Algoritmos Avançados
1st Project - Finding minimum weight edge dominating set

Miguel Miragaia
Student 108317, DETI
Universidade de Aveiro
Aveiro, Portugal
miguelmiragaia@ua.pt

***Abstract*—The main objective of this article is to analyze the Minimum Weight Edge Dominating Set Problem with the exhaustive search and Greedy search algorithms and for different sizes of the problem. Formal analysis of the computational complexity of each algorithm are made, and complemented with an experimental analysis based on their execution times and number of basic operations.**

***Index Terms*—Minimum Weight Edge Dominating Set, Exhaustive Search, Greedy Search, Algorithms**

I. Introduction

The Minimum Weight Edge Dominating Set (MWEDS) problem has significant applications in network optimization and resource minimization problems. In a given undirected graph $G = (V, E)$, where V is a set of vertices and E is a set of edges, an edge dominating set (EDS) is defined as a subset of edges such that every edge not in this subset shares at least one vertex with an edge in the subset. The MWEDS problem focuses on identifying the edge dominating set with the minimum total weight, where the weight of an EDS is calculated as the sum of the weights of the edges within this subset [1].

EDS has important real-world applications, such as in telecommunications, where minimizing the number of edges needed for network reliability and coverage is essential for cost efficiency [2].

II. Motivation

The primary motivation for this study is to understand the efficient solutions for the Minimum Weight Edge Dominating Set (MWEDS) problem, a critical challenge in graph theory with practical applications across various domains. Additionally, exploring both exhaustive search and heuristic approaches provides insights into computational complexity and the trade-offs between solution precision and algorithm efficiency.

III. Problem Analysis

The Minimum Weight Edge Dominating Set (MWEDS) problem can be formally defined in the context of an undirected weighted graph $G = (V, E)$, where V represents a set of vertices and E represents a set of edges connecting the vertices. For each edge $e \in E$, a non-negative weight $w(e)$ is assigned, reflecting the cost or significance associated with that edge. An edge dominating set (EDS) is a subset $D \subseteq E$ such that every edge in $E \setminus D$ is adjacent to at least one edge in D. The objective of the MWEDS problem is to find an edge dominating set D such that the total weight $\sum_{e \in D} w(e)$ is minimized.

The complexity of this problem lies in its NP-complete classification, which implies that exact solutions are generally impractical for large-scale graphs due to the exponential growth in computation time as the graph size increases. Yannakakis and Gavril demonstrated that the EDS problem remains NP-complete even for specialized graph classes. [3]. While some specific graph structures admit polynomial-time solutions or approximation schemes, the general MWEDS problem necessitates computationally intensive methods for exact resolution.

The importance of solving the MWEDS problem extends to fields like network design, sensor placement in surveillance systems, and resource allocation in logistical networks. In these applications, achieving a minimal edge dominating set with a low total weight translates directly to reduced costs and improved resource efficiency. Through computational experiments, it is pretended to evaluate the scalability of the algorithms, provide insights into their performance on varied graph structures, and identify the largest problem instance solvable on standard computational resources within a feasible timeframe. [4]

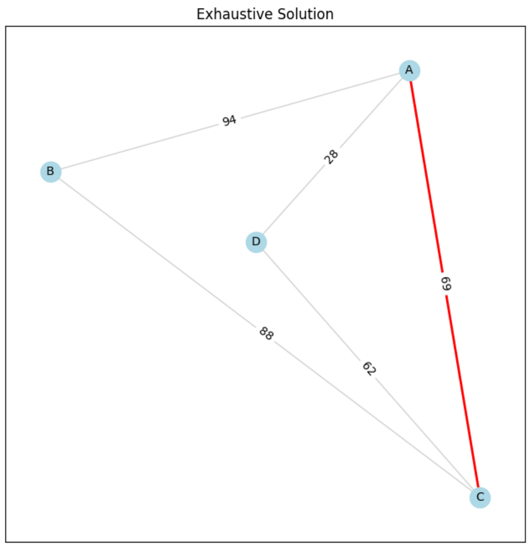


Fig. 1: MWEDS Solution for graph with 4 nodes and 75% edge density

IV. Problem Solution

To address the Minimum Weight Edge Dominating Set (MWEDS) problem, were implemented two distinct algorithmic approaches: an exhaustive search algorithm and a greedy heuristic algorithm. Each of these methods offers unique strengths and is suited to different scales of graph complexity. Here, are detailed their methodologies, benefits, and limitations.

