Algoritmos Avançados - Exhaustive Search

Tiago Santos, 89356

Abstract **–This report absorbs an exhaustive search and a greedy heuristics approach to solve the problem of the maximum clique in a graph, by comparing the time of completion and its results.**

I. Context and Objectives

The objective of this project was to determine the **maximum clique** of a given undirected graph using 2 different approaches: **exhaustive search** and **greedy heuristics**. A clique of a graph is a subgraph where all of its nodes are adjacent to each other, meaning they are all connected. The maximum clique is the one with the largest number of nodes possible.

After applying both methods, we compare times of completion, by the number of nodes and percentage of edge creation, to then explain their differences.

II. Code

A. Exhaustive search

Exhaustive search, or brute-force search, is a more straightforward method for solving problems by exploring all possible solutions. In this approach, the algorithm generates all possible combinations or permutations of solutions and evaluates each one to find the optimal solution. While exhaustive search is conceptually simple and guarantees finding the best solution, it can take longer, especially when the search space is large.

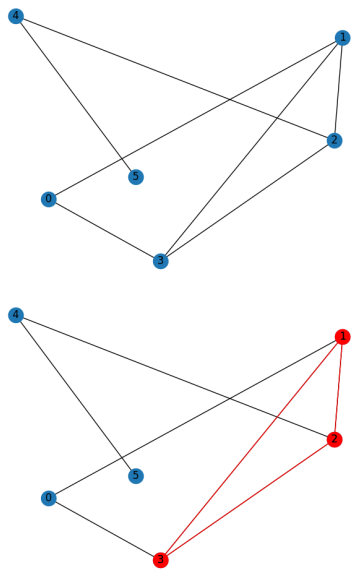

Fig. 1: Example of maximum clique

In the context of this project, we iterate through all the possible combinations of nodes, considering different clique sizes, and save these combinations in a list. We then iterate through this list and check if each combination forms a clique by verifying the connections between nodes. If a combination satisfies the conditions of a clique, meaning that every pair of nodes within the combination is connected, we consider it a potential clique. We then identify the maximum clique by selecting the combination with the largest number of nodes.

```
    for clique_size in range(2, len(graph.
    nodes) + 1):

        for nodes_combination in itertools.
    combinations(graph.nodes, clique_size):
            aux_clique = list(
    nodes_combination)
            all_combinations.append(aux_clique
    )

    max_clique = []

    for comb in all_combinations:

        if check_if_clique(comb,graph):
            if len(comb) > len(max_clique):
                max_clique = comb
...
def check_if_clique(combination, graph):
    global op_count
    for n1 in combination:
        neighbors = set(graph.neighbors(n1))
        for n2 in combination:
            op_count += 1
            if n2 != n1 and n2 not in
    neighbors:
                return False
    return True
```

These loops indicate a search through all possible combinations of nodes, and the check_if_clique function verifies if each combination forms a clique. The worst-case time complexity can be analyzed as follows:

- Outer Loop: The outer loop iterates for each possible clique size from 2 to the number of nodes in the graph (len(graph.nodes)).
- Inner Loop: The inner loop iterates over all combinations of nodes for the current clique size. For each combination, the check_if_clique function is called.

Considering the worst-case scenario where all combinations are checked, the time complexity of this algorithm is often expressed as 2^n, where n is the number of nodes in the graph. This is because, for each node, you have two choices (include it in the clique or not), and there are n nodes.

B. Greedy Heuristics

Greedy heuristics, on the other hand, are algorithms that make locally optimal choices at each step with the hope of finding a global optimum. This method the best possible decision at each stage without considering the consequences of that decision on future steps. While greedy algorithms are computationally more efficient compared to exhaustive search, they do not always guarantee finding the globally optimal solution.

In the context of this project, the algorithm begins by sorting the nodes in the graph based on their degrees in descending order, considering nodes with higher degrees to be first. It then iterates over the sorted nodes, starting with the node with the highest degree and initializing the clique with it. For each subsequent node in the sorted list, the algorithm attempts to add the node to the current clique, if possible. After considering all nodes, the algorithm returns the nodes representing the maximum clique in the graph.

```
    def max_clique_greedy(graph):
    if not graph.nodes:
        return []

    # Sort nodes by degree in descending order
    nodes_sorted_by_degree = sorted(graph.
    nodes, key=lambda x: graph.degree(x),
    reverse=True)

    # Initialize the clique with the first
    node
    clique_max = [nodes_sorted_by_degree[0]]

    for node in nodes_sorted_by_degree[1:]:
        if check_in_clique(node, clique_max,
    graph):
            clique_max.append(node)

    return clique_max

    def check_in_clique(node,clique_max,graph)
    :
    global op_count
    neighbors = set(graph.neighbors(node))
    for n in clique_max:
        op_count += 1
        if n not in neighbors:
            return False
    return True
```

