Exhaustive and Greedy Search to Determine the Existence of an Edge Cover in an Undirected Graph

Liliana Ribeiro, 108713

***Resumo* – Dado um grafo não direcionado, será que ele possui um edge cover com k arestas? Para responder a essa questão, foram implementados dois algoritmos de pesquisa distintos: um algoritmo exaustivo e uma heurística gulosa. Ao longo deste relatório, é analisada e comparada a eficiência de ambos os métodos na determinação da existência de um edge cover para o problema proposto.**

***Abstract* - Given an undirected graph, does it have an edge cover with k edges? To answer this question, two different search algorithms were implemented: an exhaustive search algorithm and a greedy heuristic. This report analyzes and compares the efficiency of both methods in determining the existence of an edge cover for the given problem.**

***Abstract* – Python, Greedy Search, Exhaustive Search, Graph, Execution Time, Computational complexity.**

I. INTRODUCTION

This project was designed to answer the question:

> "For a given undirected graph G(V, E), with n vertices and m edges, does G have an edge cover with k edges? An edge cover is a set C of edges such that each vertex of G is incident to, at least, one edge in C."

The values for *k* are set at 12.5%, 25%, 50%, and 75% of the total number of edges.

To answer this question, the project is divided into three main phases: first, graph generation; second, exhaustive algorithm; third, greedy heuristic; and finally, computational results from the algorithms are gathered and analyzed.

II. GRAPHIC GENERATION

A. How to use the Script

The graph generation process uses the networkx library to assist in creating the graphs. This script requires the desired number of vertices for the graph as an argument. Additionally, there is an optional image argument that can be included if the user wishes to obtain a PNG visualization of the generated graph. Here's an example of how to use this script:

```
$ python3 graph_generator.py <vertices>
[image]
```

B. Logic and Results

The graphs are 2D with *m* vertices. The vertices are plotted within an xOy plane, and points can be located anywhere from 0 to 1000 on each axis. These coordinates are generated using a specific seed (student number - *"número mecanográfico"*) to ensure controlled randomness.

Each graph has a specific number *n* of edges. This number is calculated based on the maximum possible edges for the graph size, creating four distinct graphs with the same number of vertices but with 12.5%, 25%, 50%, and 75% of the maximum edge count. The edges are randomly distributed among the vertices, and each vertex's number of edges is also randomized.

Each generated graph is saved as a JSON file with a title format like graph_<vertex_number>_<edge_prob>.json. The data is stored in an adjacency dictionary format, where each key represents a vertex's coordinates, and the associated value is a list of vertices it is connected to by edges. Below is an example of this representation for the file graph_004_50.json, which contains a graph with 4 vertices and 3 edges, along with its corresponding visualization.

```
{
      "(745, 997)": [],
      "(331, 617)": [
          "(665, 355)",
          "(400, 551)"
      ],
      "(400, 551)": [
          "(331, 617)",
          "(665, 355)"
      ],
      "(665, 355)": [
          "(331, 617)",
          "(400, 551)"
      ]
}
```

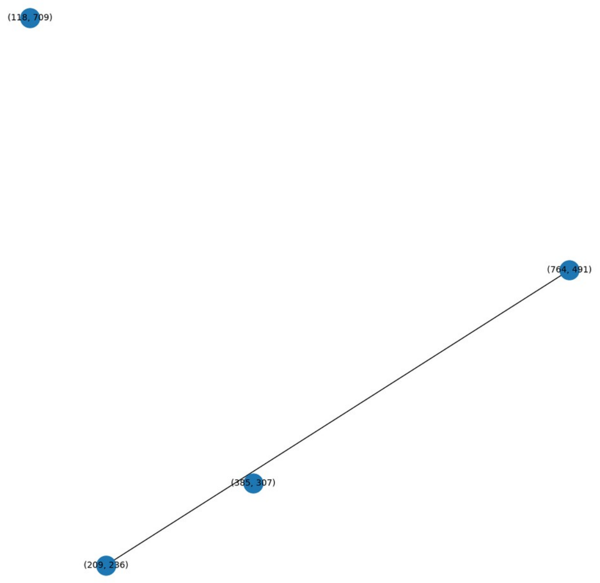

Fig. 1 – Graph with 4 vertices and 50% edges

III. EXHAUSTIVE ALGORITHM

A. Algorithm Description

The exhaustive search algorithm employs a systematic approach to find a k-edge cover by examining all possible combinations of k edges from the graph. For each combination, it verifies whether it forms a valid edge cover.
In pseudo-code, the algorithm operates as follows:

```
1. Generate all possible combinations of k
   edges from the edge set E
2. For each combination:
   • Create a set of vertices covered by
     the selected edges
   • Verify if all vertices in the graph
     are incident to at lease selected
     edge
3. Return the first valid edge cover found,
   or indicate that no solution exists
```