A. Exhaustive Search Algorithm

The exhaustive search algorithm provides an exact solution to the MWEDS problem by evaluating all possible subsets of edges in the graph. For each subset, it checks if it constitutes a valid edge dominating set (EDS)—meaning each edge not in the subset shares at least one vertex with an edge within the subset. If it does, the algorithm computes the total weight of the subset and keeps track of the smallest-weighted set that satisfies the edge domination requirement.

This approach ensures an optimal solution but comes with high computational costs due to exponential complexity, requiring the examination of every subset of edges. This exponential growth in computation time limits the algorithm's feasibility to graphs with a large number of edges. Despite this, it serves as a baseline solution, providing an exact measure against which other algorithms, such as heuristics, can be evaluated in terms of solution quality and efficiency.

Algorithms 1 and 2 represent the exhaustive search.

Algorithm 1 Exhaustive Search for Minimum Weight Edge Dominating Set (MWEDS)

```
1: Procedure ExhaustiveSearchMWEDS($G$)
2: $min\_weight \leftarrow \infty$
3: $min\_weight\_set \leftarrow []$
4: $edges \leftarrow$ list of edges in $G$ with their weights
5: $all\_edges \leftarrow$ number of edges in $G$
6: for $r \leftarrow 1$ to $all\_edges$ do
7:     for each subset $edge\_set$ of size $r$ from $edges$ do
8:         if IsEdgeDominatingSet($G$, $edge\_set$) then
9:             $weight \leftarrow$ sum of weights of edges in $edge\_set$
10:            if $weight < min\_weight$ then
11:                $min\_weight \leftarrow weight$
12:                $min\_weight\_set \leftarrow edge\_set$
13:            end if
14:        end if
15:    end for
16: end for
17: return $min\_weight\_set, min\_weight$
18: End Procedure
```

Algorithm 2 Check Edge Dominating Set

```
1: Procedure IsEdgeDominatingSet($G$, $edge\_set$)
2: for each edge $(u, v)$ in $G$ do
3:     if no edge $(u1, v1, w)$ in $edge\_set$ has $u$ or $v$ as an endpoint then
4:         return False
5:     end if
6: end for
7: return True
8: End Procedure
```

Algorithm 3 Greedy Minimum Weight Edge Dominating Set (MWEDS)

```
1: Procedure GreedyMWEDS($G$)
2: $edge\_list \leftarrow$ sorted list of edges in $G$ by weight in ascending order
3: $dominating\_set \leftarrow []$ {Initialize an empty list for the dominating set}
4: $covered\_edges \leftarrow \emptyset$ {Initialize an empty set for covered edges}
5: for each $(u, v, weight)$ in $edge\_list$ do
6:     if $(u, v) \notin covered\_edges$ then
7:         Add $(u, v, weight)$ to $dominating\_set$
8:         Add all edges incident to $u$ and $v$ to $covered\_edges$
9:         if every edge in $G$ has at least one node in $dominating\_set$ then
10:            break
11:        end if
12:    end if
13: end for
14: $total\_weight \leftarrow$ sum of weights of edges in $dominating\_set$
15: return $dominating\_set, total\_weight$
16: End Procedure
```

B. Greedy Heuristic Algorithm

The greedy heuristic algorithm offers a computationally efficient, approximate solution to the MWEDS problem by iteratively selecting edges based primarily on their weights. Unlike the exhaustive approach, the greedy algorithm does not evaluate all possible subsets of edges, making it more feasible for larger graphs. Although it prioritizes lightweight edges, it does not guarantee the optimal solution but provides a good approximation with significantly lower computational requirements.

This algorithm is represented in Algorithm 3.

C. Comparative Evaluation of Algorithms

To assess the effectiveness of these algorithms, were conducted computational experiments on varied graph instances. The exhaustive search solution provides a benchmark, confirming the optimal edge dominating set and its weight. The performance of the greedy heuristic is measured by comparing its solution quality to the exhaustive search output, using metrics such as total weight, number of basic operations, computation time, and scalability.

The evaluation focuses on:

- **Solution Quality**: The proximity of the greedy algorithm's solution to the optimal solution found by exhaustive search. Figure 2 provides a visualization of the proximity of the Greedy Heuristic Algorithm to the optimal solution given by the Exhaustive Search Algorithm.

