Maximum Weight Clique

Hugo Gonçalves

Resumo – **O presente artigo apresenta a análise detalhada a algoritmos de otimização usados para obter o clique com maior peso dentro de um grafo não orientado cujos vertices têm pesos positivos. Este artigo é uma compilação do resultado do primeiro trabalho prático da cadeira de Algoritmos Avançados efetuado pelo autor.**

Ao longo do presente documento é apresentado o problema que os algoritmos pretendem resolver de um ponto de vista teórico. De seguida, inicia-se a análise de cada um dos algoritmos desenvolvidos onde é feita uma análise em termos de complexidade temporal recorrendo a métodos formais seguida de análise dos resultados práticos obtidos. Para cada um dos algorimos é feita uma discussão dos resultados, onde se compara os resultados obtidos através da análise formal contra os resultados práticos obtidos. Para além disso, são ainda apresentados os limites de cada um dos algoritmos e quanto tempo estes algoritmos demorariam a encontrar uma solução nessas casos limite. Por último, tendo em conta os resultados analisados, apresenta-se uma comparação entre os dois algoritmos bem como alguns casos de uso onde cada um deles pode ser utilizado.

Abstract – **This paper presents a detailed analysis of optimization algorithms to get the maximum weight clique in an undirected graph whose vertices carry positive weights. It is a compilation of the results obtained for the first practical assignment of Algoritmos Avançados developed by the author. This paper first presents an overview of the problem these algorithms intend to address. Afterward, the paper presents the time complexity analysis for each algorithm developed using formal methods and the practical results obtained. For each algorithm there is a discussion over the obtained results comparing the formal results with the practical results obtained by running the algorithms and a prediction of the limits of each algorithm and how long would it take for them to reach that limit. Finally, considering the algorithms' analysis, the paper presents a comparison between both algorithms and some use cases for each one of them.**

I. INTRODUCTION

All solvable problems have at least one optimal solution. An optimization problem intends to find a solution that will maximize or minimize some kind of metric associated with the problem [1]. Optimization algorithms are mathematical tools, that can be implemented using computers and will comparate several solutions for the problem and find the best one, i.e., the most optimal one [1].

Throughout the history of computers, several algorithms were developed to address different types of problems. These algorithms are based on different techniques such as Brute-Force, Divide-And-Conquer, Decrease-and-Conquer, Exhaustive Search, Greedy Algorithms, etc. For the sake of this assignment, the two most important techniques are Exhaustive Search and Greedy Technique.

An Exhaustive Search is a Brute-Force approach to combinatory problems that suggests generating every element of the problem domain and selecting the one that returns the optimal solution [2]. Although this technique is slow for problems with a big input, it is the easiest to implement and will always find the global optimal solution.

A Greedy Technique is a general algorithm design technique that is only applicable to optimization problems. This technique suggests constructing a solution based on several consecutive steps [3]. In each one of the steps, we improve our solution until the solution to the problem is reached [3]. The Greedy Technique is usually straightforward to implement and very fast to get a solution to the problem. However, in some cases, algorithms based on Greedy Techniques may not find an optimal solution for the problem, as they can only find the local optimal solution in each step [3], as we will see later on.

During the practical assignment on which this paper is based, the author implemented two different algorithms based on these two approaches: Exhaustive Search and search using Greedy heuristics. These implementations allowed us to see, in practice, the difference in the time complexity and the solutions found between the two approaches. Both algorithms were used to find the maximum weight clique for an undirected graph with v vertices and e edges. More details about the problem explored throughout the assignment can be found in the next section.

II. MAXIMUM WEIGHT CLIQUE PROBLEM – THEORETICAL VIEWPOINT

In order to understand the problem is important to know what a Clique is. In Graph Theory, a Clique is a subset of vertices of a graph G, all adjacent to each other defining a complete subgraph of G [4]. The proposed

problem was to find the maximum weight clique inside an undirected graph in which the vertices carry positive weights and the weight of the clique is the sum of the weights of the vertices that are part of the clique.