B. Formal Analysis

The exhaustive search algorithm checks every possible subset of k edges to see if it forms a valid edge cover, only stopping when it reaches a valid solution.
For a graph with *m* edges and *k* being the desired edge cover size, there are *C(m,k)* possible combinations, and the algorithm searches through each one of them.
After, for each combination of *k* edges, to check if it forms a valid edge cover we need to:

1. Create a set of covered vertices from the *k* edges - *O(k)*
2. Check if all n vertices of the graph are covered by at least one edge in our set - *O(n)*

This means that for each combination, we perform *O(k + n)* operations.

This yields a total time complexity of *O(C(m,k) * (k + n))*, where:

- *m* is the total number of edges in the graph
- *k* is the size of edge cover we're looking for
- *n* is the number of vertices

When *k* is expressed as a fraction *α* of *m* (*k* = *αm*), this becomes exponential, leading to a complexity of approximately ***O(2^m * (m + n))***.

IV. GREEDY ALGORITHM

A. Algorithm Description

As we've seen, the exhaustive algorithm takes too long to execute. To address this computational problem, a different algorithm was developed. A greedy algorithm is one that follows the heuristic of making the locally optimal choice at each stage of the problem-solving process. While a greedy strategy doesn't always produce a globally optimal solution, it can often yield locally optimal solutions that approximate the global solution in a reasonable amount of time.

In this case, the greedy algorithm iteratively selects edges that cover the largest number of uncovered vertices until k edges are selected or all vertices are covered. This is an effective strategy because it allows us to quickly find an optimal solution, if one exists. If there is a k edge cover solution, the edges covering the most uncovered vertices are the most likely to be part of that solution. If none of these edges pass the test, we can quickly conclude that a k edge cover solution does not exist, as some vertices will remain uncovered.

In pseudo-code, the algorithm works as follows:

```
1. Initialize an empty edge cover and a set
   of uncovered vertices
2. While the edge cover size is less than k
   and vertices remain uncovered:
   • For each available edge, calculate
     how many uncovered vertices it would
     cover
   • Select the edge that covers the
     maximum number of uncovered vertices
```

```
        • Add the selected edge to the cover
          and update the set of uncovered
          vertices
3. Return the edge cover if all vertices are
   covered, or indicate failure
```

B. Formal Analysis

This algorithm is considerably more efficient and faster. To formally analyze its complexity, we need to calculate the computational effort for each step, excluding constant-time steps that don't significantly impact the overall computational cost.

The main loop is the core of the algorithm, running up to k times, where k is the target size of the edge cover. In each iteration, the algorithm performs the following actions:

1. Edge Evaluation: All remaining edges are evaluated to find the one covering the highest number of uncovered vertices — this takes *O(m)*, where *m* is the total number of edges.
2. Vertex Coverage Check: For each edge, we determine how many uncovered vertices it would cover — this step takes constant time, *O(1)*.
3. Updating Sets: After selecting the optimal edge, we update the sets of covered vertices and available edges — also *O(1)*.

Considering these steps, the algorithm's approximate upper bound for time complexity can be calculated as follows:

- The main loop runs at most *k* times.
- Each iteration examines up to *m* edges.
- Each edge evaluation requires *O(1)* time.

This leads to an overall complexity of *O(k×m), where*

- *m* is the total number of edges in the graph
- *k* is the size of edge cover we're looking for

When *k* is expressed as a fraction *α* of *m* (where *k*=*αm*), the complexity simplifies to a polynomial ***O(m²)***. This is a substantial improvement over the exponential complexity of exhaustive search.

Additionally, the algorithm's performance is influenced by the graph's edge density. It will perform better on graphs with less edges, because fewer edges are evaluated in each iteration, than on dense graphs with a higher edge count.

V. EXPERIMENTAL ANALYSIS

A. Method

To calculate the experimental values for both solutions, graphs with 4 to 100 vertices were used, with edge densities set at 12.5%, 25%, 50%, and 75% of the maximum possible edges.

