1st AA Project — Exhaustive Search Maximum Weight Independent Set

Rui Fernandes

***Abstract* –This report seeks to present how the author solved the maximum weight independent set graph problem with exhaustive search algorithms. The chosen coding language was Python(3.7), and the algorithms will be explained alongside a formal computational complexity analysis. Ending with a comparison of results and predictions of larger problem instances.**

***Resumo* –Este documento visa apresentar como o autor resolveu o problema de grafos que consiste em descobrir o *set* independente com maior peso, recorrendo a algoritmos de pesquisa exaustiva. A linguagem the programação utilizada foi Python(3.7) e os algoritmos seram explicados acompanhado de uma analise formal de complexidade computacional. Finalizando com uma comparação dos resultados e uma previsão para instancias de problemas maiores.**

***Keywords* –Graph, Independent Set, Maximum Weight, Vertices, Greedy, Brute Force**

I. Introduction

The problem presented for this project and that this report seeks to explain is the simple graph problem of finding a maximum weight independent set for a given independent set using a brute force algorithm and a greedy one.

With the report comes the code files used and the result files obtained, the graph files generated where not included to save space and since they can be easily generated again as the random seed is constant.

To run the code one should first generate all the graphs, then run the chosen algorithm file:

```
$ python generate_graphs.py
$ python MWIS_brute_force.py
$ python MWIS_greedy.py
```

II. The Problem

It's important to note a few things about the problem itself, starting with the graphs and their structure.

A graph in this context is represented first by it's list of vertices, which in practice is a dictionary with keys being the name of the given vertex, and with values being a tuple including the vertex's coordinates and the weight it carries, both aspects randomly determined.

An example vertex list would be:

```
{"a": [[5, 4], 81], "b": [[2, 1], 37]}
```

Additionally is the adjacency list that represents the graph's edges, and example for the above vertices would be:

```
{"a": ["b"], "b": ["a"]}
```

In relation to the names, they come from a list of all letters lowercase then uppercase, restraining the problem to a maximum n of 52. As for the coordinates they go from 0 to 9 and the weights from 1 to 100.

Another important aspect of the problem is establishing some concepts. An independent set of a graph is a subset of vertices of which none is adjacent to one another. The weight of a set is the sum of the weights off all the vertices that form it. Given this the objective of this problem, as before said, is finding the independent set of every given graph that holds the most weight.

III. The brute force algorithm

The first algorithm developed, as requested in the project, was an exhaustive search brute force algorithm that would always find the optimal solution for the given graph.

To always find the optimal solution, the only option is running trough every subset that exists within the given graph. Before doing so, null values are set for **max_weight** and **max_set** variables for comparison afterwards. With each subset analysed the algorithm first checks if it is independent and if not it is skipped, if it is then the weight is calculated and compared with the previously found maximum weight and the higher one will remain, along with the correspondent subset.

This brute force algorithm is simple but extremely costly, analysing the computational complexity we simply need to know how many subsets a graph has and subtract one since we don't analyse the empty subset, given this, the computational complexity is $2^n - 1$, n being the number of vertices, an extremely high exponential complexity that will result in quickly rising number of iterations.

IV. The greedy algorithm

In addition to the previous algorithm, a greedy approach algorithm was developed, the heuristic used was simply the weight of each vertex. This algorithm is even simpler in practice, we start by sorting the vertex list by ascending order of weight, then pick the last element, the one with higher weight. Afterwards, to maintain feasibility, we remove all adjacent vertices to the chosen one and repeat the sorting and picking process. One by one the heaviest vertices that preserve

feasibility are picked and added to the solution subset, and their weights to the total weight.

This algorithm has a massively reduced computational complexity, it can be constant at best, when all vertices are connected so it does a single iteration, and linear at worse, when no vertex is connected so it need to iterate through the **n** vertices. Regardless it has extremely low complexity, however it comes at a cost, it more often than not finds a non-optimal solution. This happens because from the get-go it can pick a vertex with high weight that as a lot of connections, limiting the amount of further options, when the true optimal solution could be, and often is, a subset with a lot of vertices of lower weight.