C. Greedy Alternative

An alternative greedy heuristics method is presented by the following Python code:

```
def max_clique_extra(graph):
    if not graph.nodes:
        return []

    clique_max = []

    for node in graph.nodes():
        aux_clique = [node]
        for n in graph.nodes():
            if n == node:
                continue
            else:
                if check_in_clique(n,
    aux_clique, graph):
                    aux_clique.append(n)
        if len(aux_clique) > len(clique_max):
            clique_max = aux_clique

    # Return the maximal clique
    return clique_max
```

This alternative method follows a similar greedy approach by iteratively adding nodes to a list and checking if they form a clique. It then compares the size of this clique with the biggest one it encontered. However, it uses a different strategy in the inner loop compared to the original greedy heuristic.

III. Experiment

To evaluate the performance of the algorithms, two Python files were developed:

- main.py: This script conducts tests on a single graph with a user-defined number of nodes. It records the execution time for the algorithm on this specific graph.
- experiment.py: This script systematically evaluates the algorithm's performance across various graph sizes. The experiment starts with 4 nodes and increments up to 26 nodes (due to computational limitations), repeating each set amount of nodes 10 times. The execution times for each run are recorded, and the median time is calculated, as well as basic operations.

The aggregated median times for each node count and edge creation percentage are compiled into a dataset. To visualize the results, a graphical representation is generated, where the x-axis represents the number of nodes, and the y-axis represents the execution time in seconds. It was also created an additional graphic comparing the number of operations done with the number of nodes.

This experiment is repeated four times, each with a different edge creation percentage (12.5%, 25%, 50%, and 75%), and a different search method.

IV. Results and Comparsion

A. Exhaustive Search

The results obtained from the exhaustive search method are presented in Figure 2. Each graph illustrates the median execution times for different numbers of nodes and edge creation percentages.

A.1 Observations

- For the exhaustive search method, the execution time tends to grow exponentially with the number of nodes. This aligns with the expected time complexity of 2^n, as discussed in the Code section.
- As the edge creation percentage increases, the search space expands, leading to longer execution times. This is evident in the graphs for 12.5% and 25% edge creation reaching around 30 plus seconds and 75% around 80 plus seconds.

B. Greedy Heuristics

The results for the greedy heuristics method are depicted in Figure 3. Similar to the exhaustive search,

these graphs illustrate the median execution times for different configurations.

B.1 Observations

- Greedy heuristics generally show more favorable execution times compared to exhaustive search, especially as the number of nodes increases.
- However, there are instances where the greedy heuristic does not yield the correct maximum clique, as evidenced by Figure 4. This highlights the trade-off between efficiency and optimality in greedy algorithms.

C. Greedy Alternative

The effectiveness and efficiency of this alternative method should be compared to the original greedy heuristic and other approaches in the context of your specific problem. This method was proven to be faster than the exhaustive search, as seen in Figure **??**, but still slower than the previous greedy heuristic and having more basic operations.

D. Comparison

Comparing the two main methods, we observe that:

- The exhaustive search provides a guaranteed optimal solution but at the cost of much longer execution times when using larger graphs. This can also be confirmed by the number of basic operations, for example when checking if nodes are connected or not, where an exhaustive search contains much more operations than greedy heuristics.
- Greedy heuristics, while more computationally efficient, with much less amount of operations, may sacrifice optimality in some cases, as shown by the incorrect result in Figure 4.

V. CONCLUSION

Overall, the choice between these methods depends on the specific requirements of the application. If obtaining the absolute maximum clique is crucial and the graph size is manageable, an exhaustive search might be preferred. On the other hand, for larger graphs where efficiency is a priority, greedy heuristics can be a reasonable compromise, even if it's necessary to repeat the search to confirm the result.

