Minimum Vertex Cover Problem

Manuel Diaz

Resumo - **Este relatório é um estudo aprofundado do problema *Minimum Vertex Cover*, uma variante mais pequena de um dos problemas mais básicos da teoria dos grafos. Adoptamos duas abordagens algorítmicas: uma utiliza uma estratégia de pesquisa exaustiva de força bruta, enquanto a outra se baseia numa abordagem heurística gulosa para encontrar o conjunto de vértices de menor dimensão que cubra todas as arestas do grafo não direcionado G(V, E), em que o número total de vértices é 'n' e o número de arestas é 'm'. Enquanto o método exaustivo verifica diretamente todos os subconjuntos de vértices para obter uma solução exacta através de um método exaustivo, a heurística gulosa aproxima-se de uma solução selecionando iterativamente os vértices que estão ligados ao número máximo de outros vértices. Realizamos experiências extensivas em grafos de diferentes tamanhos e várias densidades de arestas para analisar a complexidade computacional de cada algoritmo, a eficiência do tempo de execução e a precisão na procura de uma solução.**

Abstract - **This report is an in-depth study of the Minimum Vertex Cover problem, a smaller variant of one of the most basic problems in graph theory. We adopt two algorithmic approaches: one uses a brute-force exhaustive search strategy, while the other is based on a greedy heuristic approach to find the smallest vertex set that covers all the edges in the undirected graph G(V, E), where the total number of vertices is 'n' and the number of edges is 'm'. While the exhaustive method directly checks all subsets of vertices to obtain an exact solution through an exhaustive method, the greedy heuristic approaches a solution by iteratively selecting vertices that are connected to the maximum number of other vertices. We carried out extensive experiments on graphs of different sizes and various edge densities to analyse the computational complexity of each algorithm, the efficiency of the execution time and the accuracy in finding a solution.**

I. INTRODUCTION

The Minimum Vertex Cover problem is a well known challenge in graph theory and combinatorial optimization, with applications across a variety of fields, including network security, bioinformatics, scheduling and resource management. Formally, given an undirected graph, $G = (V, E)$, the objective is to identify the smallest subset of vertices $C \subseteq V$ such that every edge in E has at least one endpoint in C. This subset is called a vertex cover and indeed covers all edges, so the problem is a typical problem of a complete covering of edges using the minimum amount of resources.

In this report, we investigate two algorithmic approaches for solving the MVC problem: an exhaustive search method and a greedy heuristic.

The exhaustive search enumerates all the possible subsets of vertices, which guarantees an optimum but is prohibitively expensive. This is feasible only for very small graphs, as the number of sets of vertices grows exponentially. In contrast, the greedy heuristic approach is a really fast approximation that constructs the solution iteratively by selecting vertices with the highest degree of connectivity, aiming the covering of all edges as quickly as possible. Although this heuristic does not guarantee an optimal solution, it reduces the computation time greatly and is, therefore, more suitable for large graphs.

In the subsequent sections we will comparatively analyse both algorithms for computational complexity, execution time, and the quality of the obtained solution by a series of experiments on randomly generated graphs of various sizes and edge densities

II. EXHAUSTIVE SEARCH

The exhaustive search approach to solving the Minimum Vertex Cover (MVC) problem involves evaluating all possible subsets of vertices in a graph to find the smallest subset that covers all edges. This algorithm is designed to provide an exact solution, ensuring that the minimum vertex cover is identified for any given graph.

A. Algorithm Overview

In this approach, we consider every possible subset of vertices of the vertex set V of the graph $G = (V, E)$, testing each one to determine if it constitutes a valid vertex cover. For each subset $C \subseteq V$, the algorithm checks whether C covers all edges in E. If it does, it compares its size to the current minimum vertex cover found and updates the minimum if C is smaller.

B. Complexity analysis

The exhaustive search algorithm is computationally expensive due to the combinatorial nature of subset generation. Given n vertices, there are 2^n possible subsets, leading to an exponential time complexity of $O(2^n)$. This exponential growth in complexity significantly limits the practicality of the exhaustive search for larger graphs, as the number of subsets grows to rapidly to handle efficiently.

Despite these limitations, this algorithm consistently identified the optimal minimum vertex cover, affirming its effectiveness for smaller graphs where the computational expense is not a critical factor.

C. Implementation

The following code outlines the steps of the exhaustive search algorithm:

```
def is_vertex_cover(graph, vertex_set):
    for u, v in graph.edges():
        if u not in vertex_set and v not in vertex_set:
            return False
    return True

def find_minimum_vertex_cover(graph):
    vertices = list(graph.nodes())
    n = len(vertices)

    for r in range(1, n + 1):
        for subset in combinations(vertices, r):
            if is_vertex_cover(graph, subset):
                return set(subset)
```

Algorithm 1: Exhaustive search for Minimim Vertex Cover.

