1st Project - Maximum Weight Clique

Hugo Correia 108215

Resumo – Este relatório apresenta uma análise detalhada de algoritmos de otimização aplicados para identificar o clique de maior peso em grafos não orientados, onde cada vértice possui um peso positivo. Este estudo resulta do primeiro trabalho prático da disciplina de Algoritmos Avançados, realizado pelo autor. Inicialmente, o problema abordado pelos algoritmos é introduzido sob uma perspectiva teórica, definindo o contexto e os desafios da busca por cliques de peso máximo. Em seguida, cada algoritmo desenvolvido é examinado com uma análise formal de complexidade temporal, complementada por experimentos práticos que medem o desempenho computacional em termos de tempo de execução, número de operações realizadas e número de configurações testadas. Para o algoritmo heurístico, uma análise de precisão também é conduzida, comparando-o com métodos exatos. Adicionalmente, discute-se os limites de aplicabilidade de cada algoritmo, identificando o maior grafo processável sem comprometer o tempo de execução. Para cenários maiores, são feitas estimativas de tempo de execução com base nos padrões observados. Por fim, o relatório compara os resultados experimentais e teóricos, destacando as forças e fraquezas de cada abordagem e sugerindo casos de uso apropriados para os diferentes algoritmos.

Abstract – This report presents a detailed analysis of optimization algorithms designed to identify the maximum-weight clique in undirected graphs, where each vertex has a positive weight. This study is part of the first practical project in the Advanced Algorithms course, conducted by the author. The problem tackled by the algorithms is initially introduced from a theoretical perspective, defining the context and challenges in searching for maximum-weight cliques. Each developed algorithm is then examined through formal time complexity analysis, complemented by practical experiments that measure computational performance in terms of execution time, number of operations performed, and number of configurations tested. For the heuristic algorithm, an accuracy analysis is also conducted, comparing it to exact methods. Additionally, the limitations of each algorithm are discussed, identifying the largest graph that can be processed without excessive execution time. For larger instances, estimated execution times are provided based on observed patterns. Finally, the report compares experimental and theoretical results, highlighting the strengths and weaknesses of each approach and suggesting suitable use cases for the different algorithms.

I. Introduction

Optimization problems are central to computer science and operations research, involving the task of finding the best solution from all feasible solutions [1]. Specifically, an optimization problem seeks to maximize or minimize a particular objective function while satisfying a set of constraints. In graph theory, one prominent optimization challenge is identifying the maximum weight clique within an undirected graph where each vertex carries a positive weight [2]. Over the years, a multitude of algorithmic strategies have been devised to address such complex problems. Among these strategies are exhaustive search, backtracking search, and greedy algorithms. Each offers a different balance between computational efficiency and solution optimality.

An **exhaustive search** [3] algorithm systematically explores all possible subsets of vertices to guarantee the discovery of the optimal solution. While this method is straightforward and ensures optimality, it suffers from combinatorial explosion, making it impractical for large graphs due to its exponential time complexity.

The **backtracking search** [4] technique enhances the exhaustive approach by pruning branches of the search tree that cannot possibly lead to a better solution than the best one found so far [3]. This method reduces unnecessary computations by eliminating suboptimal paths early, thus improving efficiency while still ensuring an optimal solution is found.

In contrast, **greedy algorithms** [5] build a solution incrementally by making a sequence of choices, each of which is locally optimal at the moment [5]. Greedy methods are typically faster and simpler to implement but do not always yield a globally optimal solution because they do not reconsider their choices.

In this study, the author implemented and analyzed two different algorithms to find the maximum weight clique in an undirected graph with *v* vertices and *e* edges, only using the Backtracking Search for comparison purposes.

1. **Exhaustive Search Algorithm**: Explores all possible vertex subsets to identify the clique with the maximum total weight.

2. **Greedy Heuristic Algorithm**: Quickly constructs an approximate solution by iteratively selecting the vertex with the highest weight connected to the current clique.[6]

By comparing these algorithms, we aim to analyze their computational complexities, execution times, the number of operations performed, and the quality of solutions found. This comparative study provides insights into the trade-offs between computational effort and solution optimality inherent in different algorithmic approaches.

The remainder of this report is structured as follows. **Section II** provides a theoretical overview of the maximum weight clique problem. **Section III** introduces and analyses the *Exhaustive Search algorithm*, followed by the *Greedy Heuristic algorithm* in **Section IV**, where both the implementation details and formal complexity analyses are discussed. **Section V** presents a comparative analysis of the two algorithms, including an evaluation of accuracy and computational efficiency. Finally, **Section VI** concludes the report, summarizing key findings and considering potential improvements and future directions.

II. Maximum Weight Clique Problem - Theoretical Viewpoint

To address the maximum weight clique problem, it is essential first to understand the concept of a **clique** in graph theory. A **clique** is a subset of vertices in a graph *G* in which every pair of vertices is connected by an edge, forming a complete subgraph of *G*. In other words, all vertices within a clique are adjacent to one another, creating a fully connected subset. [7]

This work tackles the specific problem of identifying the **maximum-weight clique** in an undirected graph where each vertex is assigned a positive weight. A clique's **weight** is defined as the sum of the weights of the vertices that compose it. This optimization problem aims to find the clique with the greatest cumulative weight.

