Randomized 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, adicionalmente aos dois algoritmos de pesquisa já implementados: um algoritmo exaustivo e uma heurística gulosa, será implementado uma algoritmo de pesquisa aleatória. Ao longo deste relatório, é analisada e comparada a eficiência dos 3 métodos na determinação da existência de um edge cover para o problema proposto.**

***Abstract* - Given an undirected graph, does it have edge coverage with k edges? To answer this question, in addition to the two search algorithms already implemented: an exhaustive algorithm and a greedy heuristic, a random search algorithm will be implemented. Throughout this report, the efficiency of the 3 methods in determining the existence of edge coverage for the proposed problem is analyzed and compared.**

***Abstract* – Python, Randomized Search, Greedy Search, Exhaustive Search, Monte Carlo Algorithm, 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.

This article continues the work developed in the first project of the Advanced Algorithms course, where the computational results of the Exhaustive Search and Greedy Search approaches were analyzed for the given problem.

Following a structure similar to the previous article, this work begins with a detailed explanation of the problem and the generation of the graphs used. Next, it provides a summary of the algorithms previously studied before introducing the logic behind the new algorithm: Randomized Search. Finally, the results of the new algorithm will be analyzed and compared with the previous methods, highlighting advantages and limitations.

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)"
        ]
}
```


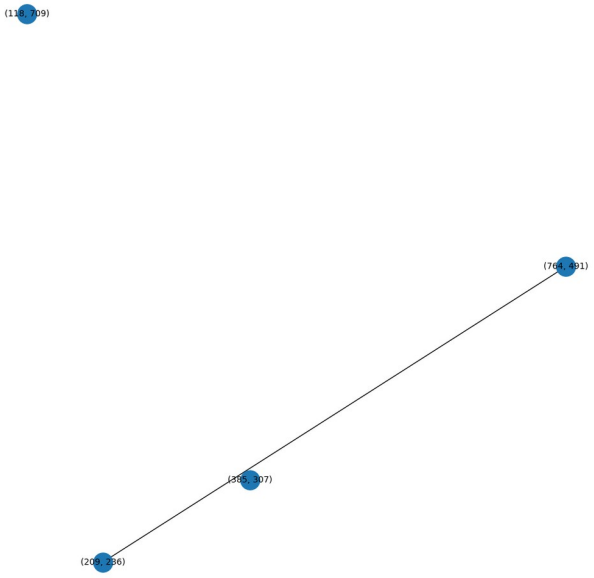

Fig. 1 – Graph with 4 vertices and 50% edges

III. ALGORITHMS ALREADY DEVELOPED

A. Exhaustive Search

The exhaustive search algorithm systematically explores all possible combinations of k edges from the graph to determine whether they form a valid edge cover. As a brute-force approach, it examines every edge combination, leading to an exponential computational cost of ***O(2^m * (m + n))***. This means that as the number of vertices and edges increases, the time required to find a solution grows significantly. In previous tests, the algorithm was able to handle up to 20 edges without excessive runtime.

On the other hand, for graphs with high density, this algorithm performs well. Dense graphs typically have numerous valid solutions, and the exhaustive algorithm only needs to find one. If a solution is discovered early in the process, the algorithm avoids the worst-case scenario of exponential growth. In the study case, when the graph had 75% of edge configuration, the exhaustive search performed efficiently.

B. Greedy Algorithm

A greedy algorithm is a heuristic-based approach that makes locally optimal choices at each stage of the problem-solving process. In this case, the greedy algorithm iteratively selects edges that cover the largest number of uncovered vertices until either k edges are selected or all vertices are covered. This strategy is effective because it quickly identifies a potential solution, if one exists. If a k-edge cover solution is possible, the edges that cover the most uncovered vertices are likely to be part of it. Conversely, if no edges meet the criteria, the algorithm can efficiently determine that a k-edge cover solution does not exist, as some vertices will remain uncovered.

This algorithm works well for various types of graphs, with a polynomial growth complexity of $\boldsymbol{O(m^2)}$ relative to the number of vertices. It performs especially efficiently on graphs with lower density. In the study case, the algorithm yielded excellent results, with the maximum runtime for a graph search being only 58 milliseconds for graphs ranging between 0 and 100 vertices.

