithm}" regarding the algorithm that was run. The plots and aggregated data that can be found in the path directory "results/aggregate" after running the programs.

II. The Problem

Well, before even starting to think about how to solve the problem, we must build our graph generator and define the structure of our graph, which will be undirected and without weights associated with the edges.

A file *Graph.py* was created in order to manage all the logic behind graph generation. I created several methods in order to separate the logic of the events.

The most important methods are the ones that create the skeleton of the graph: *gen_vertice()* which will create a vertex with random coordinates in a 2D plane (x and y being integers between 1 and 20) and check if a vertex with those coordinates was already created, if not, simply return the vertex; *gen_edge()*, a function that generates an edge - first, we must guarantee that every vertex is connected so that we don't get any isolated vertices and, consequently, a disconnected graph, that can't happen, otherwise we have no solutions - after all of them are connected, we can generate edges choosing vertices randomly, the method also forbids the generation of an edge that was already created, calling the method again in that case; *gen_graph()* which finally generates our graph, bearing in mind the Graph Theory that says that the number of edges must be between V-1 and V*(V-1)/2, with V being the number of vertices in order to have a valid connected graph. Different probabilities of the maximum number of edges are also tested in this graph generation: 12.5%, 25%, 50% and 75%.

In what comes to the representation of the graphs, the object graph contains list attributes for the vertices and the edges in the constructor.

The vertices are represented as tuples like these:

```
vertice = ("a", (3, 14))
vertices = [("a", (3, 14)), ("b", (18, 6)), ("c", (10, 10))]
```

The edges are also represented as tuples like these:

```
edge = ("a", "b")
edges = [("a", "b"), ("b", "c"), ("c", "a")]
```

Besides these methods, others were created because they were found useful later on implementing the algorithms (ex: *gen_adjacency_matrix()* and *gen_incidence_matrix()* to generate both matrices and *gen_adjacency_list()* to get the adjacency list out of the adjacency matrix.)

After the graph generation is completed, we have the list of edges and are now able to create sets of edges and explore the algorithms to find the **min edge cover**.

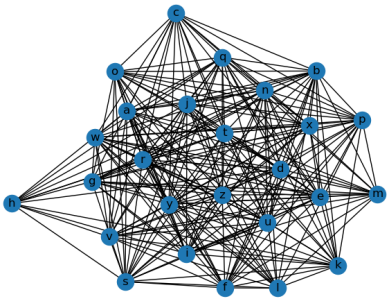

Fig. 1 - Example of a graph with 26 vertices and 75% of probability for the maximum number of edges.

A **min edge cover** is a set of edges so that each vertex of the graph is incident to at least one of the edges in the set.

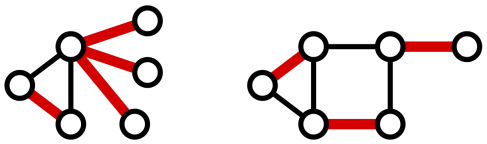

Fig. 2 - Example of **min edge covers** in two different graphs.

So, in order to find the min edge covers, two separate files were created for the logic of the two different algorithms, *exhaustive_search.py* and *greedy_search.py*.

III. Exhaustive algorithm

The exhaustive approach might have two different interpretations: either try to see every set of edges from the graph and return every solution (because there might be more than just one **min edge cover**) or do the same thing but just until one solution is found. Either way, it is already clear that this implementation will significantly cost computational resources since we need to calculate every possible set of edges for each graph.

After storing all the possible sets of edges in a list, the next step is to filter only the ones with `length = (number of vertices) / 2`, according to the website on reference [1], because the **min edge cover** always has this size. When the number of vertices is odd, the formula should be `length = ((number of vertices) / 2) + 1`.

Now, the strategy is simple. Let's think about a practical example… Imagine that our graph has 6 vertices and 75% probability of the maximum number of edges, that gives us 11 edges, but we've seen that the **min edge cover** size is half of the number of vertices, so we'll only iterate through the sets which size is equal to 3. Then, we analyse every edge of the set and check if the whole set contains every vertex. If so, a solution was found and the number of solutions is incremented, if not, we'll keep looking. This is actually a good implementation for smaller graphs that have few edges but it's not suitable for bigger graphs with several edges since its computational complexity is not that small, $O(2^n)$, with n being the number of vertices, to calculate every possible set of edges.

The great benefit of this algorithm is that it gives an insight of the total number of solutions for a given graph, the downside is that it doesn't work for more than 9 vertices (in my personal computer, maybe due to my faulty RAM), terminating the program with a *Segmentation Fault* error. I tried different approaches to fix this problem but they didn't work out.

IV. Greedy algorithm

The Greedy algorithm had a totally different approach from the previous one. Instead of looking around for every possible set of edges and every possible solution, we'll construct our own solution for the given graph, step by step.

