Exhaustive and Greedy Search for finding the Maximum Independent Set of a Graph

Miguel Matos, 103341

***Abstract* – Design, testing, and analysis of two computational algorithms to find the maximum independent set for a given undirected graph.**

I. Introduction

In this assignment, we design, test, and analyse two computational algorithms to find the maximum independent set for a given undirected graph *G(V, E)*, with *n* vertices and *m* edges. An independent set of G is a subset of vertices, no two of which are adjacent; thus, the maximum independent set of G is the independent set of largest size. A practical example would be a graph that represents a social network, where the vertices represent the people and the edges are the connections between these; the maximum independent set here represents the largest subset of people who do not know each other.

II. Exhaustive Search Algorithm

A. Design

An exhaustive search algorithm, also knows as a brute-force algorithm, systematically enumerates all possible candidates for the solution and check whether each candidate satisfies the problem's solution.

In this case, we generate all possible subsets of the given graph, and for the independent ones, check which one is the largest.

Translating this to pseudocode, we get the following outline:

```
1. Initialize max_set as an empty set
2. For each subset of vertices S:
    a) If S is an independent set and the
size of S is greater than the size of max_set
        i. Set max_set to S
3. Return max_set
```

For each subset, we check if it's an independent set by ensuring no two vertices in the subset are connected by an edge.

B. Formal Analysis

The exhaustive search algorithm check every subset of vertices to see if it forms an independent set, looking for the largest one. For a graph with *n* vertices, there are 2^n possible subsets, and the algorithm searches through each one of them.

After, for each subset, to check if it is independent we need to check if any edge connects two vertices of the subset. This yields a time complexity of *O(m)*, where m is the number of edges of the subset.

This means that the exhaustive search algorithm has an exponential time complexity of $O(2^n \cdot m)$.

C. Experimental Analysis

In order to verify the formal analysis results, the algorithm was run on randomly generated graphs with different number of vertices. For each vertex amount, four different graphs were generated, each with a different percent of edge density. The graphs had the following properties:

- $n \in [4, 25]$, as 25 was the computationally feasible number of vertices
- *e* being one of the four proportions of the maximum number of edges {0.125, 0.25, 0.5, 0.75}
- vertex coordinates in the range of [1, 100]
- minimum distance between vertexes of 4

Three metrics were evaluated: execution time, number of basic operations performed, and the number of combinations tested.

The number of basic operations is plotted in Figure 1 over the number of vertices of each graph. The *y* axis is on a logarithmic scale. Since the behaviour of the plotted line is linear and the scale of the *y* axis is logarithmic, we can infer that the number of basic operations grows exponentially with the number of vertices, corroborating the results obtained in the formal analysis.

The execution time results are plotted in Figure 2, with the same setup as the number of basic operations. Again, since the *y* axis scale is logarithmic and the behaviour of the line is linear, we can conclude that the execution time of this algorithm grows exponentially with the number of vertices.

With this insight, we can make a prediction for execution time of this algorithm for much larger graph instances. As seen in Figure 2, for 20 vertices, at 50% edge ration, the algorithm takes around 1 second to execute. Assuming that the execution time grows by 1000 times for each jump of 10 vertices, a graph with 50 vertices with the same edge ratio should take around 10^{16} seconds, which equates to around 316 887 646 years. This is a lot.

Figure 3 depicts the number of combinations tested by the algorithm over each number of vertices. Again, the *y* axis is on a logarithmic scale. Since the number of possible subsets for a given number of vertices *n* is 2^n, the edge ratio of each graph has no impact on the amount of combinations tested, since all sets will be tested anyway. Thus, the growth of the number of combinations tested is exponential as well.

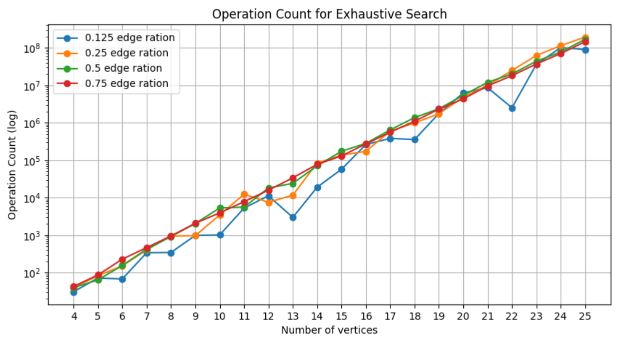

Figure 1 - Number of basic operations performed by the exhaustive algorithm over the number of vertices