An important property of a clique used in the author's algorithms is the **number of edges** it contains. In a clique with v vertices, every possible pair of vertices is connected by an edge. Therefore, the number of edges E in a clique of v vertices can be calculated using the combination formula: [8]

$$E = \binom{v}{2} = \frac{v(v-1)}{2} \quad (1)$$

For instance, a clique with 3 vertices has exactly 3 edges, while a clique with 4 vertices has 6 edges. **Figure 1** illustrates an example of a complete subgraph (clique) K_4 within a larger graph G.

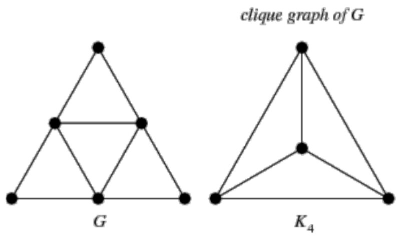


Fig. 1 - Example of a Clique K4 inside a Graph G.

The maximum weight clique problem is computationally challenging and is classified as NP-hard. This complexity is due to the combinatorial nature of finding subsets of vertices that meet the criteria of maximum weight and complete connectivity. In this study, the author approaches the problem with different algorithmic strategies, detailed in the following sections, and evaluates their effectiveness and efficiency in identifying maximum weight cliques in various graph structures.

III. Exhaustive Search Algorithm For Maximum Clique

The most straightforward approach to solving the maximum weight clique problem is through an algorithm based on the **Exhaustive Search** technique. In this approach, often referred to as brute force, all potential solutions are systematically generated, and each is evaluated to determine if it meets the problem's constraints. Despite being computationally intensive, this technique is guaranteed to find the optimal solution by exploring every possible subset. [3]

For a graph *G* with *v* vertices, the algorithm must generate all possible subsets of vertices and check each subset to determine if it forms a clique. If a subset is a clique, the algorithm calculates its weight by summing the weights of all vertices within the subset. Among all the cliques found, the algorithm identifies the one with the maximum weight.

The implementation of this algorithm involves two main parts:

- **Subset Generation**: Generate all possible subsets of vertices. This covers all possible configurations within the graph.
- **Subset Evaluation**: For each subset:
    - Verify if is a clique by ensuring that every vertex in the subset is adjacent to every other vertex within the subset.
    - Calculate the subset's weight by summing the weights of its vertices
    - If this weight exceeds the highest recorded weight, update the record with the current subset as the maximum weight clique

To confirm if a subset is a clique, the algorithm uses the graph's edge data. For a subset with *n* vertices, the subset forms a clique if it has exactly $(\frac{n}{2})$ edges, which represents all possible edges between pairs of vertices in the subset. If the subset contains fewer edges, it is discarded as it does not meet the clique criteria. This verification process, discussed in Section II, is critical to ensuring the accuracy of the algorithm.

A. Formal Analysis of the Algorithm's Complexity

To formally analyze the complexity of this algorithm, we start by defining the input size. In the maximum weight clique problem, the input size is primarily determined by the number of vertices v in the graph [9]. Given that the algorithm iteratively generates and evaluates all possible subsets of vertices to identify the optimal clique, the key computational task is the subset generation, which directly impacts execution time. [9]

The core operation in this exhaustive approach is subset generation and evaluation, making it the primary contributor to the algorithm's execution time. Since the process is exhaustive and not affected by input order, we can focus on analyzing the average-case complexity. The total number of possible subsets of v vertices, including the empty set, is $2^v - 1$, based on the Binomial Theorem: [10]

$$(x + y)^v = \sum_{i=0}^{v} \left(\left(\frac{v}{i}\right)\right) \times x^{(v-i)} \times y^i \qquad (2)$$

To facilitate, if we set $x = y = 1$, we obtain:

$$(1 + 1)^v = 2^v = \sum_{i=0}^{v} \left(\left(\frac{v}{i}\right)\right) \times 1^{(v-i)} \times 1^i = \sum_{i=0}^{v} \left(\left(\frac{v}{i}\right)\right) \quad (3)$$

However, this expansion includes the subset with zero elements, represented by $(\frac{v}{0})$. By excluding this choice, we are left with:

$$2^v - 1 = \sum_{i=1}^{v} \left(\left(\frac{v}{i}\right)\right) \qquad (4)$$

The complexity of generating these subsets, therefore, is proportional to $2^v - 1$, simplifying to $O(2^v)$.

For each generated subset, the algorithm performs additional operations to determine if it is a clique by checking all pairwise edges within the subset. If a subset has k vertices, checking for a clique involves verifying $\left(\frac{k}{2}\right)$ potential edges. This clique-checking operation is repeated for each subset, further reinforcing the exponential complexity of the algorithm. The overall complexity remains $O(2^v)$, as subset generation is the dominant operation, especially for large v.