This algorithm benefits from two methods of class Graph, *gen_adjacency_list()* and *get_list_with_num_of_neighbours()*, the first one analyses the adjacency matrix and returns a list of lists containing the neighbours (or connected vertices) for each vertex, in alphabetical order, like the example below:

```
neighbours = [["b, d"], ["a, c"], ["b, "d"], ["a", "c"]]
```

This means that vertex *"a"* is connected to vertices *"b"* and *"d"*, vertex *"b"* is connected to vertices *"a"* and *"c"*, and so on.

The other method, *get_list_with_num_of_neighbours()*, returns a list with the lengths of each element of the previous list. In this particular case:

```
num_of_neighbours = [2, 2, 2, 2]
```

These methods are essential in order to perform the algorithm. The steps are:

1. Lookup for the vertex with the least number of neighbours.
2. For that vertex, pick up the edge that connects it to the vertex that has the least number of neighbours.
3. Then repeat the process until it is guaranteed that every vertex was covered. The result is an optimal solution to the problem.

However, there are some things to point out: the priority here is to connect every vertex in order to get the **edge cover** with the least possible number of edges so, for that matter, when a vertex is covered, its number of neighbours is updated to 1000 in a virtual list, so that we can guarantee that it is not selected again over vertices that weren't picked yet.

Bearing this in mind, we've provided a much faster way to determine a **min edge cover** of a given graph.

The advantage of this algorithm is that its computational complexity is way less heavy than the exhaustive search one *O(1)*, since we're always looking for the vertex with fewer neighbours, which is our heuristic. The downside is that it requires more time to think and implement the algorithm and it only returns one possible solution unlike the exhaustive approach.

V. Results

In this section, we will analyse the resulting plots obtained by running the algorithms. It is important to refer that the maximum number of vertices considered for the exhaustive search was 9 because, as explained before, as of 9 vertices and a probability of 75% for the max number of edges, the program would exit with a *Segmentation fault (core dumped)* error, this might be due to the faulty RAM of my personal computer, in normal conditions it should go a little bit further. For the greedy approach, the maximum number of vertices considered was 52 because that's the length of the list used to name the vertices, `list(string.ascii_letters)`.

The execution times and number of basic operations was analysed for both algorithms. Although, only exhaustive search has a plot for the number of solutions. It wouldn't make sense to include a plot for the greedy algorithm because it only returns one solution per graph.

Execution times

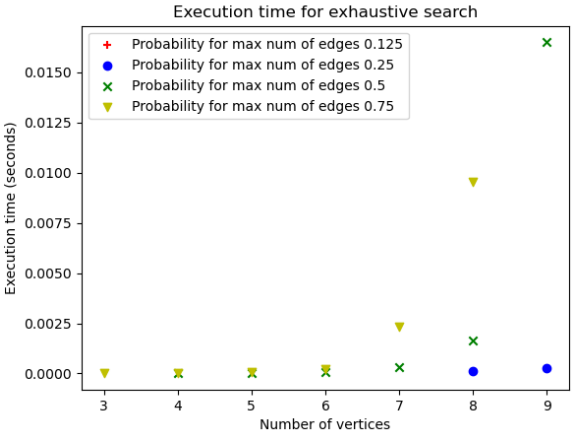

Fig. 3 - Plot of the execution times for the exhaustive search.

Even though there isn't that much data to analyse, we can prove a point. We can see that for few vertices, this algorithm works just fine and has execution times close to 0 but there are two spikes: for 8 vertices and a probability of 75% and for 9 vertices and a probability of 50%, those two are fairly distant from the rest. That allows us to infer that this algorithm quickly escalates when it comes to computational complexity also, it can't compute more than 9 vertices and a probability of 50%.

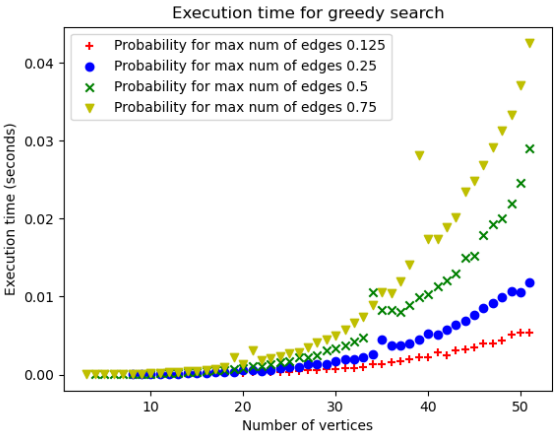

Fig. 4 - Plot of the execution times for the greedy search.

For the greedy algorithm we have plenty more data. It is clear that this approach is way lighter than the exhaustive search when it comes to execution times. We can easily see that while only 9 vertices with 50% probability are processed in around 0.016s with the exhaustive search, the greedy algorithm can process more than 4 times the amount of vertices, for a 75% probability. That's a huge gap between both algorithms.

Number of basic operations

In order to count the number of basic operations the following were considered:

- variable assignments;
- array accesses and appends;
- function calls.