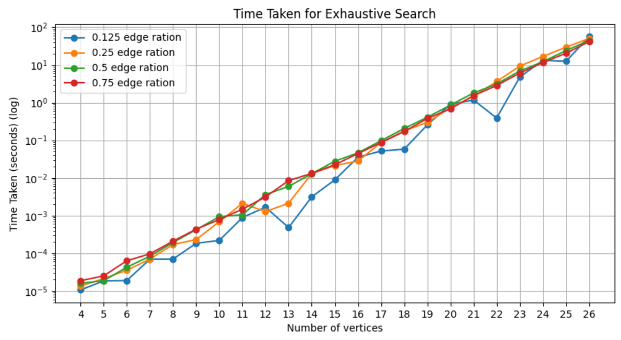

Figure 2 - Execution time for the exhaustive algorithm over the number of vertices

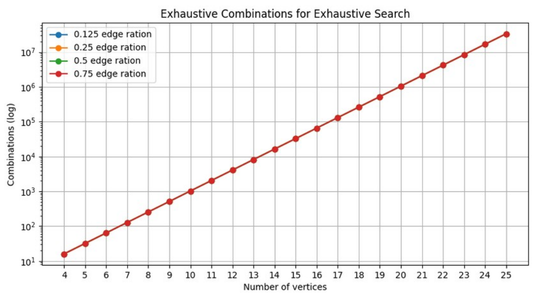

Figure 3 - Number of combinations tested over the number of vertices

III. GREEDY SEARCH ALGORITHM

A. Design

Searching a solution space exhaustively is not feasible when this space is exponentially proportional to the input size. In order to improve performance, one may apply an heuristic that may or may not yield the optimal solution.

A greedy search algorithm functions by using the given problem-solving heuristic to make the locally optimal choice at each stage of computing. In many problems, it ends up not producing an optimal solution, but it can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

For this specific problem, the designed heuristic works by iteratively adding vertices to the independent set that have the least number of edges connecting them to vertices not already in the set. This heuristic assumes that vertices with fewer edges are less likely to reduce the potential size of the independent set in future steps.

Again, translating this to pseudocode:

```
1. Initialize max_set as an empty set
2. Sort the vertices by degree in ascending order
2. While there are still vertices for consideration not in independent_set:
    a) Select the vertex v not in max_set with the minimum degree
    b) Add v to max_set
    c) Remove v and all its neighbors from consideration
3. Return max_set
```

B. Formal Analysis

To make a formal analysis on the computational complexity of this algorithm, one must first understand each of its steps. Excluding steps with constant time operations, like initializing *max_set*, we have a sorting step and a loop step.

The sorting algorithm used is a merge sort, which has a complexity of *O(n log n)* in average and worst-case scenarios. Here, sorting is done based on vertex degree.

The loop step iterates based on a list that at start contains all vertices, which we call the vertices up for consideration. Each iteration removes vertices from consideration, which means that this loop will never execute more than *n* times. Inside each iteration, we are required to iterate through all edges of the vertex we are iterating on to remove them from consideration; in the worst case, we might have to traverse all edges of the graph, as each vertex might be connected to many others.

Considering these steps, we can estimate a rough upper bound for the time complexity of this algorithm, given that:

- the sorting step is *O(n log n)*
- the while loop iterates at most *n* times
- each iterations complexity can be assumed as *O(m)* for the worst case

Putting these steps together results in an approximated complexity of *O(n log n + n m)* which can be simplified to $O(n^2 + m \log n)$. This complexity is polynomial, which is significantly better than the exhaustive search that has an exponential complexity. With the number of vertices being the input, we can say that this method features quadratic complexity.

The algorithms performance is also dependent on the graph's edge density. We expect its performance to be generally better on sparser graphs than on graphs that are more dense (have a higher edge ratio).

C. Experimental Analysis

Again, to validate the results of the former formal analysis, experimentation was conducted over graphs of the same characteristics as the exhaustive algorithm, with the following difference:

- $n \in [4, 200]$, since the algorithm is computationally lighter

Three metrics were evaluated: execution time, number of basic operations performed, and the size of maximum independent set found, to compare against the results obtained by the exhaustive algorithm, since the greedy algorithm may not always find the maximum independent set. Number of combinations was not analysed because this algorithms does not traverse a solution space, constructing a single solution instead.