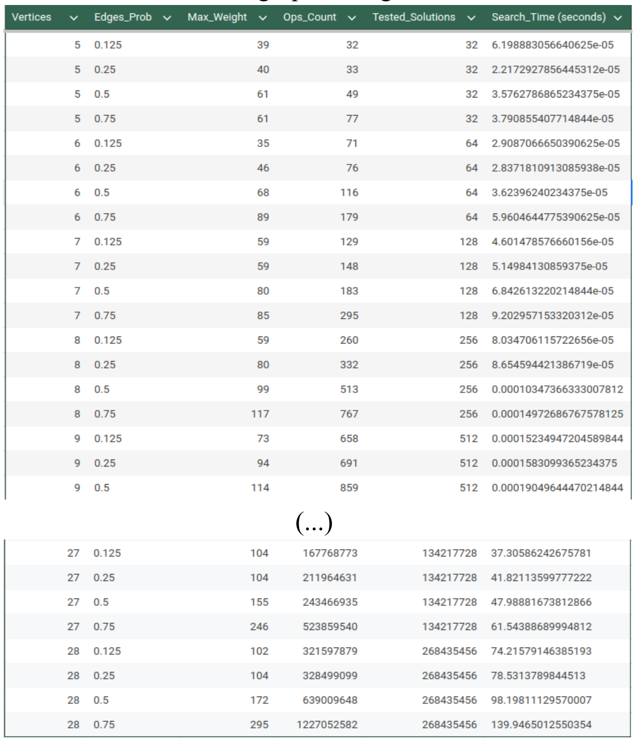
Exhaustive search algorithms that generate all subsets inherently exhibit exponential complexity, as documented in computational theory, as the number of vertices v increases, the algorithm's runtime scales exponentially, limiting its feasibility to small-to-moderate graph sizes.

In summary, the exhaustive search algorithm has a complexity $O(2^v)$, driven by the need to evaluate every subset of vertices. This formal analysis aligns with practical observations, as the algorithm's performance degrades rapidly with increasing v. [11]

B. Practical Analysis of the Algorithm's Complexity

To evaluate the performance of the Exhaustive Search and Greedy algorithms in finding the maximum weight clique within an undirected graph, the author conducted extensive experiments on graphs of increasing complexity. Specifically with Exhaustive Search, the experiments included graphs with up to 28 vertices. This experiment allowed for a deeper understanding of the algorithms' scalability and limitations. For each graph size, multiple configurations were generated by varying the edge probability (0.125, 0.25, 0.5, and 0.75), enabling to examine of how edge density impacts algorithm behavior.

During these experiments, several key metrics were recorded to capture the computational demands and efficiency of the algorithms. These metrics included the number of basic operations performed, execution time, the number of tested solutions (configurations evaluated to locate the maximum weight clique), and the maximum weight of the cliques found. The results of these experiments are summarized in **Table 1**, where the author present data on the operations count, tested solutions, and execution time for each graph configuration:



Vertices	Edges_Prob	Max_Weight	Ops_Count	Tested_Solutions	Search_Time (seconds)
5	0.125	39	32	32	6.198883056640625e-05
5	0.25	40	33	32	2.2172927856445312e-05
5	0.5	61	49	32	3.5762786865234375e-05
5	0.75	61	77	32	3.790855407714844e-05
6	0.125	35	71	64	2.9087066650390625e-05
6	0.25	46	76	64	2.8371810913085938e-05
6	0.5	68	116	64	3.62396240234375e-05
6	0.75	89	179	64	5.9604644775390625e-05
7	0.125	59	129	128	4.601478576660156e-05
7	0.25	59	148	128	5.14984130859375e-05
7	0.5	80	183	128	6.842613220214844e-05
7	0.75	85	295	128	9.202957153320312e-05
8	0.125	59	260	256	8.034706115722656e-05
8	0.25	80	332	256	8.654594421386719e-05
8	0.5	99	513	256	0.00010347366333007812
8	0.75	117	767	256	0.00014972686767578125
9	0.125	73	658	512	0.00015234947204589844
9	0.25	94	691	512	0.0001583099365234375
9	0.5	114	859	512	0.00019049644470214844
(...)					
27	0.125	104	167768773	134217728	37.30586242675781
27	0.25	104	211964631	134217728	41.82113599777222
27	0.5	155	243466935	134217728	47.98881673812866
27	0.75	246	523859540	134217728	61.54388689994812
28	0.125	102	321597879	268435456	74.21579146385193
28	0.25	104	328499099	268435456	78.5313789844513
28	0.5	172	639009648	268435456	98.19811129570007
28	0.75	295	1227052582	268435456	139.9465012550354

Table 1 - Part of the Experimental Data for Exhau. Search Algorithm

A quick analysis of the algorithm's operations count and search time using *Python* reveals an exponential fit $r^2 \simeq 1$. This supports the conclusions drawn from the formal analysis, confirming that the algorithm's average-case complexity is indeed exponential, $O(2^v)$. **Figures 2, 3,** and **4** provide graphical representations of the functions that best fit the data – with the Operations Count illustrated in **Figure 2** and the Search Time in **Figures 3** and **4**.

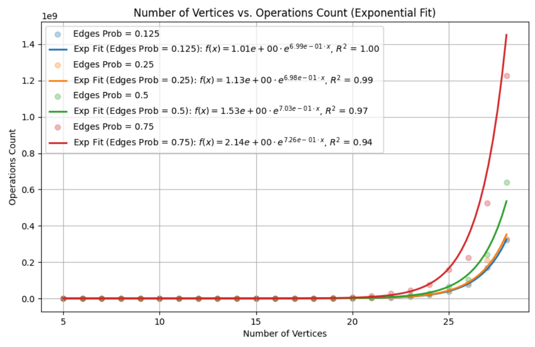


Fig. 2 - Exponential Fit ($f(x) = a \cdot e^{bx}$) for Operation Count in Exhaustive Search Algorithm with $r^2 \simeq 1$