For each graph, the existence of an edge cover of size k was determined using both algorithms. To avoid excessively long computation times, especially for larger graphs, a timeout mechanism was implemented: if the execution time for processing a graph with n vertices exceeded 1 minute, the process is terminated. The value of n at which this occurs was noted as the maximum number of vertices that the algorithm could process for a given edge density. This approach was particularly necessary for the exhaustive algorithm, as its execution time grows exponentially, quickly increasing from 1 minute to 10 minutes, and so on.

For each algorithm, the results were analyzed by plotting the number of vertices on the *x-axis* and the following metrics on the *y-axis*: execution time, configurations tested, and basic operations. In these graphs:

- *Density* represents the percentage of edges relative to the maximum number of edges possible for a graph with n vertices.
- *k* represents the percentage of edges in the edge cover relative to the total number of edges in the graph.

Across all these graphs, a parallelism was observed between execution time, configurations tested, and basic operations. This correlation makes sense, as the number of basic operations is directly related to the configurations tested, and the execution time is directly influenced by the number of operations performed.
Below is an example of these different graphs, to exhibit their correlation, for the greedy algorithm with k = 0.125:

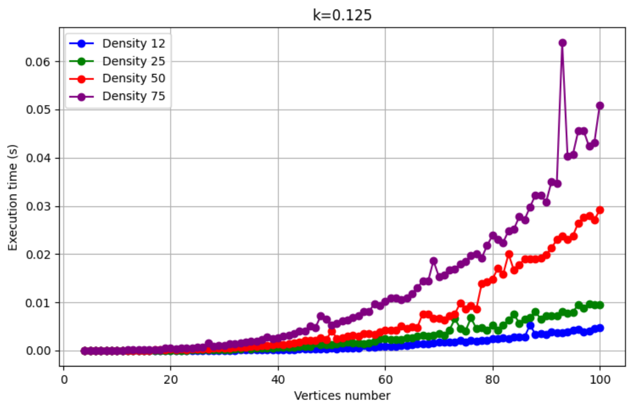

Fig. 2 – Results of Greedy Algorithm Graph with k = 0.125 for execution time.

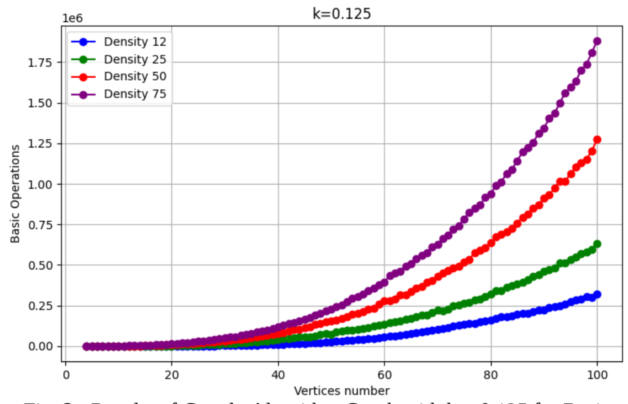

Fig. 3 - Results of Greedy Algorithm Graph with k = 0.125 for Basic Operations.

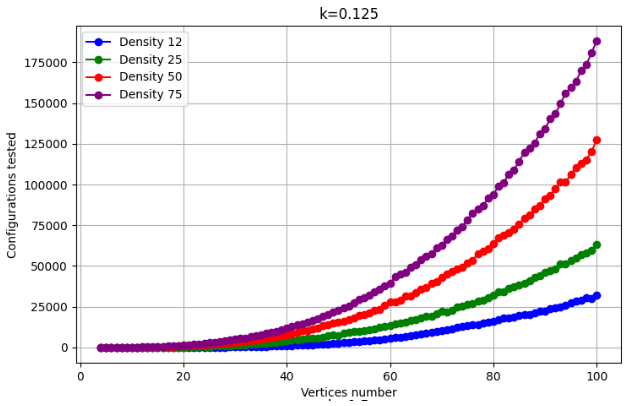

Fig. 4 - Results of Greedy Algorithm Graph with k = 0.125 for Tested Configurations.

It was noted that, regardless of the algorithm, the execution time graph exhibited variances not justified by formal analysis. These variances may be explained by external factors, such as system load. Since the execution time values are directly proportional to the configurations tested and basic operations, the number of basic operations will be used to analyze the behavior and complexity of the algorithms more accurately.

