1st Project: Minimum Weight Cut with Greedy and Exhaustive Search

Advanced Algorithms

José Miguel Costa Gameiro, 108840

***Resumo* - This document presents a brief study conducted within the Advanced Algorithms course to analyse Exhaustive Search and Greedy approaches, as well as the collected results and conclusions reached.**

I. Introduction

The **Minimum Weighted Cut** problem is a fundamental challenge in graph theory, focusing on partitioning the vertices of an undirected graph in a way that minimises the sum of edge weights crossing the partition. Formally, given a graph G(V, E) with a set of vertices V and edges E, where each edge has an associated weight, the goal is to split the vertices into two disjoint subsets, S and T, such that every vertex in V is assigned to one of these subsets. The objective is to ensure that the total weight of edges connecting vertices in S to vertices in T is as low as possible, yielding a "cut" with minimal weight.

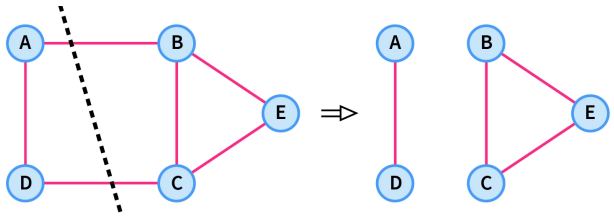

Fig. 1 - Example of the Minimum Weight Cut problem

Fig. 1 illustrates a minimum weighted cut in a graph. On the left, the original undirected graph is shown with vertices A, B, C, D, and E, where the dashed line represents the cut that will divide the graph into two subsets. On the right, the result of this partition, separating the vertices into two groups, with the minimum weight edges cut to achieve this division.

However, finding a minimum weighted cut can be computationally intensive, especially as the number of vertices and edges in the graph grows. **Exhaustive search**, which explores every possible partition, guarantees an optimal solution but is typically infeasible for large graphs due to exponential growth in computational complexity. This complexity makes heuristic approaches, such as **greedy algorithms**, appealing alternatives; these methods do not guarantee optimal solutions but can provide near-optimal results with significantly lower computational requirements.

II. Study Environment

The study environment consists of randomly generated graphs created using a seed based on the student's number and stored locally. Each graph instance is defined by a set of 2D vertices with unique integer coordinates and a set of edges, both determined randomly.

A. Generated Graphs

The graphs used to solve the problem were generated randomly with a defined seed. For each number of vertices, four graphs were typically generated with edge densities of **12.5%**, **25%**, **50%**, and **75%** of the maximum possible edges. Coordinates between one and twenty were assigned to each vertex. The weight of each edge is the distance between two points, obtained with the function presented in Fig. 2.

```python
def calculate_distance(self, node_1, node_2) -> float:
    """Calculate the distance between 2 nodes"""

    x1, y1 = self.node_positions[node_1]
    x2, y2 = self.node_positions[node_2]

    return round(((x2 - x1) ** 2 + (y2 - y1) ** 2) ** 0.5, 2)
```

Fig. 2 - Obtain the distance between two nodes, to assign it to the density of the edge between the two nodes in question

If a graph does not meet the minimum edge requirement of N-1 (where N is the number of vertices), it is not generated, and a new edge density is selected. The files containing a image of the graph follow this naming structure:

"*Graph<graph_number>Vertex<vertex_number>Edges<edge_number>.png*"

An image that contains the graph is only generated when a solution is found, to highlight the set of edges that were removed with the different colour (red).

All the graphs are generated and drawn using *Python*'s "*Networkx*" and "*Random*" packages and stored in a file to be used for the analysis.

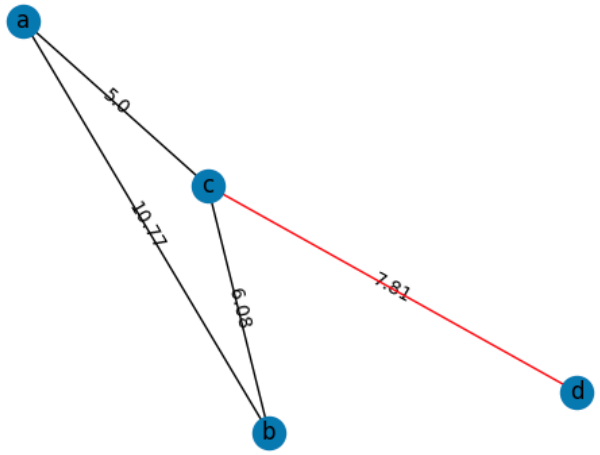

Fig. 3 - Example of a generate graph

III. COMPUTATIONAL ANALYSIS

A. Exhaustive Search

The exhaustive search algorithm consists of a *brute-force* approach usually used to solve combinatorial problems. By using this approach, it is guaranteed to obtain the minimum weight cut for a graph, but it may lead to execution of unnecessary operations with high complexities and too inefficient to solve the problem in larger scales.