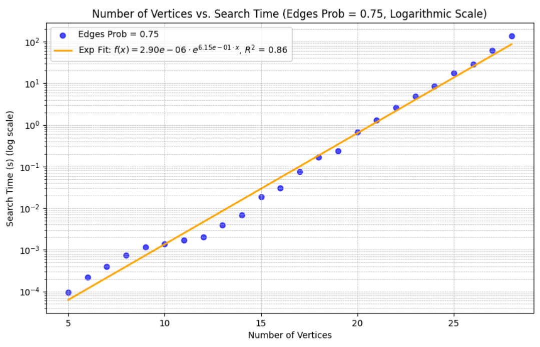


Fig. 3 - Logarithmic scale of vertices vs. search time (Edges Prob = 0.75), illustrating exponential growth in search time

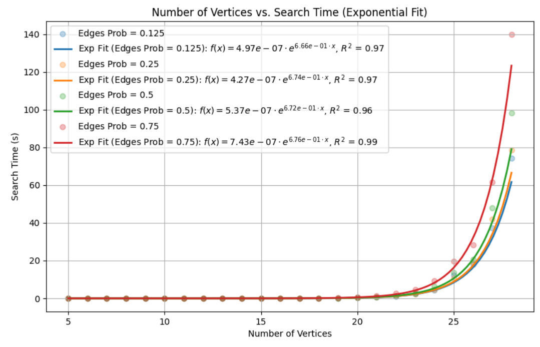


Fig. 4 - Exponential Fit ($f(x) = a \cdot e^{bx}$) for Search Time (seconds) in Exhaustive Search Algorithm with $r^2 \simeq 1$

C. Analysis Of The Algorithm's Limits

After determining the exponential function that best fits the correlation between Search Time and the Number of Vertices for this algorithm (with an edge probability of 75%), we can estimate the execution time required for larger graphs. For instance, by calculating $f(28)$, the time needed for a graph with 28 vertices can be approximated. This calculation is shown in equation (5):

$$f(28) = 7.43 \times 10^{-7} \times e^{0.676 \times 28} \approx 2\, minutes \tag{5}$$

This result indicates that for graphs with 28 vertices and a 75% edge density, the time required to execute the algorithm, above this number of vertices, becomes impractical.

Although this analysis was conducted specifically for graphs with 75% edge density, similar conclusions can be drawn for graphs with lower densities, as the algorithm's complexity remains exponential about the number of vertices, regardless of edge density. This exponential complexity implies that, as the number of vertices increases, the execution time will grow drastically, making the exhaustive search infeasible for larger graphs.

Extending this analysis further, we can estimate the time required for graphs with 40 vertices. The calculation yields:

$$f(40) = 7.43 \times 10^{-7} \times e^{0.676 \times 40} \approx 5\, days \tag{6}$$

The *Edge Probability vs. Search Time (Figure 5)* plot reveals that higher edge probabilities (0.125, 0.25, 0.5, and 0.75) contribute slightly to increased search times due to the added complexity of denser graphs. However, this effect is secondary to the impact of vertex count, which remains the primary driver of exponential growth in search time. Thus, while denser graphs require slightly more processing time, the overwhelming factor limiting this algorithm's practicality is the rapid increase in vertex count, which dominates the time complexity.

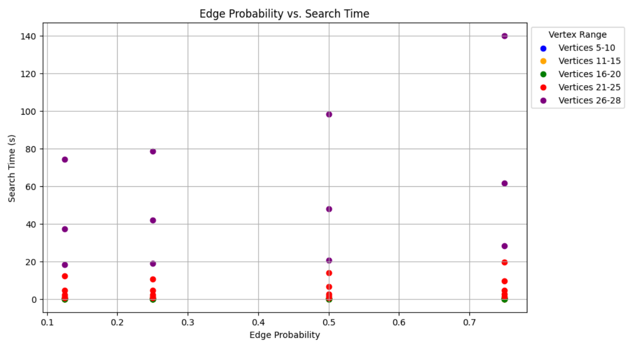


Fig. 5 - Edge Probability vs. Search Time for various vertex ranges, showing the relationship between edge density and computational time.

Such findings highlight the necessity of alternative algorithms, such as the Backtracking method, which can efficiently reduce the search space for graphs with a high number of vertices and connections.

D. Optimizations to improve the algorithm's efficiency

Building on the limitations identified in the previous section, it is clear that the exponential growth in execution time makes exhaustive searches impractical for large graphs. To address this challenge, several optimizations can be introduced to improve the algorithm's efficiency without altering its fundamental approach.

One such enhancement is vertex **pruning**: eliminating isolated vertices (vertices with zero degree) from the graph before processing. Since isolated vertices cannot be part of a clique, filtering them out at the outset reduces the overall number of vertex subsets, which in turn decreases the combinations the algorithm must evaluate. This enables quicker computation, making it feasible to handle graphs with a higher vertex count:

```
def remove_isolated_vertices(self, graph):

        isolated_nodes = [node for node, degree
        in graph.degree() if degree == 0]

        graph.remove_nodes_from(isolated_nodes)

        return graph
```

Snippet 1 - Function that removes all the isolated vertices from the graph to optimize search

The **backtracking search algorithm** represents further optimization by incorporating a branching approach with bounds to prevent redundant calculations. It begins with a set of candidate vertices and attempts to expand a clique incrementally. At each step, it calculates the maximum potential weight for any clique that could include the current clique and remaining candidates. If this potential weight is less than the current maximum found, the branch is pruned immediately, preventing unnecessary exploration. Additionally, vertices are initially sorted by weight, which allows the algorithm to prioritize vertices with higher weights and find promising cliques sooner.