However, as a heuristic, the greedy algorithm has limitations. It can occasionally produce errors due to its nature. In this study, the algorithm demonstrated a 1.5% error rate, exclusively generating false negatives, where valid solutions were missed. Despite this drawback, the algorithm remains a robust and efficient choice for many scenarios.

IV. RANDOMIZED ALGORITHM

A randomized algorithm introduces probabilistic strategies to solve computational problems, leveraging randomness to explore solution spaces more efficiently. In this case, the randomized algorithm builds upon the greedy algorithm's foundation by classifying uncovered vertices and assigning probabilities to edges based on how likely they are to contribute to the solution (edges covering the most uncovered vertices).

Unlike greedy algorithms, which always select the definitively best edge, this approach introduces controlled randomness by assigning probabilities proportional to the edge's potential impact. This randomness allows the algorithm to escape potential local optima, increasing its ability to find solutions in complex graphs.

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 uncovered vertices remain:
   - For each available edge, calculate how many uncovered vertices it would cover.
   - Assign each edge a probability proportional to the number of uncovered vertices it addresses.
   - Randomly select an edge using these probabilities.
   - Add the selected edge to the edge cover and update the set of uncovered vertices.
3. Repeat the process for a fixed number of iterations or until a solution is found.
4. Return the edge cover if all vertices are covered and the size constraint (k) is satisfied; otherwise, indicate failure.

This algorithm can be classified as a Monte Carlo algorithm, as it employs randomness to explore the solution space and offers a probabilistic guarantee of finding a solution, rather than a deterministic one.

B. Formal Analysis

The randomized algorithm's complexity depends on the number of edges (*m*), vertices (*n*), and the maximum number of iterations (*max_iterations*). The steps involved in each iteration are as follows:

1. Edge Evaluation: For each edge, the algorithm calculates how many uncovered vertices it would cover, which takes *O(m)*.
2. Probability Assignment: Probabilities for edges are computed based on their density, which requires summing the edges *O(m)* and normalizing them.
3. Edge Selection: A random choice is made based on probabilities, which takes *O(m)*.
4. Updating Sets: The algorithm updates the set of uncovered vertices and removes the selected edge, which is *O(1)* per edge.

Since the main loop repeats up to *max_iterations* times, the overall time complexity is: ***O(max_iterations*m)***, which corresponds to a linear function.

V. EXPERIMENTAL ANALYSIS

A. Method

The method used is the same as the one employed in the first project:

To evaluate the performance characteristics of both algorithmic solutions, experiments were conducted using graphs ranging from 4 to 100 vertices, 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.

A consistent parallelism was observed across all graphs between execution time, configurations tested, and basic operations, which is expected since the number of basic operations is directly tied to the configurations tested, and execution time is influenced by the operations performed. Below are example graphs illustrating this correlation for the greedy algorithm with k = 0.125.

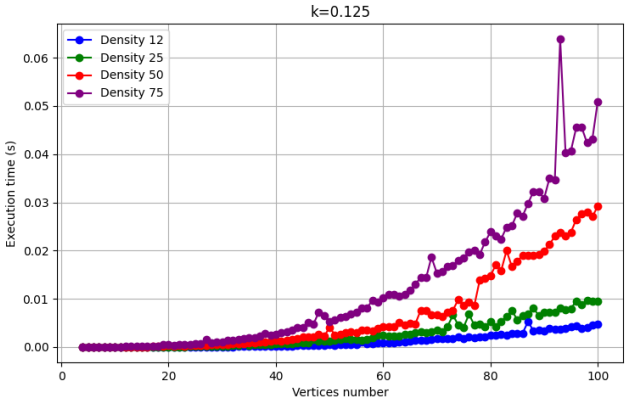

Fig. 2 – Results of Greedy Algorithm Graph with k = 0.125 for execution time.

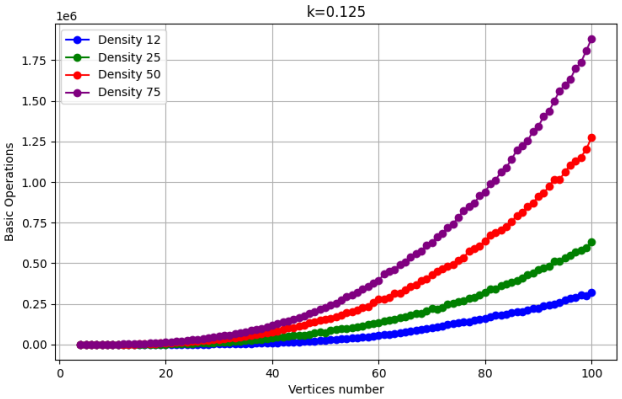

Fig. 3 - Results of Greedy Algorithm Graph with k = 0.125 for Basic Operations.