When using exhaustive search to solve the problem for a graph, it generates all combinations of vertices, leaving out the empty and complete set.

Following the algorithm implemented in the *exhautive_search.py* file, its divided into three parts:

- **Generating Cut Combinations:** A list that contains all the possible combinations of node subsets is generated to represent the potential cuts.
- **Evaluating Each Cut:** Then it iterates through each subset/partition and computes its complementary subset. For each combination, it calculates the *cut cost* (the sum of weights of edges crossing between the two subsets)
- **Tracking the Minimum Cut:** During each iteration, if the calculated *cut cost* is lower than the previously recorded, the *minimum cost* and best subsets are updated.

The computational complexity for the function used to obtain all the possible vertex combinations is $O(2^n)$ and can be represented by the following formula:

$$\sum_{k=2}^{n} \binom{n}{k} = 2^n - 2$$

Where **n** represents the number of nodes. By removing two, we're guaranteeing that the empty and containing all the elements subsets are not being included. The computational complexity to remove this 2 subsets is $O(1)$, while the operation to calculate the associated cost to each subset has a computational complexity of $O(n)$.

Combining these two computational complexities we get this

$$O(n) \times O(2^n) = O(n2^n)$$

B. Greedy Search

The greedy search algorithm is an approach that builds a solution by making a sequence of choices, each of which is optimal. Unlike exhaustive search, it aims to find an acceptable solution more quickly by prioritising certain elements according to a heuristic. Although it does not guarantee an optimal solution, this method often provides a close approximation efficiently.

In this case, the greedy search is combined with a heuristic to accelerate finding a solution to the problem in question. The heuristic involves sorting the adjacency list in ascending order, based on the number of edges associated with each vertex. Vertices with fewer edges are prioritised, as they are more likely to yield a lower cut cost, increasing the chances of finding a viable solution efficiently.

Considering the graph presented in Fig.3, it presents the following adjacency matrix:

a⟶b⟶c

b⟶a⟶c

c⟶a⟶b⟶d

d⟶c

For this example the solution cost is **7.81** and is associated with the cut of the **CD** edge. The subset **[S]** would contain only the **D** vertex, while the subset **[T]** would contain the [**A, B, C**] vertices.

From the perspective of Greedy Search, if the adjacency list is ordered by the number of connected nodes, the solution can be found immediately. This approach allows certain subsets to be skipped, resulting in a faster solution. In this context, only the smallest adjacency lists (and those with the same size) are considered, while the rest is discarded. In the example shown in Fig. 3, only the following adjacency list would be taken into account:

d⟶c

For this algorithm it exists 2 main operations: sort the adjacency list and obtain the cost of a cut, with a computational complexity of $O(n)$. Combining the complexity of both operations we obtain the following complexity $O(nlog(n))$, where **n** represents the number of vertices.

IV. RESULTS

To test the solutions developed it generated graphs with a number of vertices between **4** and **20.** For each graph generated, a text file was created with the following name structure:

"*Graph_<graph_number>__Vertex_<vertex_number>_ _Edges_<edges_number>.txt*"

And in each file it would appear the following information:

- Number of nodes and edges
- Cost
- Subsets generated
- Adjacency matrix
- Execution time
- Basic operations performed
- Number of solutions found

A. Execution time

To evaluate the execution time for each graph, a graphic displaying the evolution of the execution times for each approach, changing the number of nodes.

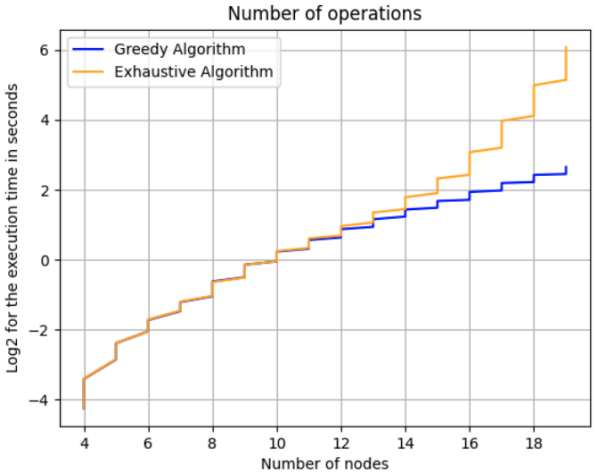

Fig. 4 - Line chart with the evolution of the execution time for the Greedy and the Exhaustive Search

To normalise the data, base-2 logarithmic transformation was applied to the execution times. Analysing the chart, we observe that the greedy algorithm exhibits a faster computational performance compared to the exhaustive algorithm.