The *find_minimum_vertex_cover* function generates all possible vertex subsets (*vertices*) and checks each one (*subset*) through the function *is_vertex_cover* to identify the smallest subset that covers all edges in the *graph*.

D. Experimental Analysis

To evaluate the performance of the exhaustive search algorithm in solving the Minimum Vertex Cover problem, we conducted a series of experiments using randomly generated graphs with varying sizes where the number of vertices is $n \in [4, 29[$ and edge densities were generated defining the probability of two vertices have an edge $p \in \{0.125, 0.25, 0.5, 0.75\}$. The random seed was fixed to ensure reproducibility. For each graph configuration, we recorded the following metrics: **number of solutions tested**, **number of operations required**, and the **computational time taken**, which provide insight into the algorithm's scalability and resource requirements.

D.1 Number of Solutions Tested:

The number of solutions tested increases with the number of vertices, but it also varies significantly with graph density. For lower-density graphs, a higher number of vertex subsets can potentially cover all edges, leading to more candidate solutions tested. In contrast, in high-density graphs, fewer vertex subsets meet the criteria for a vertex cover due to the greater connectivity between vertices, which reduces the number of valid solutions.

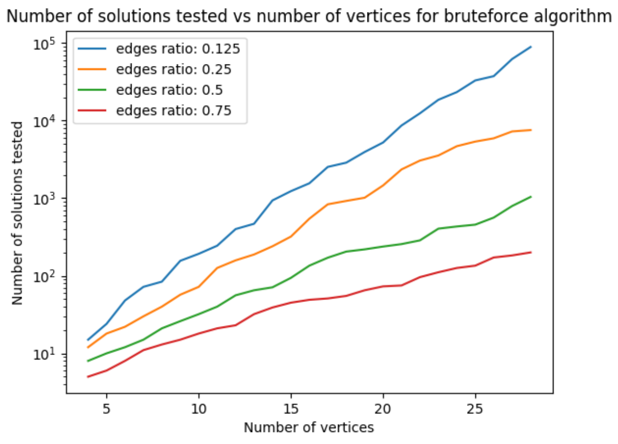

Figure 1: The Exhaustive Search algorithm's number of solutions tested in function of the number of vertices and edge density.

D.2 Number of Operations:

The operation count, which includes checks for each vertex subset, mirrors the exponential growth of the solutions curve as the number of vertices increases. However, the effect of edge density is the opposite: higher densities significantly increase the number of operations due to the greater number of edges requiring validation. For larger graphs, particularly those with dense edge configurations, the operation count becomes prohibitively high, underscoring the impracticality of this method for large-scale applications, as illustrated in Figure 2.

D.3 Execution Time:

Execution time increases dramatically with graph size and edge density, demonstrating the algorithm's exhaustive nature that mandates checking all possible combinations (Figure 3). This step increase highlights the method's unsuitability for large, complex graphs.

The results demonstrate that while search approach guarantees an optimal solution for the MVC problem, its application is limited to small graphs. These findings underscore the limitations of exhaustive search and emphasise the need for more scalable approaches, such as heuristics, when dealing with larger instances.

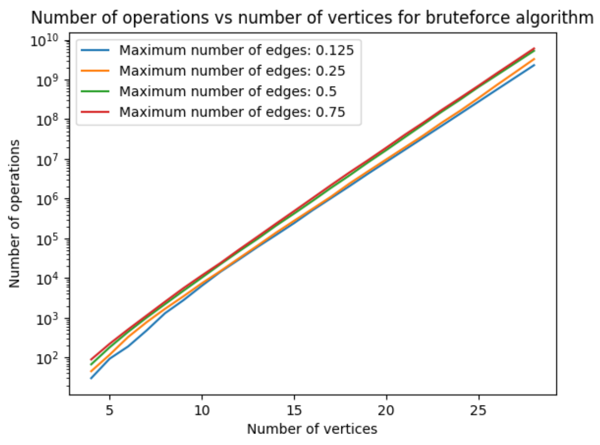

Figure 2: The Exhaustive Search algorithm's number of operations in function of the number of vertices and edge density.

III. GREEDY SEARCH

The greedy search algorithm provides an approximate solution to the Minimum Vertex Cover problem by iteratively selecting vertices based on a heuristic criterion rather than exhaustively evaluating all possible subsets. This approach sacrifices optimality for efficiency, allowing for faster computation and greater scalability compared to exhaustive search, especially for larger graphs.