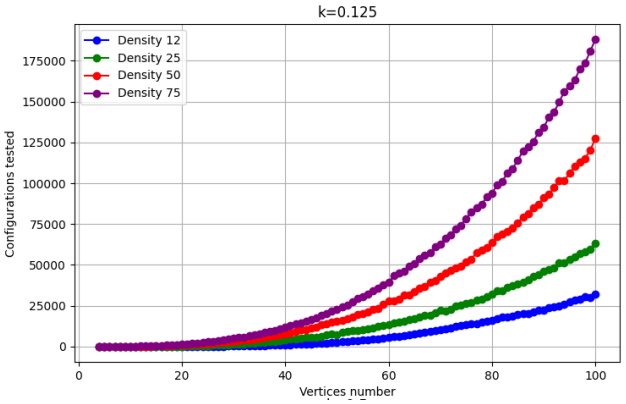

Fig. 4 - Results of Greedy Algorithm Graph with k = 0.125 for Tested Configurations.

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. Randomized Algorithm Analysis

In the analysis of this algorithm, the results aligned with the expectations. The formal analysis confirmed a linear growth of ***O(max_iterations·m)***, representing the worst-case scenario (Big-O complexity) for this algorithm.

However, the observed values indicate that the worst case was not consistently achieved. The worst-case scenario occurs when the algorithm returns a result of "False," as it must exhaustively test all edge combinations to conclude that no valid edge cover exists.

This algorithm is primarily designed to find an edge cover efficiently and is less focused on optimizing performance for cases where no edge cover exists. As a result, we observe significant discrepancies in computational effort between graphs where a valid edge cover is found ("Success" cases) and those where no solution exists ("Failure" cases). These discrepancies are particularly noticeable, as "Success" cases often result in very low computational times, while "Failure" cases take significantly longer.

In Fig. 5, this behavior is illustrated using the k=0.125 graph. For lower values of k, it becomes less likely to exist a valid edge cover, making this graph an ideal representation of the linear trend.

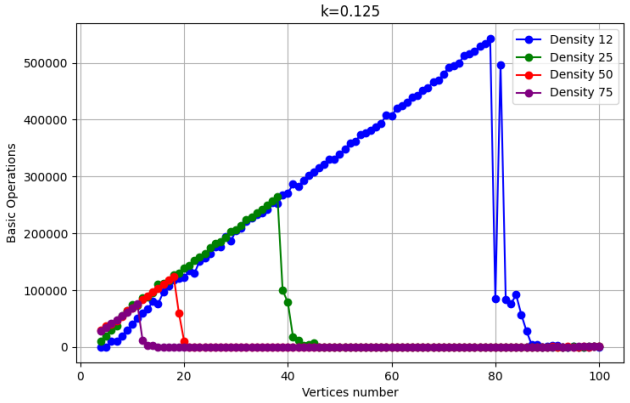

Fig. 5 – Results of Randomized Algorithm Graph with k = 0.125 for Basic Operations.

In this figure we can see that the significant discrepancy between "Success" and "Failure" cases is evident, with computational times for "Success" cases approaching zero. Denser graphs also tend to reach valid solutions ("Success") more quickly.

As k increases, and more valid solutions exist, the linear tendency becomes less visible. Instead, we observe greater variability, with jumps between computational times for "Success" and "Failure" results. This trend is better visualized in the k=0.75 graph (Figure 6).

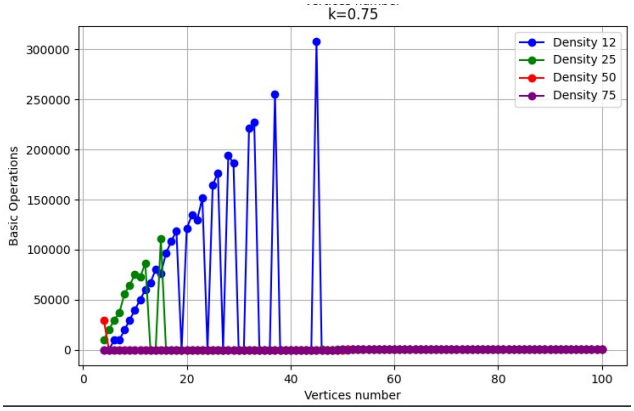

Fig. 6 – Results of Randomized Algorithm Graph with k = 0.75 for Basic Operations.