Although these optimizations can improve the algorithm's performance, they were excluded from the formal experimental analysis. Including them would introduce variability in the results, making it challenging to draw consistent conclusions on the baseline performance of the exhaustive approach. Nevertheless, these strategies offer theoretical insights for potentially more scalable solutions.

IV. Greedy Algorithm For Maximum Weight Clique

As discussed in the previous section, the Exhaustive Search Algorithm becomes impractical for larger graphs due to its exponential complexity, $O(2^v)$ severely limiting its scalability. To address this limitation, the author implemented a Greedy Algorithm to approximate the maximum weight clique solution for graphs with higher vertex counts (more than 28 vertices). This alternative approach sacrifices optimality for efficiency, allowing it to handle larger graphs that would otherwise be infeasible with Exhaustive Search.

The Greedy approach constructs the solution incrementally by making a sequence of locally optimal choices, as suggested in classical greedy heuristics for optimization problems of evaluating all possible subsets, the algorithm builds a clique by initially selecting the vertex with the highest weight. At each subsequent step, it adds the highest-weighted neighbor that maintains clique connectivity, aiming to maximize the total weight of the clique without backtracking.

This Greedy Algorithm can be described through the following sequence of steps:

1. Sort all vertices in descending order based on their weights.
2. For each vertex in this sorted list, consider it as a potential starting point for building a clique:
    a. Initialize a list L with this starting vertex.
    b. Set the initial weight of the clique to the weight of the starting vertex.
    c. Identify all neighbors of this vertex as candidates for clique expansion.
3. While there are remaining candidates:
    a. Select the candidate with the highest weight.
    b. Check if this candidate is connected to all vertices in L
        i. If so, add this vertex to L and update the clique weight.
        ii. Update the candidate list to retain only neighbors connected to all vertices in the current clique.
        iii. If the candidate is not connected to all vertices in L, remove it from the list.
4. Continue this process until no more vertices can be added to the clique.

The list L will ultimately contain the vertices of an approximate maximum weight clique, and the sum of the weights in L will give the weight of this clique. While this approach is computationally efficient, it may miss the optimal solution since it only considers local choices. This limitation is apparent in cases where the optimal clique does not include the vertex with the highest individual weight or if the highest-weight vertex has limited connectivity. In the analysis section, we compare the results of this Greedy Algorithm with those from the Exhaustive Search better to understand these trade-offs in efficiency and solution quality.

A. Formal Analysis of the Algorithm's Complexity

This algorithm is based on an iterative approach, so to formally analyze its complexity, we begin by defining its input. Here, the input consists of *the graph's v* vertices and *e* edges, for which the author aimed to find the maximum weight clique. The next step involves identifying the algorithm's basic operation — the one that dominates the time complexity. In the Greedy Algorithm, the basic operation is determining whether each candidate vertex is adjacent to all vertices in the current clique, which is necessary to confirm whether it can be added to the clique. [9]

The complexity remains stable regardless of the ordering of vertices or edges in the graph; hence, we are only interested in the average-case complexity. Analyzing the best and worst cases is unnecessary in this context, as the algorithm's structure does not change. The basic operation of verifying adjacency is nested within two loops, as shown in code *snippet 2*:

```
for starting_vertex in vertices_sorted:
   current_clique = [starting_vertex]
   current_weigh=self.graph.nodes[starting_vertex]['weight']
   candidates=set(self.graph.neighbors(starting_vertex))
   while candidates:
     best_candidate = max(candidates, key=lambda v:
self.graph.nodes[v]['weight'])
     is_valid=all(self.graph.has_edge(best_candidate, node) for
node in current_clique)
     performed_operations+=len(current_clique
    if is_valid:
      current_clique.append(best_candidate)
      current_weight+=self.graph.nodes[best_candidate]['weight']
      candidates=candidates.intersection(self.graph.neighbors(
best_candidate))
    else:
      candidates.remove(best_candidate)
```

Snippet 2 - Function that examines each candidate's neighbors to check adjacency.

Thus, a closed formula can be derived for the algorithm's time complexity by summing over all vertices and the edges they connect to in the graph:

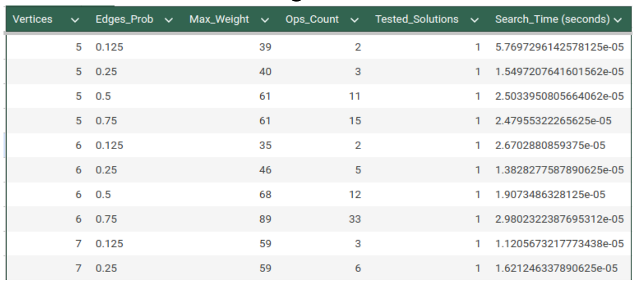
$$\sum_{i=0}^{v-1}\left(\sum_{j=0}^{e-1} 1\right) = v \cdot e \qquad (7)$$

This leads to an average-case complexity $O(v \cdot e)$, where *v* represents the vertices and *e* represents the edges in the graph. Consequently, the algorithm exhibits quadratic complexity $O\left(v^2\right)$ in graphs where each vertex is densely connected. This result aligns with expectations, as the basic operation depends on checking connections across pairs of vertices, typical for algorithms with nested loops over vertices and edges. [11]