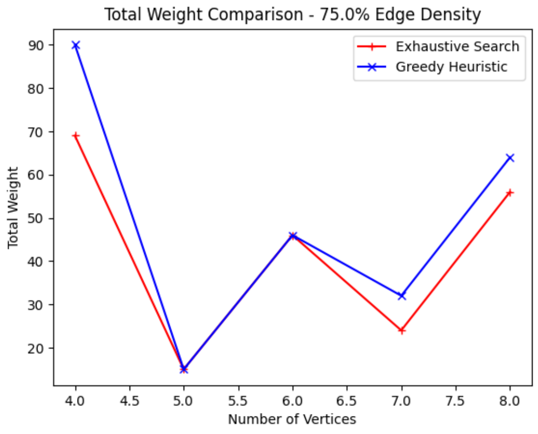


Fig. 2: Greedy Solution compared to the optimal solution in graphs with 75% edge density

- **Execution Time**: The computational efficiency of each approach, particularly the feasibility of using exhaustive search as graph size grows. Figure 3 shows a time ratio comparison between the Greedy Heuristic Algorithm and the Exhaustive Search Algorithm for an edge density of 50%.

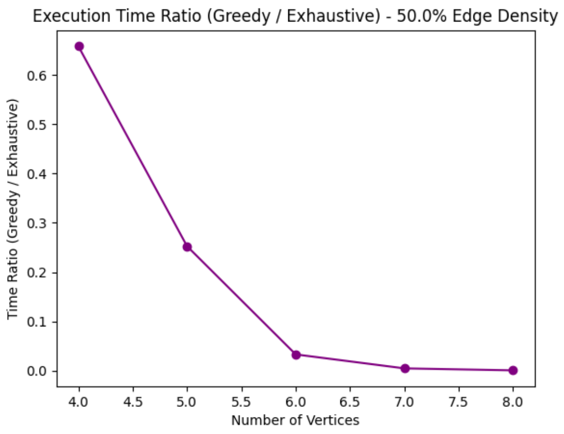


Fig. 3: Execution Time Ratio between Algorithms - (50%) Edge Density

- **Basic Operations**: The number of basic operations required by each algorithm. The exhaustive search algorithm performs an exponential number of basic operations, as it must evaluate all possible subsets to find the optimal solution, resulting in rapid increase in operations as graph size grows. In contrast, the greedy heuristic algorithm, shown in Figure 4, selects edges based on a predefined heuristic, resulting in a linear or near-linear increase in basic operations, making it far more scalable for larger graphs.

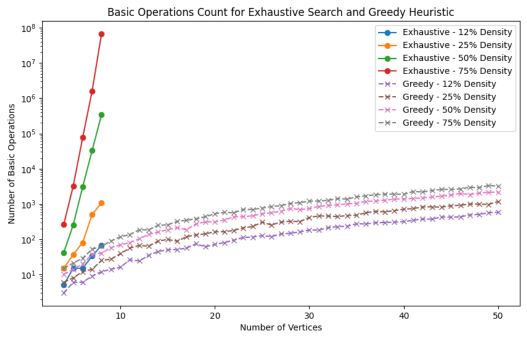


Fig. 4: Comparison of Basic Operations between Greedy and Exhaustive Algorithms

- **Scalability**: The largest problem instance solvable by each algorithm within a reasonable time frame on standard computational resources. Shown in Figure 5.

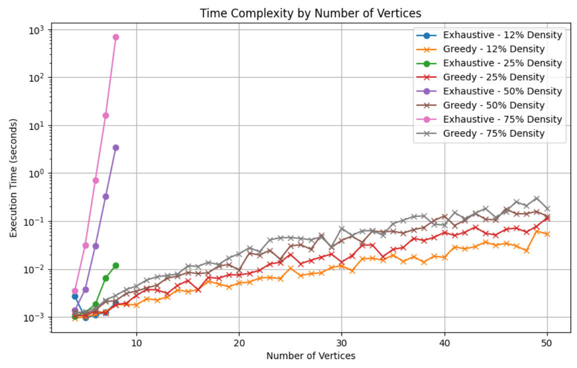


Fig. 5: Time Complexity by Number of Vertices

It is possible to visualize in Figure 5 that the largest problem (with 75% edge density) instance solvable by the exhaustive search is a graph with 8 nodes, whilst the greedy heuristic algorithm is expected to compute graphs with a much larger number of vertices in much less time.

Through this comparison, it is intended to establish the conditions under which the greedy heuristic remains an effective alternative, offering a practical solution to the MWEDS problem for larger-scale applications where exhaustive search is not viable.