To better illustrate the extreme differences in computational effort between "Success" and "Failure" results, an additional graph was created, focusing on the k=0.75 graph (Fig. 6) for Density 12. In this graph (Fig. 7), we can see the linear tendency on the “Failure” line and the “Success” line close to zero.

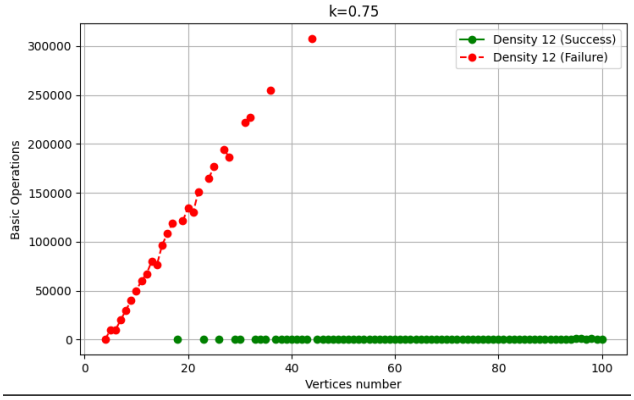

Fig. 7 – Results of Randomized Algorithm with k=0.75, Density 12 for Basic Operations, separting the Sucess and Failure cases

Focusing on the success line (Fig. 8), we can observe a clear linear trend, unaffected by density or k values, and influenced only by the number of vertices. This is because the operations that contribute to these results are those performed before the main loop to find the solution, specifically, the operations required to assign probabilistic weights to the vertices.

Additionally, in success cases involving graphs with 100 vertices, the number of operations does not even reach 800, further confirming the algorithm's efficiency in handling successful solutions. This highlights its ability to deliver excellent performance when valid edge covers are found.

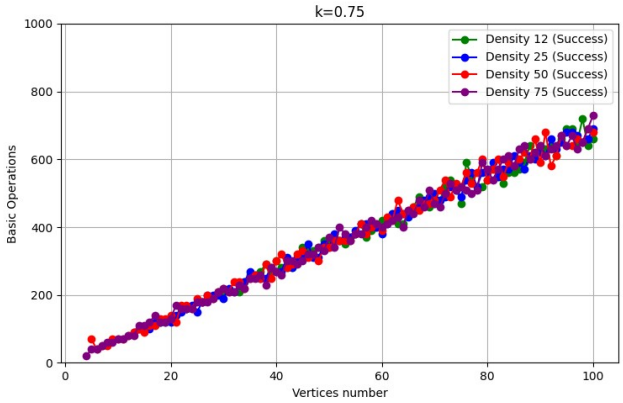

Fig. 8 – Results of Randomized Algorithm with k = 0.75 Success cases for Basic Operations

Below are the graphs of execution time (Fig. 9) and configurations tested (Fig. 10) for the exhaustive algorithm with k = 0.125.