B. Practical Analysis of the Algorithm's Complexity

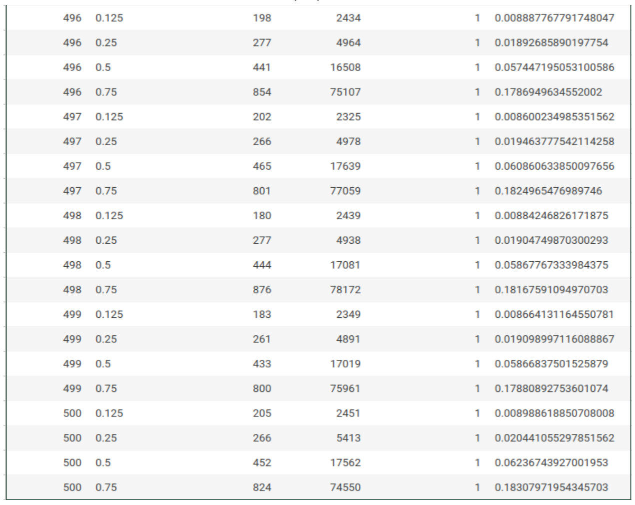
The Greedy Algorithm was evaluated using the same set of generated graphs as in the Exhaustive Search, with the number of vertices ranging up to 500. For each vertex count, we generated four graph versions with varying edge densities: 12.5%, 25%, 50%, and 75% of the maximum possible edges, as can be consulted in *Table 2*. This configuration resulted in a comprehensive dataset of 2000 graphs, ensuring a wide range of scenarios to test the algorithm's efficiency.

With the Greedy Algorithm, we successfully obtained solutions for graphs with up to 500 vertices and 75% edge density within a feasible runtime — taking less than a second in most cases. This demonstrates a significant efficiency improvement over the Exhaustive Search, which became impractical at much smaller graph sizes. Further comparison between these two approaches is discussed in the following sections.



Vertices	Edges_Prob	Max_Weight	Ops_Count	Tested_Solutions	Search_Time (seconds)
5	0.125	39	2	1	5.7697296142578125e-05
5	0.25	40	3	1	1.5497207641601562e-05
5	0.5	61	11	1	2.5033950805664062e-05
5	0.75	61	15	1	2.47955322265625e-05
6	0.125	35	2	1	2.6702880859375e-05
6	0.25	46	5	1	1.3828277587890625e-05
6	0.5	68	12	1	1.9073486328125e-05
6	0.75	89	33	1	2.9802322387695312e-05
7	0.125	59	3	1	1.1205673217773438e-05
7	0.25	59	6	1	1.621246337890625e-05

(...)



Vertices	Edges_Prob	Max_Weight	Ops_Count	Tested_Solutions	Search_Time (seconds)
496	0.125	198	2434	1	0.008887767791748047
496	0.25	277	4964	1	0.01892685890197754
496	0.5	441	16508	1	0.057447195053100586
496	0.75	854	75107	1	0.1786949634552002
497	0.125	202	2325	1	0.008600234985351562
497	0.25	266	4978	1	0.019463777542114258
497	0.5	465	17639	1	0.060860633850097656
497	0.75	801	77059	1	0.1824965476989746
498	0.125	180	2439	1	0.00884246826171875
498	0.25	277	4938	1	0.01904749870300293
498	0.5	444	17081	1	0.05867767333984375
498	0.75	876	78172	1	0.18167591094970703
499	0.125	183	2349	1	0.008664131164550781
499	0.25	261	4891	1	0.019098997116088867
499	0.5	433	17019	1	0.05866837501525879
499	0.75	800	75961	1	0.17880892753601074
500	0.125	205	2451	1	0.008988618850708008
500	0.25	266	5413	1	0.020441055297851562
500	0.5	452	17562	1	0.06236743927001953
500	0.75	824	74550	1	0.18307971954345703

Table 2 - Part of the Experimental Data for Greedy Search Algorithm

With the Greedy Algorithm, were obtained successfully solutions for graphs with up to 500 vertices and 75% edge density within a feasible runtime — taking less than 0.2 seconds in the worst case. This demonstrates a significant efficiency improvement over the Exhaustive Search, which became impractical at much smaller graph sizes. Further comparison between these two approaches is discussed in the following sections.

Analyzing the results with Python, the author found that a quadratic function provides a strong fit to the Operations Count metric, with an $r^2 = 1$. This high correlation validates the theoretical complexity derived in the formal analysis, confirming that the Greedy Algorithm exhibits quadratic complexity in the average case.