V. Formal Analysis of the Algorithms

A. Exhaustive Search Algorithm

The exhaustive search algorithm explores all possible subsets of edges in the graph to determine the minimum weight edge dominating set (MWEDS). Given a graph with $|E|$ edges, there are $2^{|E|}$ possible subsets of edges, and the algorithm must check each subset to determine if it forms a valid EDS. Consequently, the time complexity of this algorithm is $O(2^{|E|})$, making it infeasible for large graphs due to exponential growth in computation time.

Additionally, the space complexity of the exhaustive search algorithm is also significant, as it requires storage for each edge subset being evaluated. This leads to a space complexity of $O(2^{|E|})$, which becomes impractical for large values of $|E|$. Due to these factors, the exhaustive search is limited to small graphs or specialized applications where the exact solution is critical, and computational resources are less constrained.

B. Greedy Heuristic Algorithm

The greedy heuristic algorithm operates by iteratively selecting edges based on minimum weight, prioritizing lightweight edges rather than explicitly maximizing the coverage of undominated edges. This approach reduces computational overhead compared to the exhaustive search approach, providing a practical solution to the MWEDS problem, especially for larger graphs.

In each iteration, the algorithm processes the edges in sorted order by weight, adding an edge to the dominating set if it covers any remaining undominated edges. Since the edges are pre-sorted by weight, each edge addition operation involves a constant-time check to see if the edge is already covered, followed by updates to the set of covered edges.

The worst-case time complexity for this greedy algorithm is dominated by the sorting step [5], giving an overall complexity of $O(|E| \log |E|)$. This is a significant improvement over the exponential time complexity of exhaustive search, allowing the greedy approach to handle larger graphs efficiently.

The space complexity of the greedy algorithm is $O(|E|)$, as it only needs to store the current edge dominating set and the status of covered edges. This efficient space requirement makes it well-suited for large-scale graphs where memory constraints may limit other algorithms' applicability.

C. Summary of Complexity Analysis

The analysis shows that:

- **Exhaustive Search Algorithm**: Exhibits exponential time and space complexity ($O(2^{|E|})$) , making it suitable only for small graphs due to its high computational cost.
- **Greedy Heuristic Algorithm**: Offers a time complexity of $O(|E| \log |E|)$ due to the initial sorting step, and a linear space complexity ($O(|E|)$), making it a scalable alternative for larger graphs, although providing an approximate solution.

D. Time Complexity Prediction

1) Exhaustive Search Algorithm Prediction: Time complexity grows exponentially with the number of vertices, V, and can be expressed as:

$$T_{\text{exhaustive}}(V) \approx O(2^V)$$

Using a known data point (for example, with a graph of 8 vertices and 75% edge density, the execution time was 698.9304 seconds), the coefficient was calculated $C_{\text{exhaustive}}$ to model the exponential growth accurately. The coefficient is calculated as:

$$C_{\text{exhaustive}} = \frac{T_{\text{measured}}}{2^V}$$

For the known data point where $T_{\text{measured}} = 698.9304$ seconds and $V = 10$:

$$C_{\text{exhaustive}} = \frac{698.9304}{2^{10}} \approx 0.6829$$

With this coefficient, the predicted execution time for larger values of V is given by:

$$T_{\text{exhaustive}}^{\text{predicted}}(V) = C_{\text{exhaustive}} \cdot 2^V$$

2) Greedy Algorithm Prediction: The time complexity of the greedy algorithm is generally represented as a function of the number of edges, E, and vertices, V, in the graph. Since the greedy approach typically involves iterating over edges and performing operations proportional to $\log E$, the time complexity approximation is:

$$T_{\text{greedy}}(V, E) \approx O(E \cdot \log E)$$

For a fully connected undirected graph, the number of edges E is:

$$E = \frac{V(V-1)}{2}$$

Substituting E in terms of V, the time complexity for the greedy algorithm becomes:

$$T_{\text{greedy}}(V) \approx O\left(\frac{V(V-1)}{2} \cdot \log\left(\frac{V(V-1)}{2} + 1\right)\right)$$

To calibrate the model for predictions, it is computed an empirical coefficient, C_{greedy}, based on observed data points. The coefficient is given by:

$$C_{\text{greedy}} = \frac{T_{\text{measured}}}{E \cdot \log(E+1)}$$

Using this coefficient, the predicted execution time for the greedy algorithm on larger graphs is estimated as:

$$T_{\text{greedy}}^{\text{predicted}}(V) = C_{\text{greedy}} \cdot \frac{V(V-1)}{2} \cdot \log\left(\frac{V(V-1)}{2} + 1\right)$$