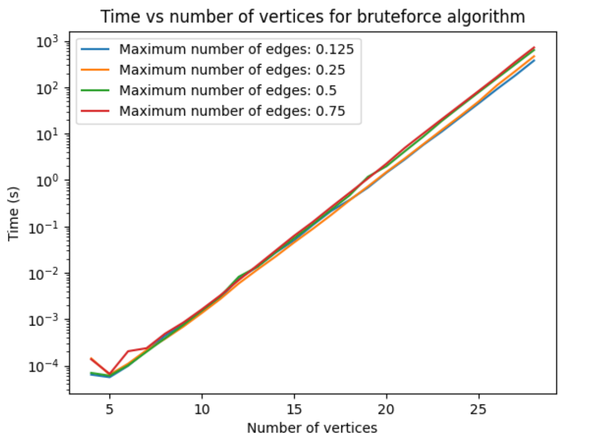

Figure 3: The Exhaustive Search algorithm's time execution in function of the number of vertices and edge density.

A. Algorithm Overview

This algorithm aims to quickly cover all edges by selecting vertices in a way that maximizes edge coverage at each step. It follows this steps:

1. **Initialize an Empty Cover:** Start with an empty cover set, C.

2. **Select Vertices Iteratively:** At each iteration, choose the vertex that is incident to the maximum number of uncovered edges (i.e., the vertex with the highest degree among uncovered edges).

3. **Update Edge Coverage:** After adding a vertex to the cover, mark all edges connected to it as covered, removing them from the list of uncovered edges.

4. **Repeat** until all edges are covered, at which point the set C is returned as the approximate vertex cover.

This heuristic does not guarantee a minimum cover, but it aims to cover the graph efficiently by selecting vertices that contribute the most to edge coverage at each step.

B. Complexity Analysis

The computational complexity of the Greedy algorithm for the MVC problem can be analysed by focusing on the two main operations it performs during each iteration: **selecting the vertex with the maximum degree** and **removing its incident edges**.

- **Vertex Selection:** At each iteration, the algorithm scans all n vertices to find the one with the maximum degree, resulting in an $O(n)$ complexity per selection. This process continues until all edges are covered. In the worst scenario – such as a complete graph with $\frac{n(n-1)}{2}$ edges – the vertex selection step could contribute an $O(n^2)$ complexity. However, empirical data suggest that the number of selections grows more slowly that in the worst case, indicating that the average-case performance is better than $O(n^2)$.

- **Edge Removal:** Once the vertex with the maximum degree is selected, the algorithm removes all edges incident to it. Since each edge is removed exactly once over the algorithm's execution, the total complexity of edge removal is $O(m)$. The plots confirm that this step does not significantly increase the computational load, showing smooth growth in time and operations as graph size increases.

In summary, while the theoretical worst-case for this algorithm is $O(n^2)$, mainly due to the vertex selection process, empirical results indicate a much more efficient average-case performance. The algorithm shows polynomial growth in the number of operations and execution time, as demonstrated by the plots, and performs significantly better than the exhaustive search approach. This makes the greedy algorithm a practical solution for the

Minimum Vertex Cover problem, particularly for larger graphs where exhaustive search is infeasible.

C. Implementation

The code for the greedy algorithm is as the follows:

```
def greedy_vertex_cover(graph):
    cover = set()

    while graph.number_of_edges() > 0:
        v = max(graph.degree, key=lambda x: x[1])[0]

        cover.add(v)
        graph.remove_node(v)

    return cover
```

Algorithm 2: Greedy Search for Minimum Vertex Cover

In the *greedy_vertex_cover* function, in each iteration, as long as there are edges in the graph, it selects the vertex connected to the most uncovered edges (v), adding it to the cover until all edges are covered and then removing it and all its edges from the graph.

D. *Experimental Analysis*

To evaluate the performance of the greedy algorithm, it was conducted experiments similar to those in the exhaustive search analysis, using randomly generated graphs with varying sizes and edge densities. It is used the same metrics used previously too, with the difference in the number of tested sizes where $n \in [4, 256[$. These metrics allow us to compare the efficiency and accuracy of the greedy approach against the exact solutions obtained from exhaustive search for smaller graphs.

D.1 Number of Solutions Tested:

The greedy algorithm demonstrates a more controlled increase in solutions tested as graph size and edge density rise (Figure 4). While the rate of increase is notable in dense graphs, it remains significantly lower that the exhaustive search, reflecting its heuristic efficiency.