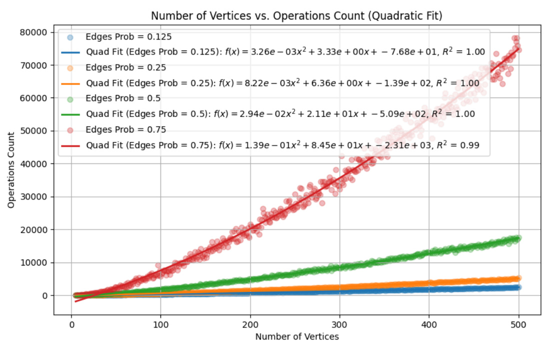


Fig. 6 - Quadratic Fit ($f(x) = ax^2 + bx + c$) for Operation Count in Greedy Search Algorithm with $r^2 \simeq 1$

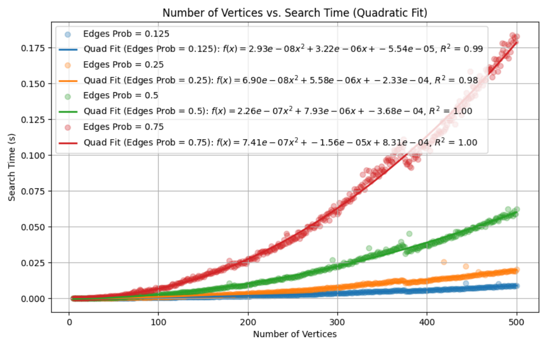


Fig. 7 - Quadratic Fit ($f(x) = ax^2 + bx + c$) for Search Time (seconds) in Greedy Search Algorithm with $r^2 \simeq 1$

In **Fig. 6** and **Fig. 7**, we can find graphical representations of the functions that best fit the data – Operations Count in **Fig. 6** and Search Time in **Fig. 7** for the Greedy algorithm.

C. Analysis Of The Algorithm's Limits

With the function that best fits the correlation between Search Time and the Number of Vertices for the Greedy algorithm (using a 75% edge probability), we can project the time required to find the maximum weight clique for larger graphs. This allows us to estimate the maximum size of a graph that the algorithm could handle within a practical timeframe. To illustrate, let's predict the search time for a graph with 500 vertices and a 75% edge density. Using the quadratic fit function:

$$f(500) = 7.41 \times 10^{-7} \cdot 500^2 - 1.56 \times 10^{-6} \cdot 500 + 8.31 \times 10^{-4} = 0.184 \quad (8)$$

For a graph with 500 vertices, the algorithm would require approximately 0.184 seconds, making it feasible to use the Greedy algorithm within a practical timeframe. The author then attempted to scale the experiment further, aiming to generate and test graphs with even larger vertex counts. However, due to computational limitations, were unable to create or process graphs with more than 1000 vertices. Nevertheless, based on the quadratic fit function, we can estimate that for a graph with 5000 vertices, the algorithm would require:

$$f(5000) = 7.41 \times 10^{-7} \cdot 5000^2 - 1.56 \times 10^{-6} \cdot 5000 + 8.31 \times 10^{-4} = 18.42 \quad (9)$$

This result suggests that, even with 5000 vertices, the algorithm would take around 18.42 seconds, which remains within a practical timeframe, albeit close to the limit for acceptable quick computation. This analysis underscores that, while the Greedy algorithm significantly outperforms Exhaustive Search, it still faces practical constraints as graph size increases, particularly with limited computational resources.

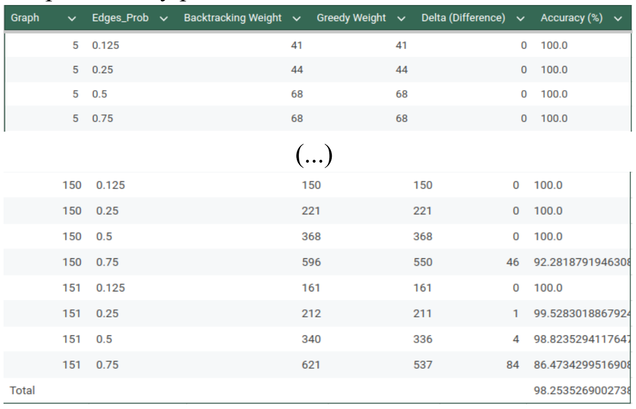
V. Comparison Between Exhaustive Search and Greedy Heuristic

Throughout this study, the author implemented and tested two algorithms—Exhaustive Search and Greedy Heuristic—to address the Maximum Weight Clique problem. Each algorithm has distinct strengths and weaknesses that make it suitable for different applications.

The Exhaustive Search algorithm's primary advantage is its accuracy; by evaluating all possible combinations, it guarantees finding the optimal solution. However, this comes at a cost of significant computational resources due to its exponential time complexity, making it impractical for larger graphs. The Greedy algorithm, by contrast, is efficient with a lower time complexity (quadratic on average), which enables it to handle larger graphs in a reasonable time. Nonetheless, this efficiency is achieved by prioritizing local optimal choices, which may not always yield the global optimal solution.

Accuracy Analysis of the Greedy Algorithm

In testing the Greedy algorithm against the Backtracking approach, the author evaluated its performance across graphs with varying numbers of vertices and edge densities. **Table 3** presents a summary of these results, showing the accuracy of the Greedy algorithm as compared to the Backtracking (Exhaustive Search) algorithm. The results indicate that, from 5 to 151 vertices, the Greedy algorithm achieved an average accuracy of 98.25%. This confirms the Greedy algorithm's viability as a faster, albeit slightly less precise, alternative for large graph instances where exact solutions are computationally prohibitive.