Figure 4 depicts the relation between the number of basic operations performed by the greedy algorithm over the number of vertices. For this algorithm, the scale of the *y* axis is almost linear, although a curve is slightly noticeable as we provide more data. The scale of the numbers on the *y* axis is noticeably smaller than the one on the exhaustive search algorithm, highlighting the performance gap between the algorithms.

The execution time results are plotted in Figure 5. The same relation between basic operation counts can be seen in the execution time plots – growth is now close to linear and the scale of the numbers is much lower.

The bar plot on Figure 6 shows the difference between the sizes of the sets found by both algorithms, since the greedy search might not always find the maximum independent set. We can see that the algorithm's precision tends to decline over the number of vertices, which is a fair trade-off given the increase in time performance.

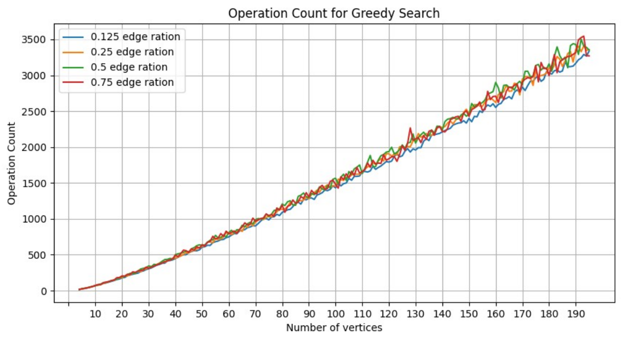

Figure 4 - Number of basic operations performed by the greedy algorithm over the number of vertices

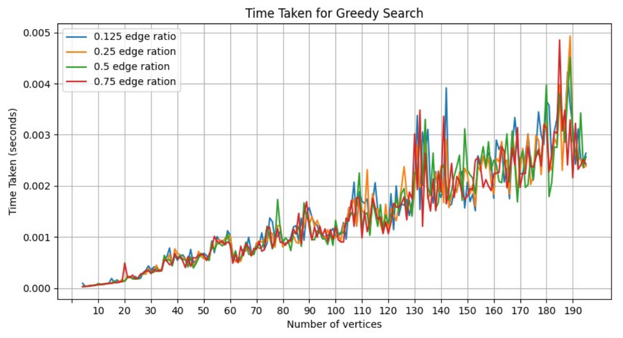

Figure 5 - Execution time for the greedy algorithm over the number of vertices

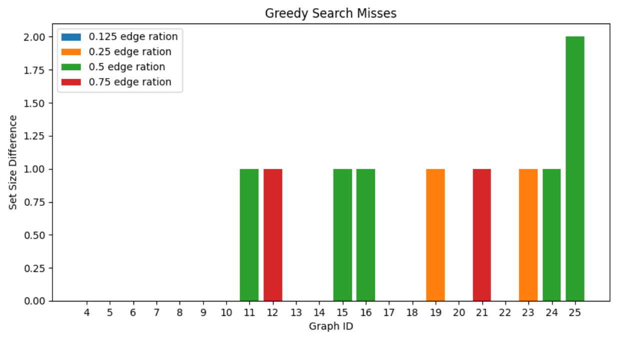

Figure 6: Difference between the sizes of the max independet set found by both algorithms

IV. CONCLUSION

In this report, two algorithms were designed to find the maximum independent set of a given graph. Each algorithm followed a different approach, being the first one exhaustive, and the second one greedy.

Both approaches were reviewed in terms of their time complexity, with both formal and experimental analysis being performed. The exhaustive search provided exponential complexity of $O(2^n)$, which is expected for algorithms of this nature, while the greedy search had a quadratic complexity $O(n^2)$, which is significantly better for the same input of number of vertices. However, the greedy method did not always find the optimal solution, since it only makes locally optimal solutions. Therefore, we end up with two different approaches that have a tradeoff relationship between correctness and computing time.

REFERENCES

[1] https://chat.openai.com/
[2] https://en.wikipedia.org/wiki/Optimization_problem
[3] https://en.wikipedia.org/wiki/Independent_set_(graph_theory)
[4] https://en.wikipedia.org/wiki/Greedy_algorithm