It is very important to note that in a clique, all the

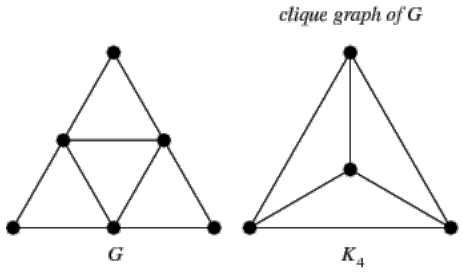

Fig. 1 - Example of a Clique K_4 inside a Graph G.

vertices must be adjacent to each other, i.e., all vertices must be connected to the others.

Another very important property of a Clique, that we use in our algorithms to check if a subset of vertices is a Clique – more details in the next sections –, is the number of edges of a clique with *v* vertices. In a Clique, all vertices must be connected to each other, therefore, there is one edge for each choice of two vertices from *v*. Consequently, mathematically, the number of edges for a clique with *v* vertices can be represented by [5]:

$$\binom{n}{x} = \frac{n!}{2!*(n-2)!} = \frac{1}{2} * n(n-1) \quad (1)$$

Applying the formula we get that a clique with 3 vertices may have 3 edges, as well as a clique with 4 vertices, must have 6 edges – as we can confirm in Fig. 1.

III. EXHAUSTIVE SEARCH ALGORITHM FOR MAXIMUM WEIGHT CLIQUE

The most simple and straightforward approach for solving the proposed problem (getting the maximum weight clique) is to create an algorithm based on the Exhaustive Search technique. The Exhaustive Search Technique is a Brute-Force in which all the possible solutions are generated and each one of them is tested to check if it satisfies the problem constraints.

In this case, for a graph G with *v* vertices, we need to generate all the possible subsets of vertices and check each subset to know if it is a clique or not. If a clique is found, we calculate its weight (summing the weight of all vertices) and save it. When we have checked all the subsets we can then sort the list of cliques to get the one with the maximum weight.

The algorithm can be defined as follows in a sequence of steps:

1. Generate all possible combinations of vertices (vertices subsets).
2. Iterate through all the subsets and check if each one of the subsets is a clique, i.e., check if there is at one edge between any pair of vertices.
3. If the subset is a clique, calculate its weight by summing its vertices' weights.
4. Store the clique with its weight in a data structure if its weight is higher than the one already stored.
5. Get the maximum weight clique by reading the values in the data structure.

An implementation for this algorithm would be divided into two different parts. In the first one, we would generate all the vertices' subsets for the graph. In the second part, we would iterate through all the generated subsets, check if they are a clique, and update a variable with the maximum clique and its weight.

To check if the subset is a clique we need to not only have all the vertices in the subset but also a list with all edges in graph G. With this input, we count the number of edges in the subset and check if the number of edges is equal to ${}_nC_2$, if it is, then we have a clique if the number of edges is inferior, however, then the subset is not a clique and can be discarded. The theory behind this approach was discussed in the previous section II.

A. Formal Analysis of the Algorithm's Complexity

Once this algorithm is an iterative algorithm, to make a proper formal analysis of its complexity we must start by defining the size of the input [6]. In this case, we can define the input as being *v* vertices. Next, we need to define the *basic operation* of the algorithm, which is the operation that contributes the most to the execution time of the algorithm [6]. In this case, the *basic operation* would be the generation of the subsets. Once the number of times the *basic operation* is performed does not change depending on the order or on the way data is organized in the input, we are only interested in the *average case* [6].

The generation of subsets is done using combinations of all *v* vertices. The total number of combinations from a set with *v* elements is $2^v - 1$, according to the *Binomial Theorem* [7]:

$$(x+y)^v = \sum_{i=0}^{v} \left(\binom{v}{i} \right) * x^{(v-i)} * y^i \quad (2)$$

If we set x = y = 1:

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

However, this equation contains the choice of zero elements, $\binom{v}{0}$. Removing the choice of those elements we get that:

$$2^v - 1 = \sum_{i=0}^{v} \left(\binom{v}{i} \right) \quad (4)$$

Therefore, a closed formula for the complexity to generate all subsets for *v* vertices is $2^v - 1$. So, in the *average case*, the complexity of the Exhaustive Search

algorithm is $O(2^n)$ with n being the number of vertices in the graph – an algorithm with **exponential complexity**.

These results were expected as, usually, algorithms that generate all subsets of an n-element set have, typically, an exponential complexity as described in [8].