VI. HOW TO USE

For main.py:

```
python3 main.py <number of nodes> <% edge creation> <method: g(greedy) || e(exhaustive)>
```

For experiment.py:

```
python3 experiment.py <% edge creation> <method: g(greedy) || e(exhaustive)>

Example for experiment with 12.5% using greedy heuristic:

    python3 experiment.py 12.5 g
```

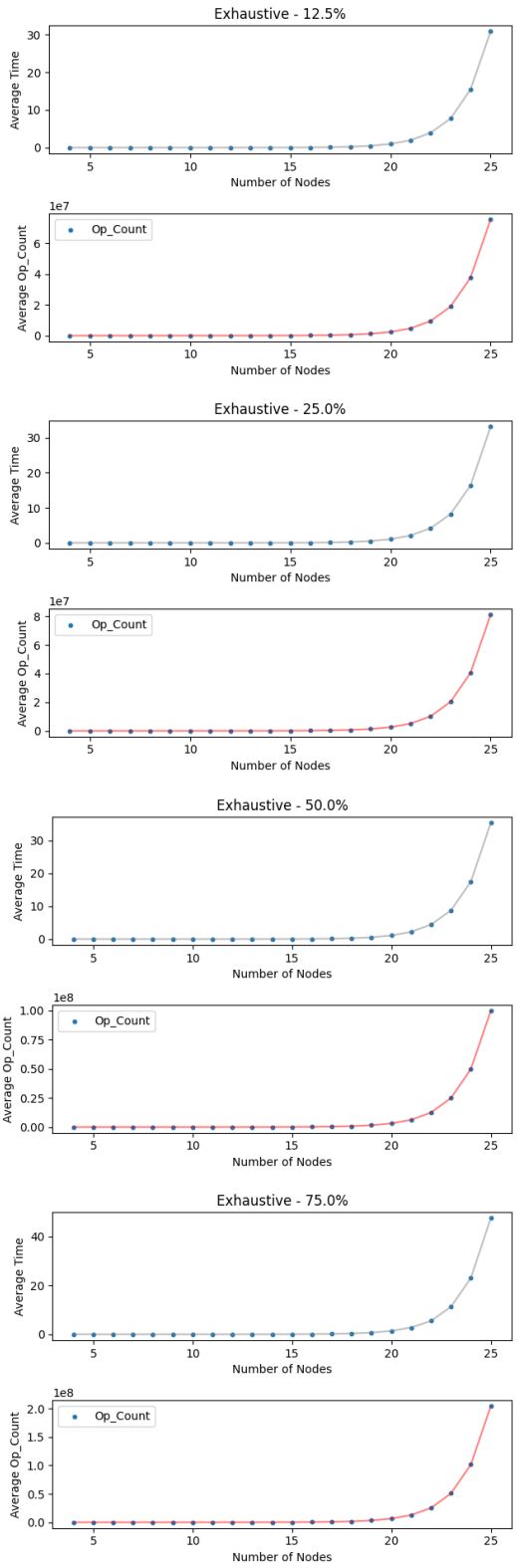

Fig. 2: Results using exhaustive search

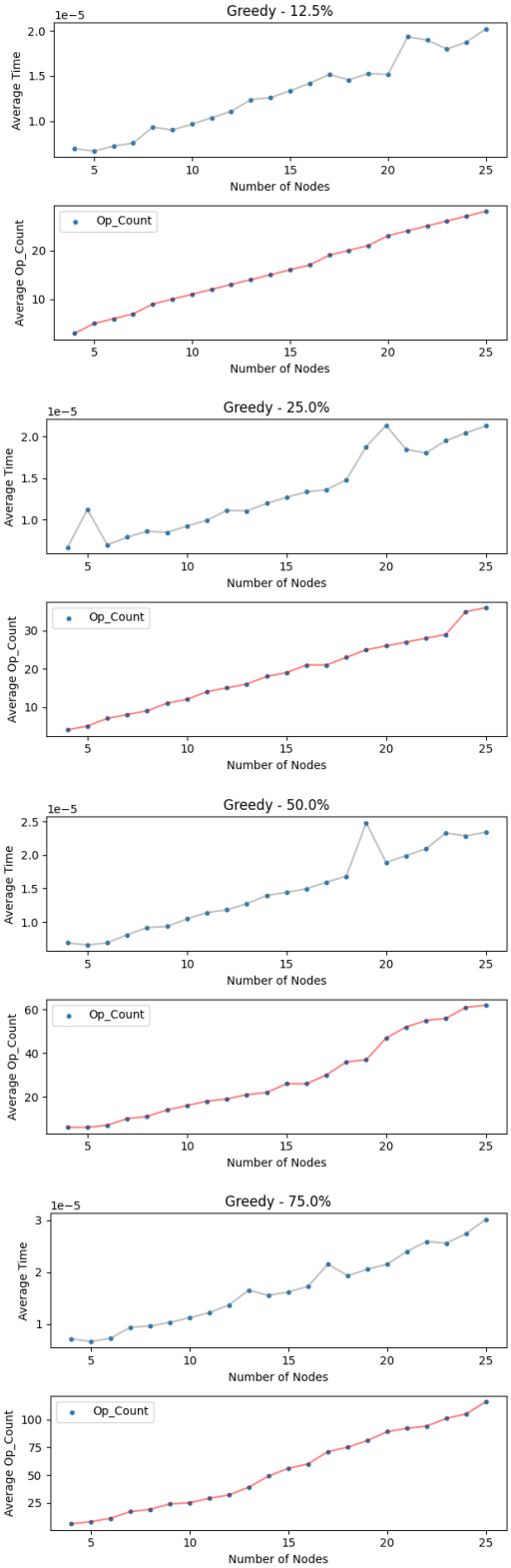

Fig. 3: Results using greedy heuristics (time in 10^{-5}s)