B. Number of operations

To analyse the number of operations taken by each approach, in this case it corresponds to each iteration taken to find the solution. The results can be viewed in the following figure:

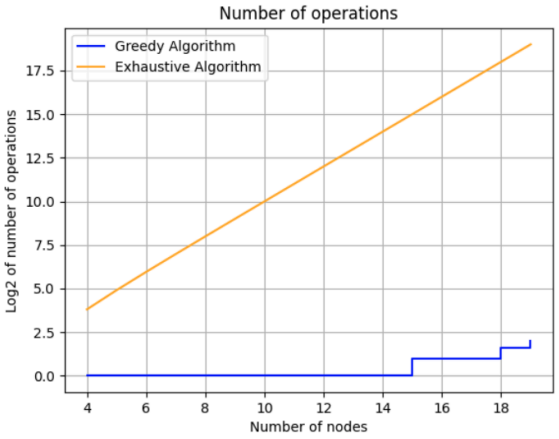

Fig. 5 - Line chart with the evolution of the number of operations done for the Greedy and Exhaustive Search Algorithms

Similar to the chart presented in Fig. 4, the values obtained for the number of operations were normalised by applying a base-2 logarithmic transformation.

We can observe that the number of operations for the exhaustive search increases at a much faster rate compared to the greedy search. This is because the exhaustive algorithm explores all possible solutions, which can be beneficial in guaranteeing an optimal solution but can also lead to significantly longer computation times, especially as the problem size grows.

C. Number of Solutions found

Regarding the number of found solutions, it was rare to encounter situations with multiple solutions due to the edge weight calculation based on the distance between two points. This makes it improbable to randomly generate vertices with identical edge counts and distances.

Neither search algorithm encountered graphs with two possible solutions. The greedy algorithm, by not exploring all possible subsets, further reduces the likelihood of finding such graphs.

D. Comparing solutions found for both approaches

As mentioned earlier, the greedy search algorithm does not always yield the optimal solution, defined as the solution with the lowest cost. In comparing the solutions produced for each graph by both approaches, some differences emerged: in certain cases, the exhaustive algorithm provided a lower-cost, optimal solution but required more operations and longer execution time. Conversely, the greedy algorithm produced solutions with a higher cost but required fewer operations and executed more quickly.

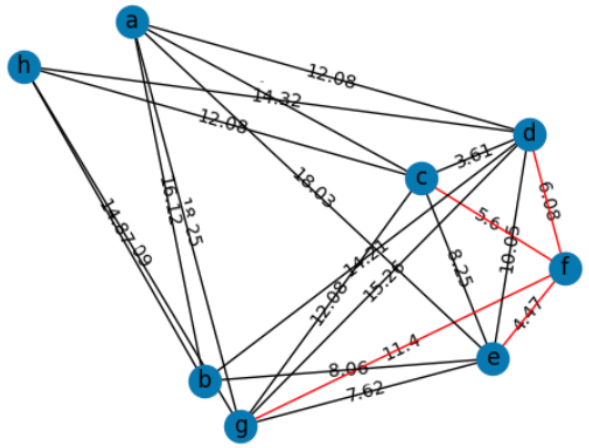

Fig. 6 - Example of a graph with the solution found using the exhaustive search approach

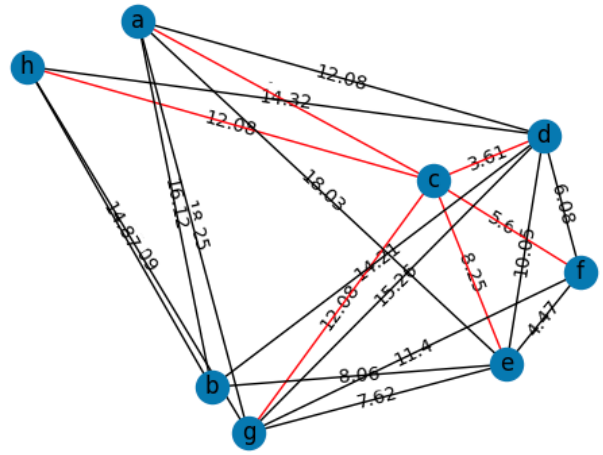

Fig. 7 - Example of a graph with the solution found using the greedy search approach

The graph in Fig. 6 and Fig. 7 illustrates the differences between the solutions. The greedy algorithm found a solution with a **cost** of **52.31**, completing in **0.649 seconds** and requiring only **2 operations**. In contrast, the exhaustive algorithm achieved a **lower-cost** solution of **27.61** but required **0.660 seconds** and **254 operations** to reach it.