B. Exhaustive Algorithm Analysis

The exhaustive algorithm demonstrates the behavior predicted in the formal analysis, showing exponential growth in execution time.

As expected, this growth sometimes exhibits unpredictable variations, as exponential growth represents only the Big-O of the worst case. These fluctuations occur when the algorithm finds a solution without having to test all possible combinations of k edges, reducing execution time. In other words, the algorithm may, by chance, find an ideal edge combination in the early stages of the search.

For the graph with k = 0.125 (Fig. 5), the exponential curve is more apparent, as this case frequently results in worst-case scenarios. With a low k value, the algorithm often needs to test all possible combinations without finding a valid solution. In the graph for k = 0.50 (Fig. 6), the fluctuations in values are more noticeable, reflecting the randomness in execution times. Both cases are illustrated in the figures below.

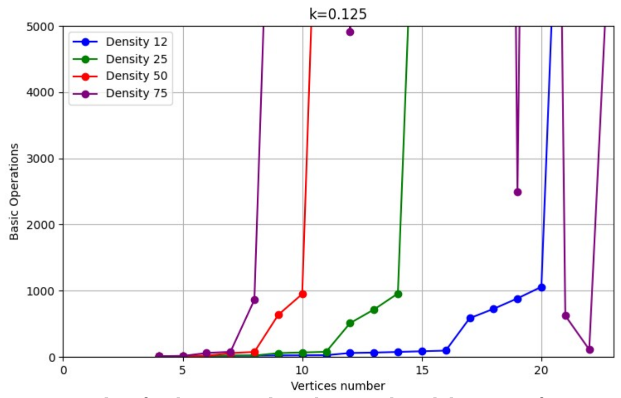

Fig. 5 - Results of Exhaustive Algorithm Graph with k = 0.125 for Basic Operations

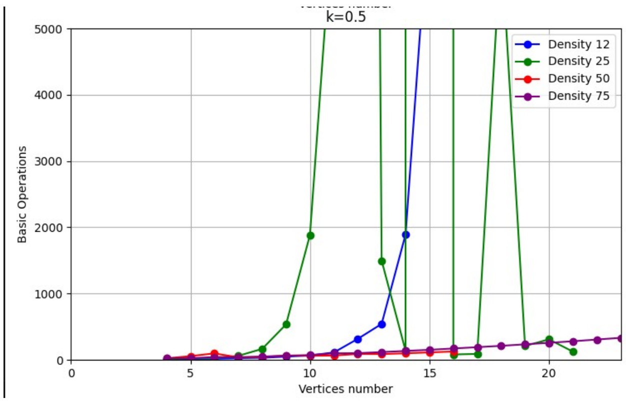

Fig. 6 - Results of Exhaustive Algorithm Graph with k = 0.50 for Basic Operations.

It can also be observed from k=0.125 (Fig. 5) that as the graph density increases, the algorithm requires more basic operations. This is because higher density implies more edges, and therefore more combinations to test.

Below are the graphs of execution time (Fig. 7) and configurations tested (Fig. 8) for the exhaustive algorithm with *k* = 0.125.

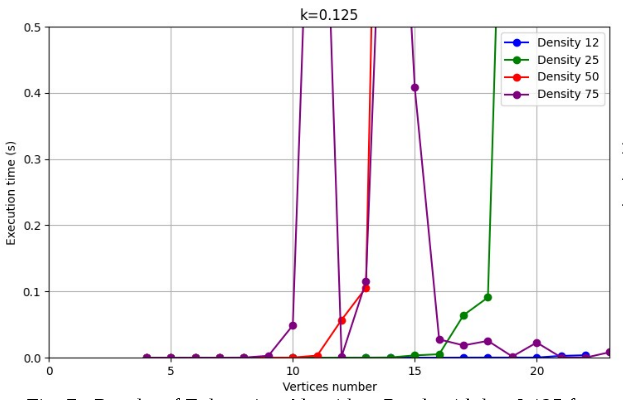

Fig. 7 - Results of Exhaustive Algorithm Graph with k = 0.125 for Execution Time.