B. Practical Analysis of the Algorithm's Complexity

To test the algorithms in practice we generated graphs from 5 to 200 vertices. For each number of vertices (5,6,7,..,200) were generated 4 different graphs. Each one of these graphs contained a different amount of edges that was, respectively, 12.5% 25%, 50%, and 75% of the maximum number of edges possible for the number of vertices.

The Exhaustive Search algorithm was run over each one of those successively larger graphs and some important metrics were registered for each experiment. The results can be consulted in Table 1.

Vertices	Edges	Max Weight Clique	Operations Count	Tested Solutions	Search Time
5	12,50%	68	33	32	0,00004
5	25,00%	69	34	32	0,00002
5	50,00%	100	52	32	0,00006
5	75,00%	117	81	32	0,00010
6	12,50%	80	65	64	0,00003
6	25,00%	80	79	64	0,00005
6	50,00%	83	141	64	0,00015
6	75,00%	103	229	64	0,00031
7	12,50%	79	130	128	0,00005
7	25,00%	97	158	128	0,00012
7	50,00%	100	388	128	0,00049
7	75,00%	149	593	128	0,00090
8	12,50%	68	277	256	0,00014
8	25,00%	97	410	256	0,00039
8	50,00%	128	844	256	0,00121
8	75,00%	137	1579	256	0,00293
9	12,50%	97	544	512	0,00026
9	25,00%	100	773	512	0,00076
9	50,00%	148	2294	512	0,00382
9	75,00%	179	3752	512	0,00776
10	12,50%	73	1069	1024	0,00048
10	25,00%	80	2047	1024	0,00269
10	50,00%	132	5842	1024	0,01146
10	75,00%	180	9175	1024	0,02219
11	12,50%	87	2324	2048	0,00159
11	25,00%	87	3738	2048	0,00516
11	50,00%	140	12362	2048	0,02783
11	75,00%	210	22630	2048	0,06493
12	12,50%	88	4544	4096	0,00293
12	25,00%	100	10272	4096	0,01764
12	50,00%	136	32080	4096	0,08216
12	75,00%	222	53733	4096	0,16933
13	12,50%	79	8795	8192	0,00514
13	25,00%	96	18870	8192	0,03428
13	50,00%	127	79016	8192	0,22753
13	75,00%	166	126280	8192	0,45365
14	12,50%	88	19673	16384	0,01823
14	25,00%	100	51276	16384	0,11098
14	50,00%	146	187249	16384	0,60997
14	75,00%	194	294028	16384	1,20417
15	12,50%	96	37682	32768	0,03238
15	25,00%	109	94648	32768	0,22030
15	50,00%	169	401916	32768	1,43916
15	75,00%	242	670652	32768	3,05714
16	12,50%	96	87601	65536	0,11194
16	25,00%	140	259816	65536	0,69072
16	50,00%	169	960316	65536	3,71940
16	75,00%	208	1530556	65536	7,59656
17	12,50%	136	164069	131072	0,19438
17	25,00%	136	469338	131072	1,33908
17	50,00%	177	2227308	131072	9,82228
17	75,00%	253	3461066	131072	18,79617
18	12,50%	140	309967	262144	0,33611
18	25,00%	140	1262798	262144	4,03306
18	50,00%	173	4719620	262144	21,80622
18	75,00%	231	7717630	262144	46,24904
19	12,50%	136	721730	524288	1,14299
19	25,00%	136	2255780	524288	7,79473
19	50,00%	198	10866238	524288	54,87886
19	75,00%	249	17281792	524288	111,76274
20	12,50%	128	1338744	1048576	2,23729
20	25,00%	128	6066390	1048576	23,07495
20	50,00%	169	24753736	1048576	135,83351

Table 1: Experimental data for Exhaustive search algorithm

With this algorithm, we could only obtain solutions up to 20 vertices and 50% of edges in useful time (a timeout of 2 minutes was implemented). For 20 vertices and 50% of edges the computer took more than 2 minutes to compute the solution.

Doing a quick analysis of the algorithm's *operations count* and *search time* using MatLab we can find an exponential fit with r^2= 1. This confirms our conclusions from the formal analysis – the complexity of this algorithm for the *average case* is indeed **exponential** $O(2^n)$. In Fig. 2 and Fig. 3 we can find graphical representations of the functions that best fit the data – *Operations Count* in Figure 2 and *Search Time* in Figure 3.