D.2 Number of Operations:

The greedy algorithm's operations show a sub-exponential increase with graph-size, as depicted in Figure 5. This pattern suggests better scalability and aligns with the algorithm's polynomial complexity. The curve's progression indicates that the increase in operations is

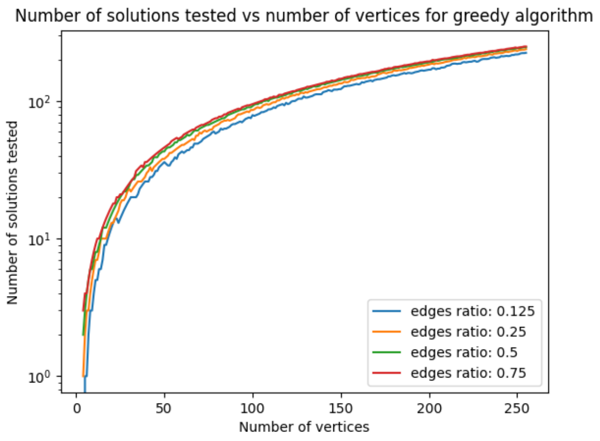

Figure 4: The Greedy Search algorithm's number of solutions tested in function of the number of vertices and edge density.

significant but remains polynomial, even in graphs with higher edge densities.

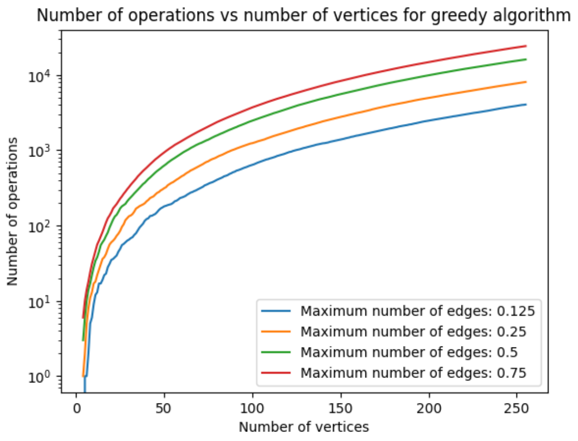

Figure 5: The Greedy Search algorithm's number of operations in function of the number of vertices and edge density.

D.3 Execution Time:

Regarding execution time, the Greedy algorithm demonstrates a gradual increase as seen in Figure 6. This gradual trend, which does not steeply rise with the number of vertices, indicates the algorithm's relatively efficient performance, even when dealing with larger and denser graphs. The execution time's modest ascent further validates the algorithm's suitability for more extensive graph instances where exhaustive search methods are impractical.

The analysis of the Greedy algorithm shows a clear, predictable increase in the number of solutions tested, operations performed, and execution time as graph size and density increase. Although the greedy algorithm consistently outperforms exhaustive methods, especially on larger and denser graphs, the result highlight the significance of considering both graph density and size when assessing its efficiency.

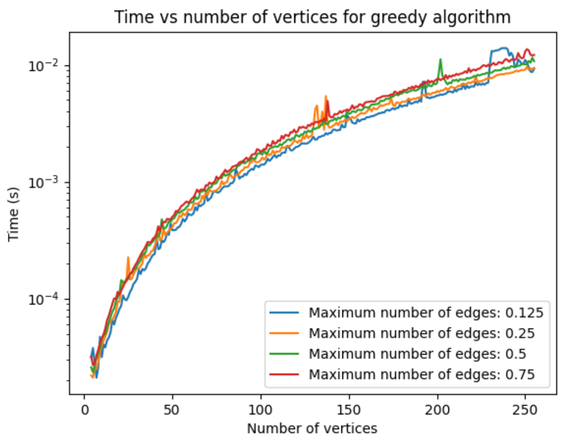

Figure 6: The Greedy Search algorithm's time execution in function of the number of vertices and edge density.

IV. CONCLUSION

The exhaustive algorithm was proven to have exponential time complexity, despite being guaranteed to find an optimal solution. Experimental results showed that both the number of operations and execution time increased exponentially with the size of the graph, aligning with theoretical expectations. This limits the algorithm's practical application to smaller graphs, where its exhaustive nature does not pose a computational barrier.

In contrast, the Greedy algorithm proved more practical for larger graphs. While it does not ensure an optimal solution, its experimental performance showed a substantial reduction in execution time compared to the Brute Force approach, especially as graphs grew larger and denser.

In conclusion, the choice of algorithm depends on graph size, density, and the need for an optimal solution. The exhaustive algorithm is suitable for small graphs or when optimality is non-negotiable. However, for larger graphs or when near-optimal solutions within practical time constraints are acceptable, the greedy algorithm is the more viable choice.

.

REFERENCES

[1] Brilliant.org. (n.d.). "*Vertex Cover*". Retrieved October 29, 2024, from https://brilliant.org/wiki/vertex-cover/.

[2] GeeksforGeeks. (n.d.). "*Introduction and approximate solution for Vertex Cover problem*". Retrieved November 8, 2024, from https://www.geeksforgeeks.org/introduction-and-approximate-solution-for-vertex-cover-problem/.

[3] Wikipedia. (n.d.). "*Vertex cover*". Retrieved November 10, 2024, from https://en.wikipedia.org/wiki/Vertex_cover.