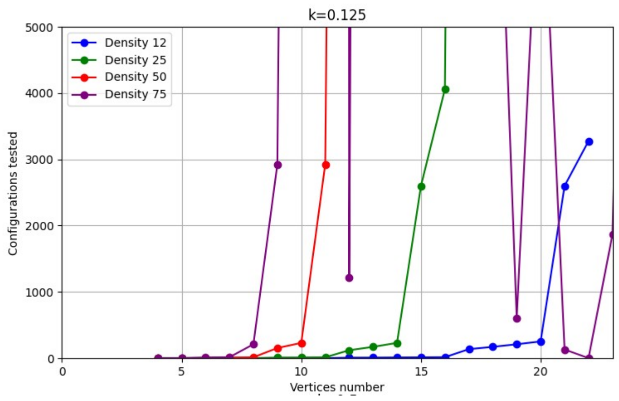

Fig. 8 - Results of Exhaustive Algorithm Graph with k = 0.125 for Tested Configurations.

The performance limits for graphs with densities of 12.5%, 25%, and 50% were reached quickly, at 22, 21, and 15 vertices, respectively. This behavior is expected: beyond these values, the computational cost of the worst case becomes significant. In these situations, especially for k = 0.125, where no small edge cover exists, the algorithm must test all possible combinations, resulting in extreme computational effort. Below is an example for a graph with 15 vertices, 50% density, and k = 0.125:

```
"execution_time": 30.939193964004517,
"configs_tested": 20358520,
"basic_operations_count": 81434128
```

In contrast, for graphs with 75% density, the exhaustive algorithm performs much faster. In these cases, the probability of finding a *k* edge cover solution is higher, reducing the likelihood of reaching the worst case. For 75% density graphs, the algorithm reached the limit of 100 vertices without difficulty. In this scenario, the growth observed did not follow the exponential pattern of Big-O but instead exhibited more of a polynomial shape, as illustrated in the figure below.

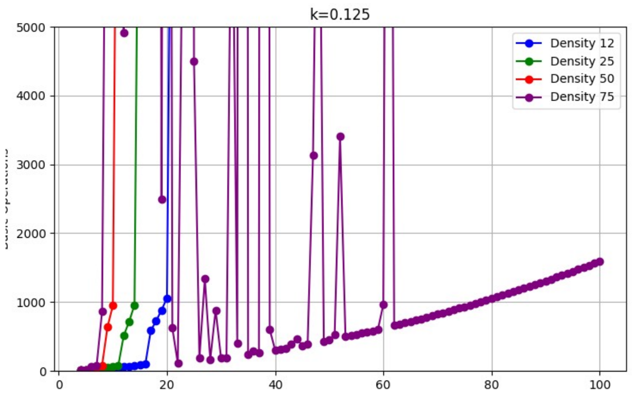

Fig. 9 - Results of Exhaustive Algorithm Graph with k = 0.125 for Basic Operations with y-axis set at 100 vertices.

C. Greedy Algorithm Analysis

An analysis of the greedy algorithm reveals results consistent with the formal analysis, showing a polynomial curve (Fig. 10) . As expected, this curve varies with graph density. For lower densities, the algorithm has lower computational complexity, as it has fewer edges to evaluate.

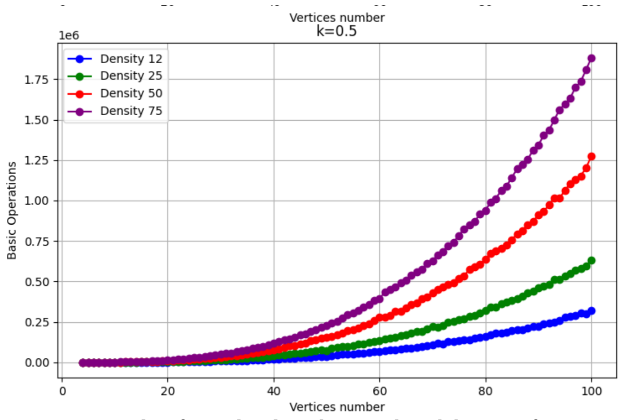

Fig. 10 - Results of Greedy Algorithm Graph with k = 0.50 for Basic Operations.

Additionally, four graphs were generated, one for each density level, with lines representing different values of k (Fig. 11) - unlike the previous figures. These graphs were created to investigate any potential changes in the algorithm's behavior when searching for an edge cover with different k values.