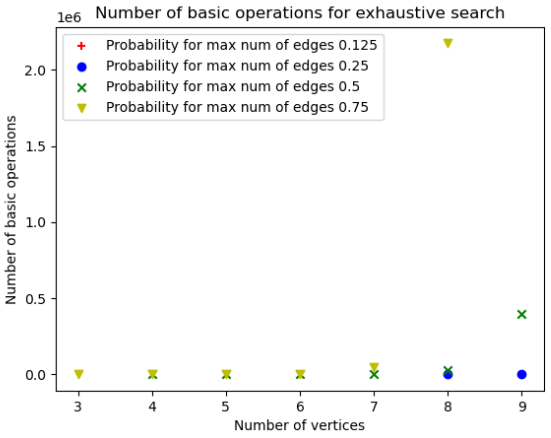

Fig. 5 - Plot of the number of basic operations for the exhaustive search.

This plot might look a bit deceiving, but that's only due to the lack of more data. We can see that there are few basic operations until n_vertices = 7, where we can see a slight increase of the number of basic operations when the probability is 75%. Well, when the number of vertices reaches 8, there is no room for doubt, with the number of basic operations surpassing the order of 2 millions for 75%. There is also a big gap from 8 to 9 vertices for 50% (29399 to 399321 operations).

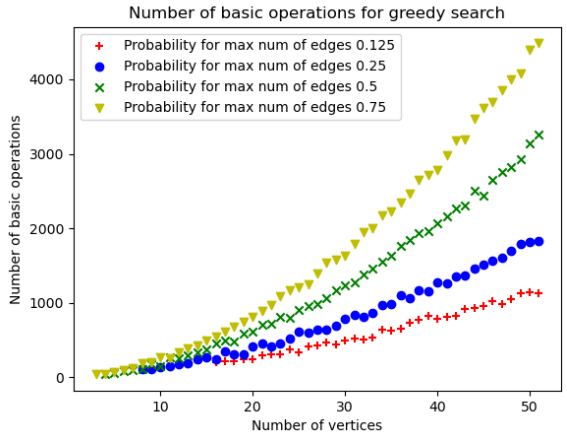

Fig. 5 - Plot of the number of basic operations for the greedy search.

The analysis of this plot is plain and simple, unlike the exhaustive approach, in the greedy search there are way fewer operations to register. The comparison between the two orders of magnitude is absurd, we're talking about millions of operations for 8 vertices in the exhaustive search and less than 5 thousand for more than 50 vertices in the greedy algorithm.

Number of solutions

For last, we're analysing the number of solutions for the exhaustive search. As concluded before, it wouldn't make sense to analyse the number of solutions for the greedy algorithm since it is always 1 for each graph.

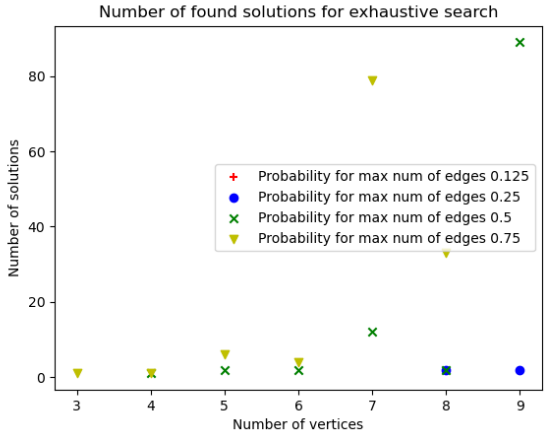

Fig. 6 - Plot of the number of solutions for the exhaustive search.

It's not that easy to analyse this plot because the randomness of the graph generator might have some implications in these results. A vertex that is only incident to one edge of the graph explains the so few solutions for 8 vertices in both 25 and 50%, that probably wouldn't happen with another random seed. Although, we can see that there is a significant peak for 9 vertices and 50% (89 solutions), which indicates that we will have more solutions as the graph grows in dimension.

VI. Conclusion

In conclusion, comparing both approaches, we can say that greedy search is preferable. Although it doesn't give us a detailed insight of the total number of solutions like exhaustive search, it is capable of finding a proper solution for much more graphs with way more vertices and in fewer time than the brute force approach. We can clearly see that by comparing the number of basic operations and the execution times for each algorithm.

```
OBJECT-TYPE MACRO::=
BEGIN
  TYPE NOTATION ::=
list(string.ascii_letters)
    ReferPart
    IndexPart
    DefValPart
VALUE NOTATION ::= value (VALUE ObjectName)
```

References

[1] Cardinality of a minimum edge cover https://www.geeksforgeeks.org/program-to-calculate-the-edge-cover-of-a-graph/

[2] Finding Edge Cover https://reference.wolfram.com/language/ref/FindEdgeCover.html

[3] Definition of minimum edge cover https://mathworld.wolfram.com/MinimumEdgeCover.html

[4] NetworkX documentation (Python library used for graph files) https://networkx.org/documentation/stable/index.html

[5] NetworkX min_edge_cover() method https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.covering.min_edge_cover.html

[6] Vertex cover problem https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/