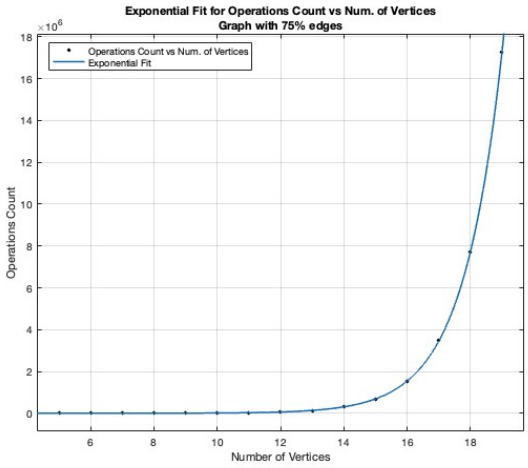

Fig. 2 - Exponential Fit (f(x) = $3.837 * e^{0.8063*x}$) for Operations Count in Exhaustive Search algorithm.

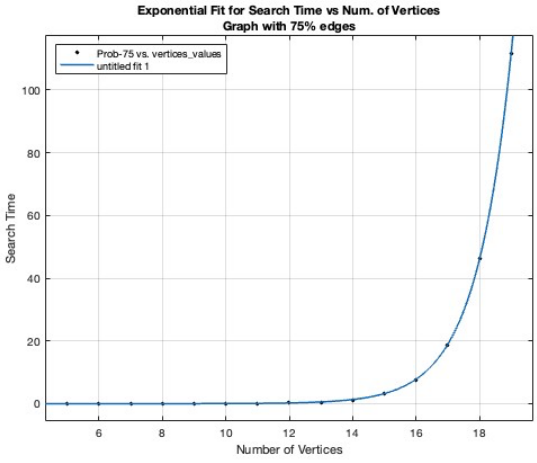

Fig. 3 - Exponential Fit (f(x) = $5.214 * 10^{-6} * e^{0.8885*x}$) for Search Time in Exhaustive Search algorithm.

C. Analysis of the algorithm's limits

Once we have the formula for the exponential function that best fits the correlation Search Time vs Number of Vertices for this algorithm (using 75% of edges), we can easily know how much time would take to run this algorithm for a higher number of vertices and 75% of

edges. For 20 vertices, for example, we can calculate $f(20)$ and know exactly how much time it would take. This calculation is done in (5).

$$f(20) = 5.214 * 10^{-6} * e^{(0.8885*20)} = 272.00943s \approx 5 \text{ min} \quad (5)$$

Again, to know how much time it would take for the algorithm to process a graph with 30 vertices and 75% of edges we can calculate $f(30)$.

$$f(30) = 5.214 * 10^{-6} * e^{(0.8885*30)} = 1964673s \approx 22\ days \quad (6)$$

Above 20 vertices it quickly becomes unpractical to use this algorithm to solve the maximum weight clique problem as the time required to run the algorithm and get a solution grows exponentially – for 30 vertices it takes 22 days to get the solution! (6)

Although these calculations were only made for graphs with 75% of edges, the same conclusions could be made for graphs with a lower percentage of edges as the algorithm to find the solutions for those graphs is the same and therefore has the same time complexity.

Therefore, another approach, using a different algorithm, is needed to process graphs with a bigger amount of vertices and consequently a bigger amount of edges.

D. Optimizations to improve the algorithm's efficiency

As observed in the latter section, the algorithm is not capable of finding solutions in a useful time for graphs with a high amount of vertices (more than 20). This high amount of time needed to compute the solute on for the problem is due to the high temporal complexity of the algorithm (exponential complexity), and although we cannot change this complexity while keeping an Exhaustive Search implementation, there are some small improvements that we can do to slightly improve the time needed to get the solutions.

The first improvement is to filter the initial list of vertices to remove all the isolated vertices (vertices with degree 0) as these ones will not participate in the creation of the maximum weight clique once they don't have any neighbors. This will reduce the number of vertices, allowing us to do fewer combinations and therefore allowing the Exhaustive Search algorithm to calculate the solution in useful time for a higher amount of vertices.