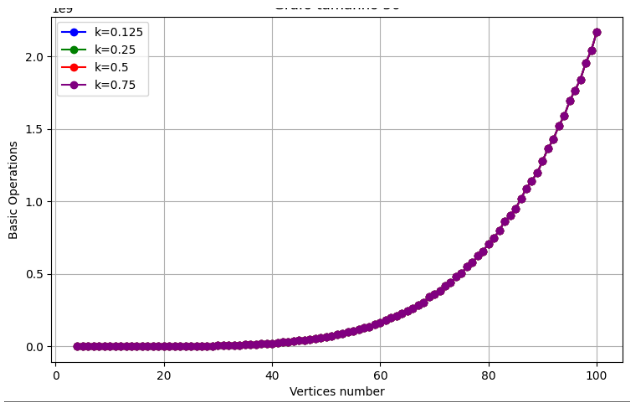

Fig. 11 - Results of Greedy Algorithm Graph with density = 50% for Basic Operations.

The results show that there is no significant difference in the computational effort required to find an edge cover across different values of k. The value of k does not affect execution time because the algorithm still needs to follow the same edge evaluation and selection process, which is driven by the density and structure of the graph.

This algorithm proved to be highly efficient, successfully processing all requested graphs in a very short time. The graph with the longest execution time took only 58 milliseconds.

Below are the execution time (Fig. 12) and decisions made (Fig. 13) graphs for the greedy algorithm with k = 0.5.

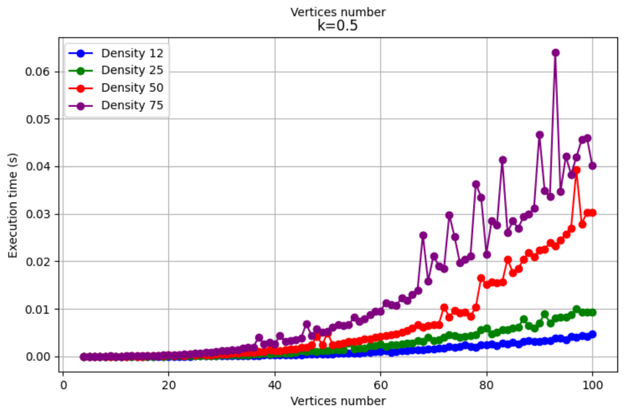

Fig. 12 - Results of Greedy Algorithm Graph with k = 0.5 for Execution Times.

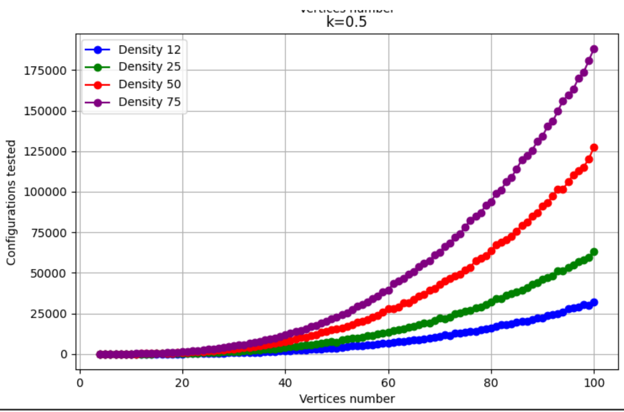

Fig. 13 - Results of Greedy Algorithm Graph with k = 0.5 for Tested Configurations.

A script was also created to analyze the number of False Positives and False Negatives produced by the greedy algorithm in order to calculate its accuracy. The following values were obtained:

- False Positives: 0
- False Negatives: 6

The absence of false positives demonstrates the algorithm's accuracy, as a false positive could imply mistakenly identifying a correct solution as incorrect, which could lead to further computational steps or misinterpretations. Regarding the false negatives, 6 out of 400 graphs resulted in a 1.5% error rate, which is considered acceptable given the algorithm's speed. Additionally, it was examined whether factors such as density, the value of k, or the number of vertices influenced these results; however, no conclusive patterns were found, suggesting that false negatives are obtained randomly.

D. Algorithms Comparison

As observed, the greedy algorithm significantly outperforms the exhaustive algorithm. This relationship is demonstrated in the graph below, which shows the number of basic operations performed by each algorithm as a function of the number of vertices, with a graph density of 25% and k = 0.25.