Time Complexity Predictions for Greedy and Exhaustive Search Algorithms are represented in Figure 6

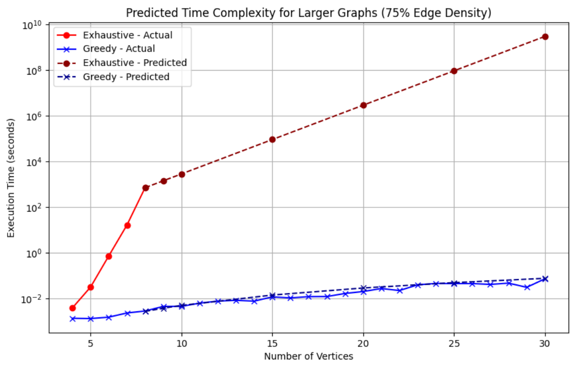


Fig. 6: Predicted Time Complexity - 75% Edge Density

E. Space Complexity Prediction

The space complexity of both algorithms is an important factor in determining their scalability. Following the calculations performed in the previous section, it was possible to estimate the memory requirements for larger graphs. Figure 7 represents the Space Complexity Prediction.

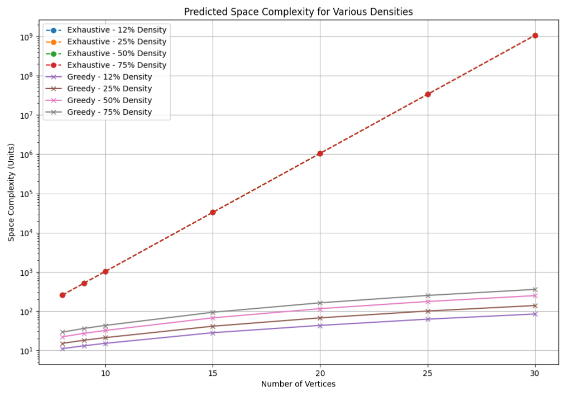


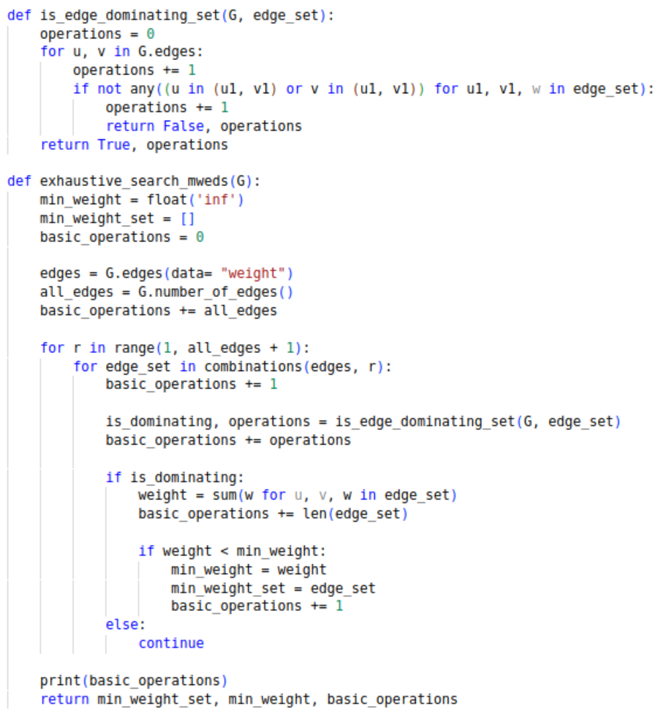
Fig. 7: Predicted Space Complexity

VI. Algorithm Development

A. Exhaustive Search Algorithm

The exhaustive search algorithm explores all possible combinations of edges in the graph to identify a minimum weight edge dominating set. Using the `combinations` function from Python's `itertools` library, the algorithm iterates through increasing subset sizes to find the smallest valid dominating set by weight. The algorithm evaluates each subset using the `is_edge_dominating_set` function, which checks if every edge in the graph is dominated by at least one edge in the subset. If a subset meets this criterion, its total weight is calculated; if this weight is lower than the previously found minimum, the subset is stored as the current optimal solution.