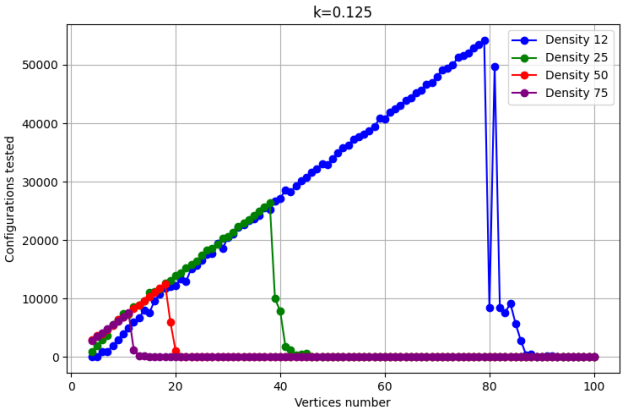

Fig. 9 – Results of Randomized Algorithm with k = 0.125 for Tested Configurations

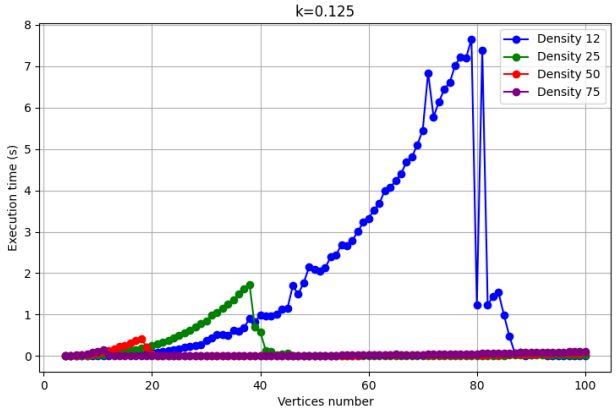

Fig. 10 – Results of Randomized Algorithm with k = 0.125 for Execution Time

We can see that this algorithm presents good times, being the bigger time 7.7 seconds.

A script was also created to analyze the number of False Positives and False Negatives produced by the Randomized Algorithm, as it is based on a Monte Carlo approach, in order to calculate its accuracy. The following values were obtained:

- False Positives: 0
- False Negatives: 1

This demonstrates that although this algorithm can generate erroneous results, it remains a very reliable algorithm

C. Algorithms Comparation

As expected, the randomized algorithm significantly outperforms the exhaustive algorithm and approaches the computational efficiency of the greedy algorithm (Fig. 11). However, a closer examination reveals key differences between the two (Fig. 12). The greedy algorithm outperforms the randomized algorithm in failure cases, while the randomized algorithm shows superior performance in success cases.

This behavior aligns with the observable trends in the study case. It is perdicted that for larger graphs, especially in failure cases, the randomized algorithm will eventually outperform the greedy algorithm. This is because the randomized algorithm exhibits linear growth, whereas the greedy algorithm's computational cost grows polynomial.

These relationships are demonstrated in the graphs below, which show 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.

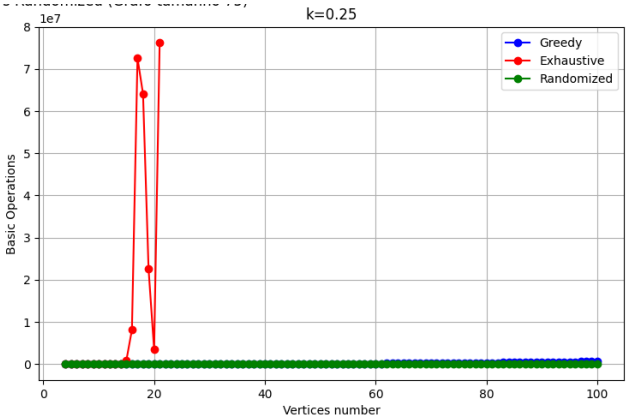

Fig. 11 - Results of Algorithm Comparison with k = 0.25 and Density = 25% for Basic Operations.

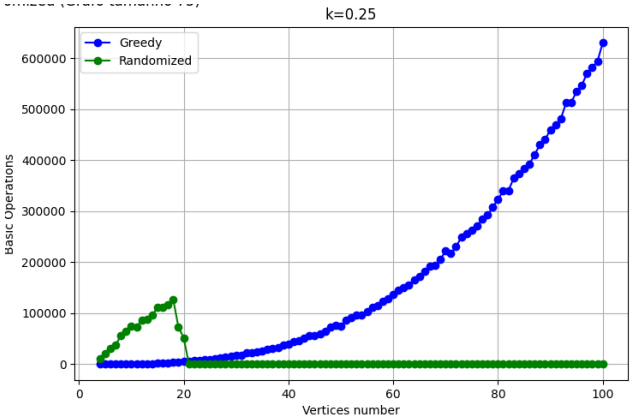

Fig. 12 - Results of Algorithm Comparison with k = 0.25 and Density = 25% for Basic Operations, only Greedy and Randomized.

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, in these cases, most of the graph solutions represent “Success” solutions so the exhaustive and randomized algorithms become the faster option. 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.