E. Scalability

For 20 vertices, the execution time of the Greedy algorithm is, approximately, 600 times faster than the Exhaustive algorithm.

Considering problems of bigger dimensions, the algorithms execution time can be determined and estimated through the following formula:

For the **exhaustive algorithm**, where the value 145 represents the execution time of the twenty vertices

$$\frac{n * 2^n}{20 * 2^{20}} * 145$$

For the **greedy algorithm**, where the value 0.2 represents the execution time for the twenty vertices:

$$\frac{n * log(n)}{20 * log(20)} * 0.2$$

V. CODE STRUCTURE AND ORGANIZATION

In the submitted zip file, there exists a directory called "*src*" where inside exists all the code developed for this project, as shown in Fig. 7.

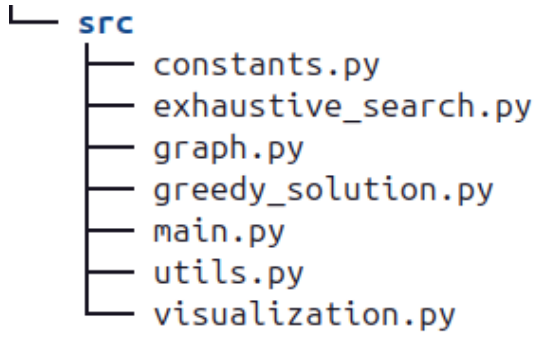

Fig. 8 - Folder organisation

The **constants.py** file contains two variables that have always the same value which are the **STUDENT_NUMBER**, and the **EDGES_PERCENTAGE** which is a list containing all the edge percentages used to generate the graphs.

The **exhaustive_search.py** file has a class called ***ExhaustiveSearch*** used to solve the minimum weight cut problem using an exhaustive search algorithm.

The **greedy_solution.py** file is similar to the file described above, with the difference of having a class called **GreedySolution** which is used to solve the problem using a greedy search algorithm.

The **graph.py** contains also a class called **Graph** which is responsible for the creation of a graph. Where it creates the vertices and associates each one with a position, it also creates the edges and associates them with weight values. When the edges are created, some validations are

performed to avoid including multiple edges between the same vertices. It also calculates the adjacency matrix for each generated graph

The **main.py** is the main file of the solution developed, meaning, it's the file that should be executed. To execute it is required to pass two arguments, where one is optional. The first is the **mode** (using "***-m***" or "***--mode***") which defines the approach that will be used to solve the problem. The possible values can either be "***greedy/GREEDY***" or "***exhaustive/EXHAUSTIVE***". There is also the **nodes argument**, represented with the "***-n***" or "***--nodes***" that represents the number of nodes used to generate the different graphs. This argument is optional, and if it's not passed then, by default, the number of nodes will be **16**.

The **utils.py** contains two functions used by both the approaches, one to calculate the cost of a cut, and then to get all possible combinations for a cut.

When running the main file with a selected approach, a directory is created to store files containing the generated graph and solution results, with naming structure and content as previously described. If the **greedy** approach is selected, the directory is named "***time_greedy_information***"; otherwise, it is named "***time_exhaustive_information***."

VI. CONCLUSION

Greedy search and exhaustive search are both effective methods for solving problems, with each being better suited to different problem sizes. For smaller problems, the exhaustive approach is generally the most appropriate as it guarantees an optimal solution. For larger-scale problems, however, greedy search is preferred due to its faster performance, though it may not always yield the optimal solution.

REFERENCES

[1] Get OPT Library, https://docs.python.org/3/library/getopt.html

[2] Stanford Education https://web.stanford.edu/class/archive/cs/cs161/cs161.1172/CS161Lecture16.pdf

[3] Minimum Weight Cut: https://en.wikipedia.org/wiki/Minimum_cut

[4] NX Graph: https://networkx.org/documentation/stable/tutorial.html