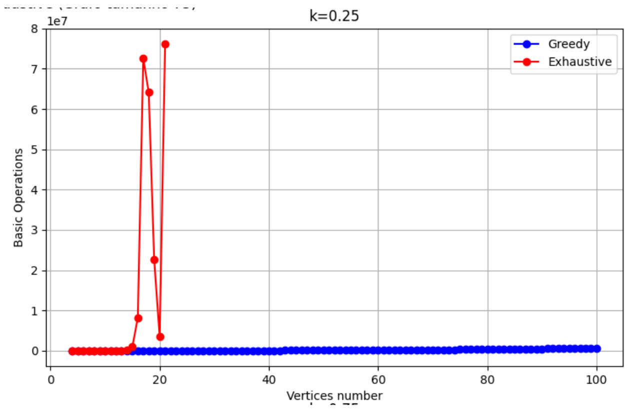

Fig. 14 - Results of Algorithm Comparison with k = 0.25 and Density = 25% for Basic Operations.

This trend holds for graphs with densities of 12.5%, 25%, and 50%. However, for graphs with a density of 75%, this pattern changes drastically, with the exhaustive algorithm becoming the faster option. This shift occurs because, as explained earlier, when there are many possible edge covers, the exhaustive algorithm can often find a valid solution quickly, as the first combinations it tests tend to be correct. Conversely, the greedy algorithm takes longer on denser graphs because it has more edges to evaluate.

Below, this tendency is illustrated in the graph showing the basic operations as a function of the number of vertices for graphs with a 75% density and k = 0.25.

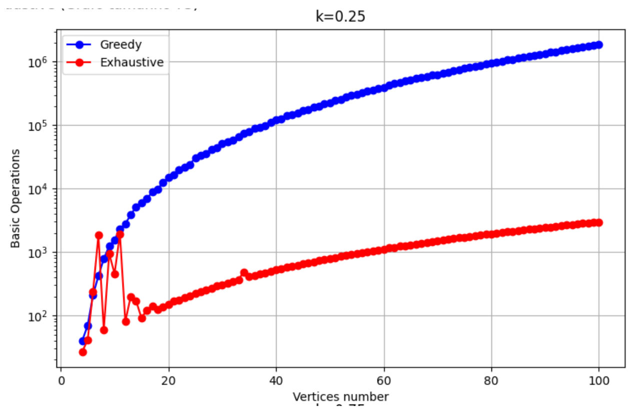

Fig. 15 - Results of Algorithm Comparison with k = 0.25 and Density = 75% for Basic Operations.

E. Execution Time for larger problems

For larger problems, the algorithms yield different performance results. A small script was created to calculate the linear constant between the number of operations performed and the execution time, providing an average of how many operations were performed per second during the execution of these algorithms. After running this script, we obtained a rate of approximately 10^7 operations per second.

To estimate the time required for the exhaustive algorithm to handle one of the largest given graphs (100 vertices with 50% density), let's go through the calculation:

Data:

- `Complexity: O(2m×(m+n))`
- `n=100 vertices`
- `m=2,500 edges`
- `k=0.5m=1,250 (50% of the edges)`

Calculation:

1. `Combinations: C(2500,1250)≈10^375`
2. `Operations per Combination: O(100+1250)=O(1350)`
3. `Total Complexity: O(10^375×1350)`

The estimated execution time is approximately 10^(375−7)=10^369 seconds, which is equivalent to 10^361 years. In contrast, for the same problem, the greedy algorithm takes only 41 milliseconds.

To test execution times for a large problem using the greedy algorithm, we use a graph with 10,000 vertices and 50% density.

Data:

- `Complexity: O(m²)`
- `n= 10 000 vertices`
- `m= 24 997 500 edges`

Calculation:

1. `Operations total: 1.25 * 10^12`
2. `Estimated execution time = 1.25 * 10¹² / 10⁷ = 125.000 secondes ≈ 34,72 hours`

This demonstrates that even with very large graphs, the greedy algorithm can analyze and provide an answer in a relatively short time.

V. CONCLUSIONS

This study demonstrates the stark contrast in efficiency between the exhaustive and greedy algorithms when determining the existence of an edge cover in undirected graphs. While the exhaustive algorithm systematically explores all possible solutions, leading to exponential complexity that renders it impractical for large graphs, the greedy algorithm offers a much faster alternative with polynomial complexity. The greedy approach proves especially effective on larger graphs, where it can handle thousands of vertices in a reasonable time frame. Ultimately, the findings suggest that for practical applications, the greedy algorithm provides a robust and efficient solution, particularly when dealing with high-density or large-scale graphs.

REFERENCES

[1] https://chatgpt.com/

[2] https://en.wikipedia.org/wiki/Edge_cover

[3] https://en.wikipedia.org/wiki/Greedy_algorithm

[4] https://en.wikipedia.org/wiki/Brute-force_search