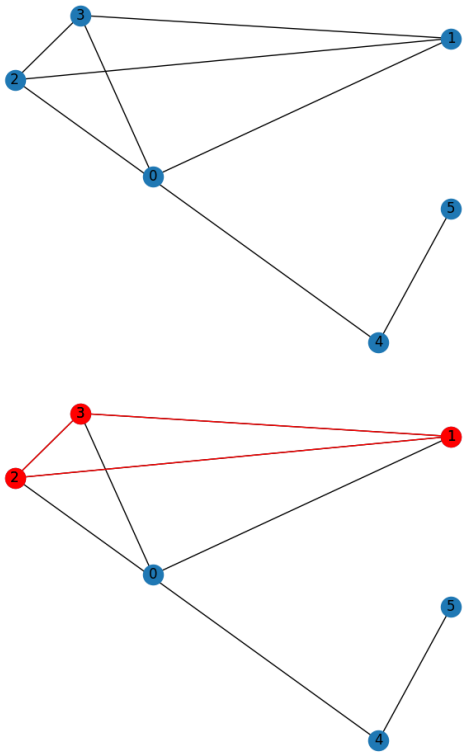

Fig. 4: Wrong result from greedy heuristic

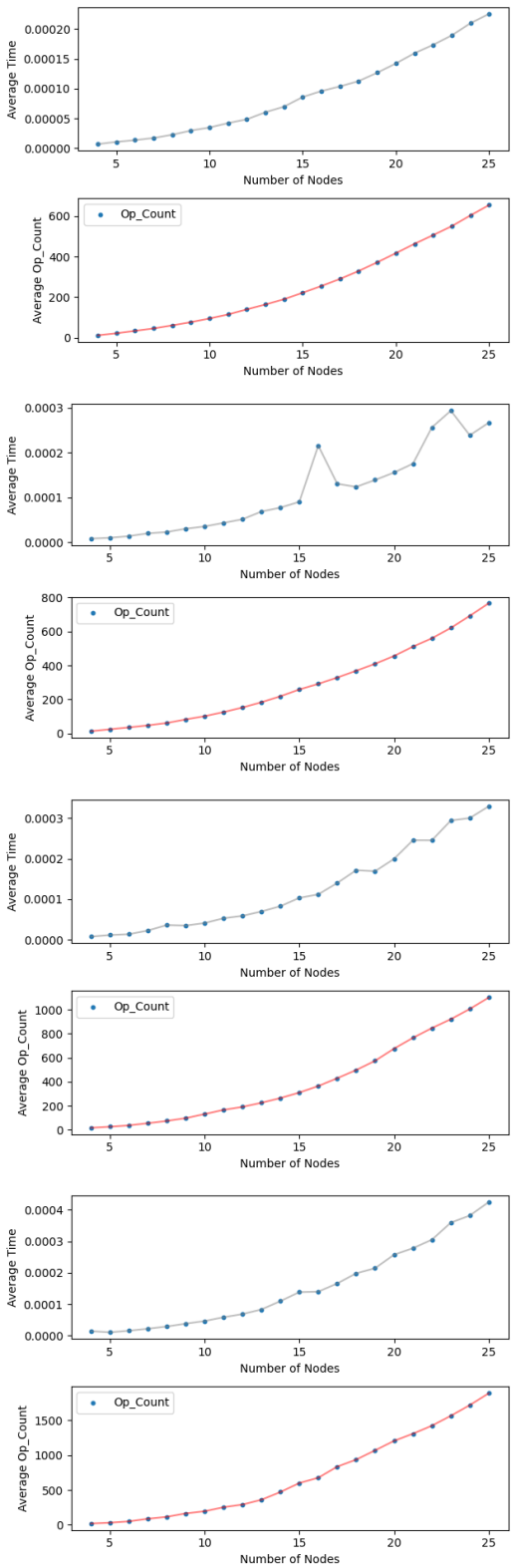

Fig. 5: Results using greedy heuristics (time in 10^{-5}s)