To better exemplify this notion, representations of the generated graph for $n = 6$ will be given, firstly the graph as it is with the vertices' weights represented alongside their name (important to note that vertex 'e' and vertex 'd' are not connected, they simply have close coordinates), then the solution, in red found by the brute force algorithm, and the same for the greedy algorithm.

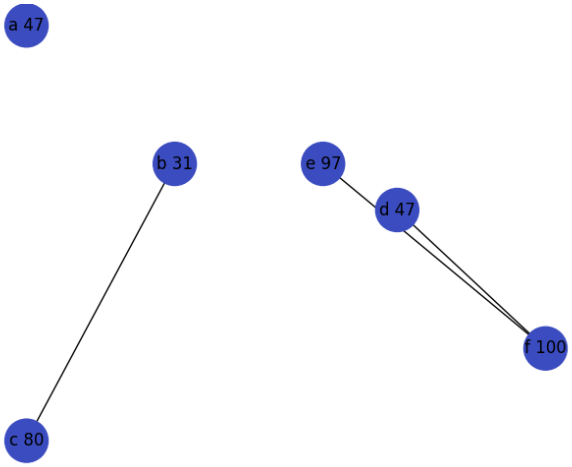

Fig. 1 - Graph for n = 6

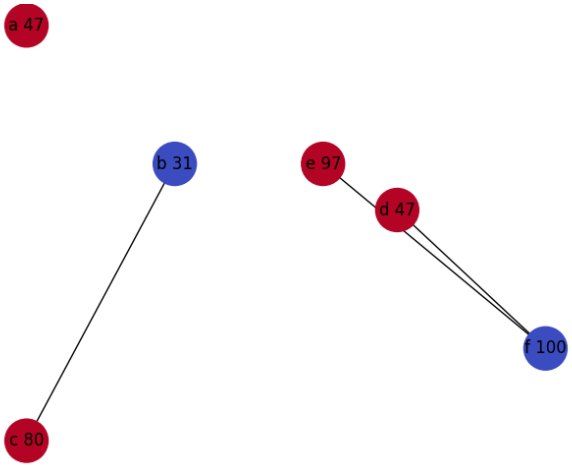

Fig. 2 - Solution found by Brute Force Algorithm

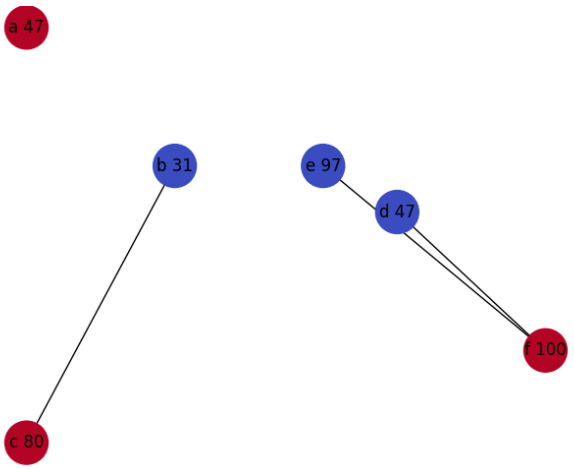

Fig. 3 - Solution found by Greedy Algorithm

V. RESULTS

In this section, the results obtained through running the algorithms and extracting information will be exposed. It's relevant to note that the experiments where from $n = 2$ to $n = 30$ for the brute force algorithm, and from $n = 2$ to $n = 52$ for the greedy algorithm.

Firstly, regarding the maximum weight solution found, both algorithms can be compared, the brute force approach in red and the greedy one in blue, as explained before it can be seen that the greedy approach often finds a solution lower than optimal. Curious to note is that because of the random generation of the graphs, at higher values of n solutions can have huge weights or quite low ones.

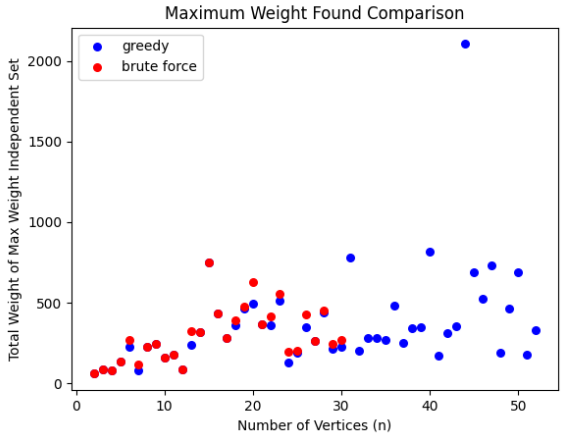

Fig. 4 - Maximum Weight Found Comparison

Secondly, analysing the amount of iterations that each algorithm took with the rising number of vertices we can see the exponential rise of the brute force algorithm, and the varying but low values of the greedy approach as it ends up representing the length of the solution subset found.