The code snippet for the exhaustive search algorithm is illustrated in Figure 8:



```python
def is_edge_dominating_set(G, edge_set):
    operations = 0
    for u, v in G.edges:
        operations += 1
        if not any((u in (u1, v1) or v in (u1, v1)) for u1, v1, w in edge_set):
            operations += 1
            return False, operations
    return True, operations

def exhaustive_search_mweds(G):
    min_weight = float('inf')
    min_weight_set = []
    basic_operations = 0

    edges = G.edges(data= "weight")
    all_edges = G.number_of_edges()
    basic_operations += all_edges

    for r in range(1, all_edges + 1):
        for edge_set in combinations(edges, r):
            basic_operations += 1

            is_dominating, operations = is_edge_dominating_set(G, edge_set)
            basic_operations += operations

            if is_dominating:
                weight = sum(w for u, v, w in edge_set)
                basic_operations += len(edge_set)

                if weight < min_weight:
                    min_weight = weight
                    min_weight_set = edge_set
                    basic_operations += 1
            else:
                continue

    print(basic_operations)
    return min_weight_set, min_weight, basic_operations
```

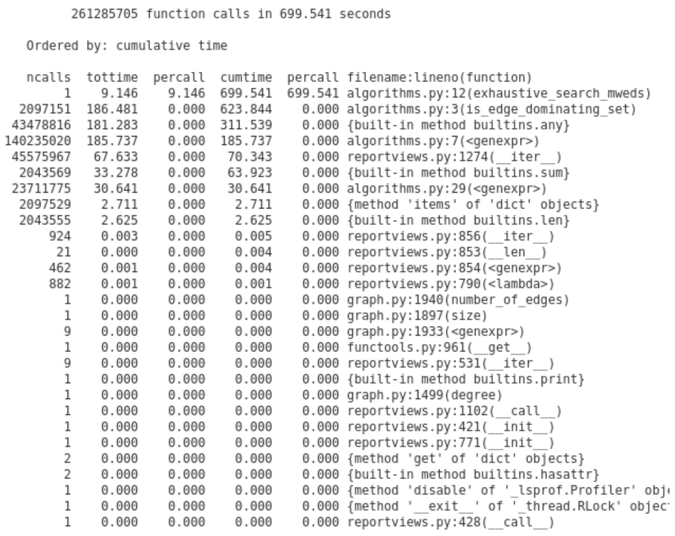
Fig. 8: Revenue Distribution Chart

Table I provides a comparison of both expected time complexity and observed runtimes for the exhaustive search and greedy heuristic algorithms across different graph sizes. The exponential time complexity of the exhaustive search is evident, as each additional node results in a significant increase in computation time. For example, when the exhaustive algorithm was executed on a 6-node graph, the runtime was 0.72 seconds; for 7 nodes, it increased to 16.17 seconds, and for 8 nodes, it escalated to 698.9 seconds. These observed runtimes align with the theoretical exponential growth rate expected for the exhaustive algorithm. In contrast, the greedy heuristic algorithm consistently maintains a much lower time complexity due to its linear selection process, showing only slight increases in runtime as graph size grows.

Nº Nodes	Edges (75% Density)	Exhaustive (s)	Greedy (s)
4	6	0.0038	0.0013
5	10	0.0322	0.0014
6	15	0.7245	0.0016
7	21	16.1784	0.0024
8	28	698.9304	0.0028
9	36	1397.8608	0.0030
10	45	2795.7216	0.0045
15	105	89463.0912	0.0203
20	190	2862818.9184	0.0696
25	300	91610205.3888	0.1926
30	435	2931526572.4416	0.1926

TABLE I: Time Complexity Predictions and Observed Runtimes for Exhaustive Search and Greedy Heuristic Algorithms at 75% Edge Density