Graph	Edges_Prob	Backtracking Weight	Greedy Weight	Delta (Difference)	Accuracy (%)
5	0.125	41	41	0	100.0
5	0.25	44	44	0	100.0
5	0.5	68	68	0	100.0
5	0.75	68	68	0	100.0
(...)					
150	0.125	150	150	0	100.0
150	0.25	221	221	0	100.0
150	0.5	368	368	0	100.0
150	0.75	596	550	46	92.2818791946308
151	0.125	161	161	0	100.0
151	0.25	212	211	1	99.5283018867924
151	0.5	340	336	4	98.8235294117647
151	0.75	621	537	84	86.4734299516908
Total					98.2535269002738

Table 3 - Part of the Comparison Between the Weight Max Results between Greedy and Backtracking Algorithms

To highlight where the Greedy algorithm diverged from the optimal solution, we provide a plot showing the discrepancies for graphs with an edge probability of 0.5. This plot focuses only on cases where the Greedy weights did not match the optimal Backtracking results, with the Greedy weights shown as blue points and Backtracking results as orange, connected by light gray lines to illustrate the differences in each graph instance.

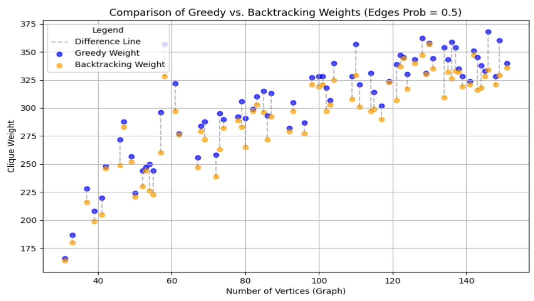


Fig. 8 - Comparison between the Max Weight Results of Greedy and Backtracking (plot 0.5)

This comparison suggests that while the Greedy algorithm is a practical choice for larger graphs, the Exhaustive Search remains the preferred method for smaller, highly constrained graphs where obtaining the optimal solution is essential.

VI. Conclusions

This study investigated the Maximum Weight Clique problem through the implementation and comparison of Exhaustive Search and Greedy Heuristic algorithms. Each approach revealed distinct advantages and limitations, providing insights into their practical applicability based on graph size and desired accuracy.

The Exhaustive Search algorithm, while guaranteed to find the optimal solution, was shown to be infeasible for large graphs due to its exponential time complexity. Experiments confirmed that beyond a certain threshold, the computational time required made its use impractical. This approach is therefore best suited for small-scale problems where absolute accuracy is essential and computational resources are less constrained.

In contrast, the Greedy Heuristic algorithm demonstrated significant efficiency, operating with quadratic complexity. Although this method does not guarantee an optimal solution, it achieved an impressive average accuracy of 98.25% for graphs with up to 151 vertices. The Greedy algorithm's speed and scalability make it a valuable tool for applications involving large, dense graphs where rapid, approximate solutions are acceptable.

Ultimately, this comparison underscores the trade-offs between accuracy and efficiency. The Greedy algorithm, with its near-optimal results and high scalability, emerges as a practical solution for large graph datasets, where exhaustive methods would be computationally prohibitive. This study highlights the importance of choosing an algorithm based on the problem scale and the acceptable margin of error, positioning the Greedy approach as a viable choice for real-world applications where performance is prioritized over exact precision. [4]

References

[1] Li, Meiyi, et al. "[2208.10611] Learning to Solve Optimization Problems with Hard Linear Constraints." arXiv, 22 August 2022, https://arxiv.org/abs/2208.10611 . Accessed 8 November 2024

[2] Rozman, K., Ghysels, A., Janežič, D. et al. An exact algorithm to find a maximum weight clique in a weighted undirected graph. Sci Rep 14, 9118 (2024). https://doi.org/10.1038/s41598-024-59689-x

[3] A. Levitin and A. Levitin, "Brute-Force and Exhaustive Search," in Introduction to the design and analysis of algorithms, Boston, Boston: Pearson, 2012, pp. 115–116

[4] Erhardt, Roman, et al. "[2302.00458] Improved Exact and Heuristic Algorithms for Maximum Weight Clique." arXiv, 1 February 2023, https://arxiv.org/abs/2302.00458 . Accessed 6 November 2024.

[5] A. Levitin, "Greedy Technique," in Introduction to the design and analysis of algorithms, 3rd ed., Boston, Boston: Pearson, 2012, pp. 315–318

[6] Grosso, A., Locatelli, M. & Croce, F.D. Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem. *Journal of Heuristics* **10**, 135–152 (2004). https://doi.org/10.1023/B:HEUR.0000026264.51747.7f

[7] Bomze, I. M.; Budinich, M.; Pardalos, P. M.; Pelillo, M. (1999), "The maximum clique problem", *Handbook of Combinatorial Optimization*, vol. 4, Kluwer Academic Publishers, pp. 1–74, CiteSeerX 10.1.1.48.4074

[8] MikeFHay and mdp, "How many edges are in a clique of n vertices?," Mathematics Stack Exchange, 01-Dec-1959. [Online]. https://math.stackexchange.com/questions/199695/howmany-edges-are-in-a-clique-of-n-vertices. [Accessed: 05-Nov2024]

[9] A. Levitin, "Fundamentals of the Analysis of Algorithm Efficiency," in Introduction to the design and analysis of algorithms, Boston, Boston: Pearson, 2012, pp. 43–50

[10] "Binomial theorem," Wikipedia, 09-Oct-2022. [Online]. Available: https://en.wikipedia.org/wiki/Binomial_theorem . [Accessed: 05-Nov-2024]

[11] A. Levitin, "Fundamentals of the Analysis of Algorithm Efficiency," in Introduction to the design and analysis of algorithms, 3rd ed., Boston, Boston: Pearson, 2012, pp. 58–59

.