When it comes to the execution time of the algorithms, it's possible to see that for the brute force approach it rises in a similar pattern than the iterations

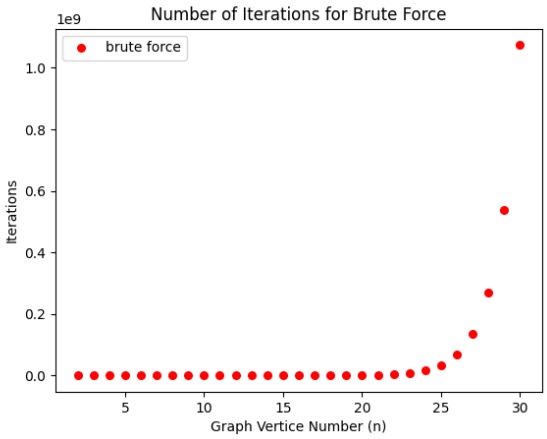

Fig. 5 - Number of Iterations for Brute Force

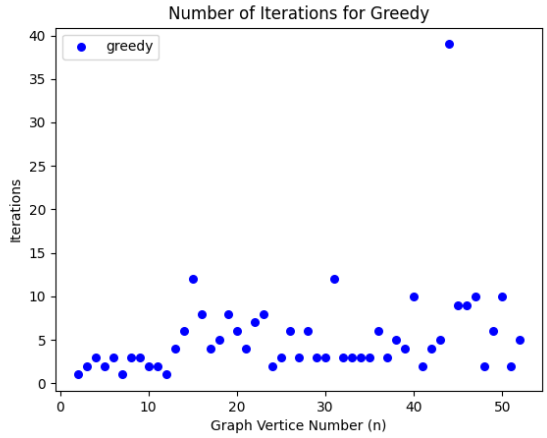

Fig. 6 - Number of Iterations for Greedy

graph. As for the greedy algorithm, it wasn't worth showing for it was always so quick that it doesn't detect a time value, giving 0 for all 52 graphs. A further experiment was made where the graph vertices where extended to maximum coordinates of 100 and enough names where provided to generate 200 graphs, considering these values, the extended greedy algorithm results still only raised to 1 millisecond on two occasions and seemingly without pattern. So we can expect times for larger instances of the problem to be exponentially higher for the brute force algorithm, and near zero for the greedy approach.

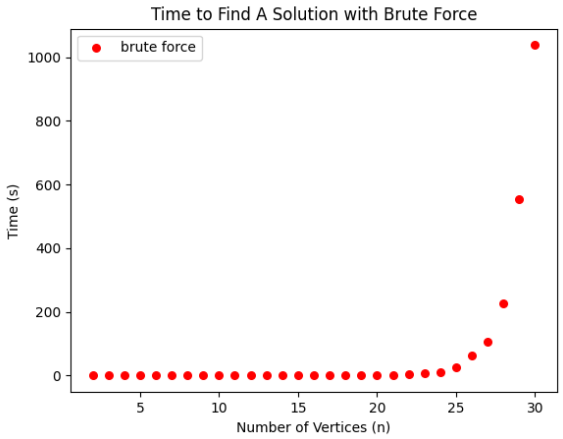

Fig. 7 - Time to find a Solution with Brute Force

VI. Conclusion

To conclude this document, we now understand that, as expected a brute force approach to this problem is very costly, more precisely it has an exponential computational complexity. On the other hand the requested greedy approach developed is extremely time efficient however is a poor choice for optimal or even near optimal results.

After some research one can find other algorithms that could be employed for this problem and yield optimal results while preserving a lower complexity. One example is a good divide and conquer approach, that while complex to make, can yield a quadratic computational complexity. Better yet would be a dynamic programming approach that would be simple and have a linear at max computational complexity.