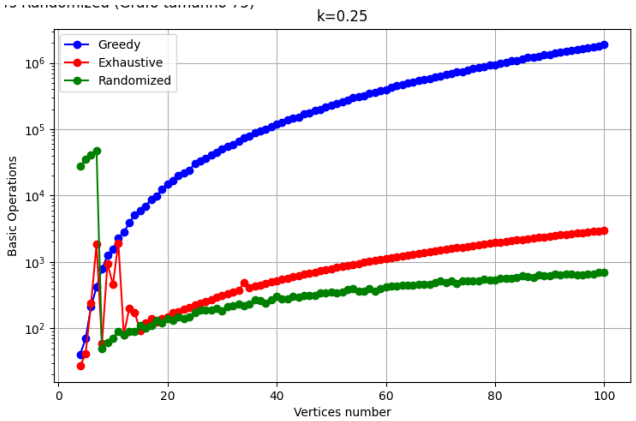

Fig. 13 - Results of Algorithm Comparison with k = 0.25 and Density = 75% for Basic Operations in a logarithm scale.

D. Execution Time for larger problems

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

Data:

- `Complexity: O(max_iterations * m)`
- `n= 10 000 vertices`
- `m= 24 997 500 edges`
- `max_iterations = 1,000`

Calculation:

1. `Operations total: 1,000·24,997,500 = 2.5×1010`
2. `Estimated execution time = 2.5×1010 /10^7 = 2,500 seconds ≈ 41.67 minutes`

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

E. Results for Benchmark graphs

The algorithm was also applied to benchmark cases using graphs from Twitch Social Networks [8], representing streamers social networks for different languages. The randomized algorithm was executed on each of these larger graphs, with a time limit of 10 minutes (600 seconds) per run.

Graph	Nodes	Edges	Density	Time (s)
DE	9 498	153 138	.003	-
ENGB	7 126	35 324	.002	-
ES	4 648	59 382	.006	287
FR	6 549	112 666	.005	-
PTBR	1 912	31 299	.017	34
RU	4 385	37 304	.004	261

For the largest graphs, the algorithm reached the maximum execution time. However, for the graphs with less than 5 000 vertices, it completed within the time limit. For example, the ES graph took 287 seconds (approximately 5 minutes), the PTBR graph completed in 34 seconds, and the RU graph finished in 261 seconds (approximately 4 minutes). These results highlight the algorithm's scalability and efficiency for moderately sized graphs.

VI. CONCLUSIONS

This study explored the efficiency and applicability of three algorithms: Exhaustive Search, Greedy Search, and Randomized Search in determining the existence of an edge cover in undirected graphs. Each algorithm offers distinct advantages and limitations, making them suitable for different scenarios.

In the first article we saw that the Exhaustive Search algorithm, while guaranteeing exact solutions, proved impractical for larger graphs due to its exponential growth in computational cost. Conversely, the Greedy Algorithm, with its polynomial complexity, performed efficiently in most cases, though it occasionally produced false negatives.

The Randomized Algorithm, introduced in this article, demonstrated promising results, combining elements of randomness and probabilistic weighting to navigate complex solution spaces effectively. With its linear growth complexity, the algorithm proved scalable and particularly advantageous for larger graphs, outperforming the exhaustive method and approaching the computational efficiency of the greedy approach. It excelled in "success" cases, showcasing its reliability and efficiency for graphs with valid edge cover solutions.

REFERENCES

[1] University of Aveiro. "Advanced Algorithms Course Materials."

[2] Wikipedia. "Edge Cover." Accessed November 12, 2024. https://en.wikipedia.org/wiki/Edge_cover

[3] Wikipedia. "Greedy Algorithm." Accessed November 12, 2024. https://en.wikipedia.org/wiki/Greedy_algorithm

[4] Wikipedia. "Brute-Force Search." Accessed November 12, 2024. https://en.wikipedia.org/wiki/Brute-force_search

[5] OpenAI. "ChatGPT." Accessed November 12, 2024. https://chatgpt.com/

[6] Wikipedia. "Edge Cover." Accessed December 2, 2024. https://en.wikipedia.org/wiki/Randomized_algorithm

[7] Wikipedia. “Monte Carlo Algorithm” Accessed December 2, 2024. https://en.wikipedia.org/wiki/Monte_Carlo_algorithm

[8] SNAP “Twitch Social Networks” Accessed December 3, 2024. https://snap.stanford.edu/data/twitch-social-networks.html