As shown in Table I, the exhaustive algorithm's runtime expands dramatically with increased graph size, making it impractical for large graphs. Profiling results, shown in Figure 9, confirm this trend, with each recursive function call increasing with graph size despite optimization attempts. Analyzing the cProfile results it is possible to have a better interpretation of why the Exhaustive Search Algorithm is not as fast as the Greedy solution, "ncalls" represents the number of calls of the discriminated functions, and "cumtime", the cumulative time of the solution.



```
         261285705 function calls in 699.541 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    9.146    9.146  699.541  699.541 algorithms.py:12(exhaustive_search_mweds)
  2097151  186.481    0.000  623.844    0.000 algorithms.py:3(is_edge_dominating_set)
 43478816  181.283    0.000  311.539    0.000 {built-in method builtins.any}
140235020  185.737    0.000  185.737    0.000 algorithms.py:7(<genexpr>)
 45575967   67.633    0.000   70.343    0.000 reportviews.py:1274(__iter__)
  2043569   33.278    0.000   63.923    0.000 {built-in method builtins.sum}
 23711775   30.641    0.000   30.641    0.000 algorithms.py:29(<genexpr>)
  2097529    2.711    0.000    2.711    0.000 {method 'items' of 'dict' objects}
  2043555    2.625    0.000    2.625    0.000 {built-in method builtins.len}
      924    0.003    0.000    0.005    0.000 reportviews.py:856(__iter__)
       21    0.000    0.000    0.004    0.000 reportviews.py:853(__len__)
      462    0.001    0.000    0.004    0.000 reportviews.py:854(<genexpr>)
      882    0.001    0.000    0.001    0.000 reportviews.py:790(<lambda>)
        1    0.000    0.000    0.000    0.000 graph.py:1940(number_of_edges)
        1    0.000    0.000    0.000    0.000 graph.py:1897(size)
        9    0.000    0.000    0.000    0.000 graph.py:1933(<genexpr>)
        1    0.000    0.000    0.000    0.000 functools.py:961(__get__)
        9    0.000    0.000    0.000    0.000 reportviews.py:531(__iter__)
        1    0.000    0.000    0.000    0.000 {built-in method builtins.print}
        1    0.000    0.000    0.000    0.000 graph.py:1499(degree)
        1    0.000    0.000    0.000    0.000 reportviews.py:1102(__call__)
        1    0.000    0.000    0.000    0.000 reportviews.py:421(__init__)
        1    0.000    0.000    0.000    0.000 reportviews.py:771(__init__)
        2    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
        2    0.000    0.000    0.000    0.000 {built-in method builtins.hasattr}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' obj
        1    0.000    0.000    0.000    0.000 {method '__exit__' of '_thread.RLock' objec
        1    0.000    0.000    0.000    0.000 reportviews.py:428(__call__)
```

Fig. 9: Profile of Exhaustive Algorithm Execution Showing Exponential Growth

Table II presents the number of operations performed by both the exhaustive and greedy heuristic algorithms for graphs with different number of nodes and a 75% edge density. The data shows a clear contrast in the number of operations of these two algorithms as the size of the graph increases.

Nº Nodes	Edges (75% Density)	Exhaustive (ops)	Greedy (ops)
4	6	265	15
5	10	3227	21
6	15	76130	29
7	21	1604395	52

TABLE II: Operations Count for Exhaustive Search and Greedy Heuristic Algorithms at 75% Edge Density

Table III presents the number of configurations tested by both the exhaustive and greedy heuristic algorithms for graphs with different number of nodes and a 75% edge density. The data illustrates a significant difference in the number of configurations explored by these two algorithms as the size of the graph increases, with the exhaustive search algorithm testing a much larger number of configurations compared to the greedy heuristic. This highlights the exhaustive nature of the former, which evaluates all possible combinations, while the latter makes decisions more efficiently by iterating through the edges in sorted order.

Nº Nodes	Edges (75%)	Exhaustive (Configs)	Greedy (Configs)
4	6	31	2
5	10	255	3
6	15	4095	5
7	21	65535	4
8	28	2097151	5

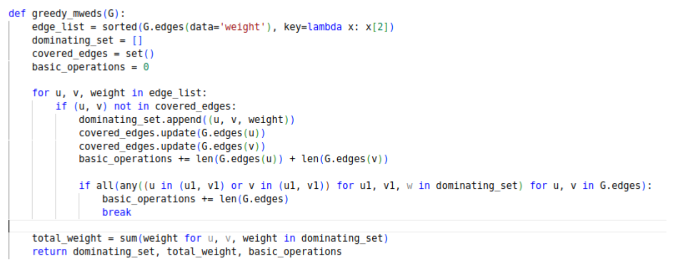
TABLE III: Configurations Count for Exhaustive Search and Greedy Heuristic Algorithms at 75% Edge Density

B. Greedy Heuristic Algorithm

The algorithm proceeds with the following steps:

1) The graph's edges are sorted in ascending order by their weight using the `sorted` function.
2) The algorithm initializes an empty list `dominating_set` to store the edges that will form the dominating set, and an empty set `covered_edges` to track the edges that have been covered by the selected edges.
3) For each edge (u, v) in the sorted edge list, it checks whether the edge is already covered by the edges in `covered_edges`. If not, the edge is added to the dominating set and the edges incident to the vertices u and v are marked as covered.
4) The algorithm uses the `G.edges(u)` and `G.edges(v)` functions to get all the edges incident to the nodes u and v, respectively, and updates the `covered_edges` set to ensure these edges are marked as covered.
5) The algorithm continues iterating through the sorted edge list, adding edges to the dominating set until all edges in the graph are covered.
6) The loop terminates when all edges in the graph are covered, and the total weight of the selected dominating set is calculated by summing the weights of the edges in the set.

The code snippet for the greedy search algorithm is illustrated in Figure 10:



```python
def greedy_mweds(G):
    edge_list = sorted(G.edges(data='weight'), key=lambda x: x[2])
    dominating_set = []
    covered_edges = set()
    basic_operations = 0

    for u, v, weight in edge_list:
        if (u, v) not in covered_edges:
            dominating_set.append((u, v, weight))
            covered_edges.update(G.edges(u))
            covered_edges.update(G.edges(v))
            basic_operations += len(G.edges(u)) + len(G.edges(v))

            if all(any((u in (u1, v1) or v in (u1, v1)) for u1, v1, w in dominating_set) for u, v in G.edges):
                basic_operations += len(G.edges)
                break

    total_weight = sum(weight for u, v, weight in dominating_set)
    return dominating_set, total_weight, basic_operations
```

Fig. 10: Revenue Distribution Chart

Figure 11 illustrates the accuracy of the greedy algorithm by comparing its results with the optimal solution obtained from exhaustive search. While the total weight achieved by the greedy approach closely approximates the optimal weight, the discrepancy reflects the heuristic's limitations in certain configurations.

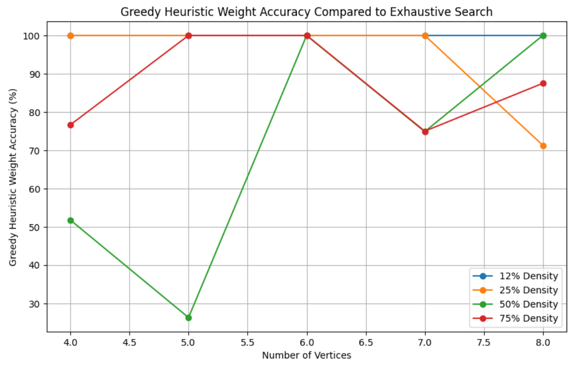


Fig. 11: Accuracy of the Greedy Heuristic Algorithm Compared to the Exhaustive Solution

VII. Code Organization

To go along with this report which results were developed in a GitHub repository, there will be a set of files organized in a set of folders, namely:

- **graphics:** contains the folders with the different visualizations
- **graphs:** contains the original graphs created
- **graphs_solution:** contains the graphs with the solution marked in red
- **results:** contains multiple csv's with the metrics analyzed in this problem
- **main.py:** main function file
- **graph_creation.py:** graph creation file
- **algorithms.py:** file that contains the Exhaustive Search and Greedy Heuristic Algorithms
- **analysis.py:** file that performs the different needed visualizations.

VIII. Conclusion

Through experimental analysis, it was demonstrated the trade-offs between solution precision and algorithm efficiency, shedding light on the scalability of both approaches for various graph sizes and densities. The results underscore the potential of heuristic methods for solving MWEDS in large graphs within reasonable computational time frames, while also pointing to the significant computational effort required for exhaustive search solutions.

References

[1] S. Bouamama and C. Blum, "The Minimum Weight Dominating Set Problem," Mayra Albuquerque, 2018. Retrieved from: https://arxiv.org/pdf/1808.09809

[2] M. Yannakakis and F. Gavril, "Edge dominating sets in graphs," Discrete Applied Mathematics, 1987. Retrieved from: https://www.sciencedirect.com/science/article/pii/S0166218X00003838

[3] M. Yannakakis and F. Gavril, "Edge Dominating Sets in Graphs," SIAM Journal on Applied Mathematics, vol. 38, no. 3, pp. 364–372, 1980. Retrieved from: https://doi.org/10.1137/0138030

[4] Dr. Mallikarjun S Patil, "A STUDY ON COMPREHENSIVE ANALYSIS OF EDGE DOMINATION IN GRAPHS" pp. 34-35, 2023. Retrieved from: https://pijaar.org/wp-content/uploads/2024/02/010206.pdf

[5] Dr. Mallikarjun S Patil, "Python Sort Algorithms: A Comprehensive Guide" September 7, 2023. Retrieved from: https://ioflood.com/blog/python-sort-algorithms/