The second improvement was made on the function to check whether a subset of vertices is a clique or not. In this function, as already explained, we get all the existent edges between the *n* vertices in the subset and count them. If the count is equal to the number of edges a clique with *n* vertices would have (1), then we have a clique. However, if our graph has fewer edges than the ones needed to create a clique for a certain subset, we can safely ignore that subset as we already know it will not form a clique.

Although these improvements could allow the algorithm to calculate results faster, they were disregarded for the results presented in the analysis as they would make the data more irregular and difficult for the analysis, having only theoric interest in this matter.

IV. GREEDY ALGORITHM FOR MAXIMUM WEIGHT CLIQUE

As previously mentioned, the Exhaustive Search Algorithm can not be used to solve the problem - get the maximum weight clique - for large graphs (graphs with a high number of vertices) due to its complexity – $O(2^n)$. For that reason, to find the maximum weight clique for graphs with a higher amount of vertices (higher than 20, at least) we need to find another algorithmic strategy to compute the solution. The chosen strategy was an algorithm based on a Greedy Heuristic.

The Greedy technique is widely used for optimization algorithms and suggests the construction of a solution step-by-step [3]. In each step, the solution is built using a greedy heuristic that will define the path we should follow to find the best solution, i.e., it will use the optimal local choice aiming to find the overall optimal way to solve the entire problem.

This solution may not be able to find the optimal solution for the overall problem as it makes decisions based only on the information it has in each one of the steps without regard to the overall problem. However, if on one side we may not find the optimal solution using this algorithm, on the other side, greedy algorithms are usually very efficient as their time complexity is usually low compared to other alternatives.

The implemented Greedy algorithm can be described briefly using the following sequence of steps:

1. Get the vertice with the highest weight and add to a list L;
2. Get all vertices that are neighbors of all vertices in L;
3. Get the neighbor with the highest weight;
4. Add the neighbor with the highest weight to L;
5. Get back to 2, until there are no neighbors for vertices in L;
6. The L list should contain the vertices of the maximum weight clique. Sum the weights of all vertices in L to get the weight of the maximum weight clique.

As stated before, this algorithm uses a heuristic that will only take into account the optimal local solution – it starts by finding the vertice with the highest weight, and then it selects the best neighbors (vertices with the highest weight) that can form a clique with the other ones in L. However, with this approach, we can easily miss the optimal solution if, for example, the maximum weight clique for a graph does not include the vertice with the highest weight, or, if the vertice with the highest weight

does not have any neighbors. In the next sections, as we compare this algorithm's results with the ones obtained by the Exhaustive Search algorithm (which always generated the optimal solution) this problem will be more evident.

A. Formal Analysis of the Algorithm's Complexity

This algorithm is based on an iterative approach, so in order to make a proper formal analysis of its complexity we need to start by defining its input. In this case, the input is the *v* vertices and *e* edges that are part of the graph for which we are trying to find the maximum weight clique. Next, we need to find the *basic operation,* which is the operation that consumes the most time in the algorithm [6]. For this algorithm, we will consider the *basic operation* as the operation for getting the vertices that are neighbors of all vertices in *L*. The complexity of the algorithm does not change with the order of the vertices or the order of edges, therefore, we are only interested in the *average case*, as studying the *best* and *worst cases* does not make sense in this case [6].

The *basic operation* is inside two "for" structures as seen in Snippet 1.

```
neighbors_of_all_vertices = []
for vertice in vertices_list:
  neighbors = []
  for edge in edges:
    operations_count += 1
    if edge.v1 == vertice:
      neighbors.append(edge.v2)
    if edge.v2 == vertice:
      neighbors.append(edge.v1)
  neighbors_of_all_vertices.append(neighbors)
```

Snippet 1: Get all vertices that are neighbors of vertices in the *L* list.

Thus, we can get a closed formula for the time complexity of this algorithm by doing:

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

The closed formula for the complexity of this algorithm in its *average case* is $v.e$, where *v* stands for each vertice in the graph and *e* for each edge. Consequently, we are facing an algorithm with **quadratic complexity** - $O(n^2)$. Once again, this complexity was expected as it is characteristic of algorithms in which the basic operation is inside two nested loops as referred to in [8].

B. Practical Analysis of the Algorithm's Complexity

The algorithm was tested using the same graphs generated for the Exhaustive Search. The number of vertices in these graphs varies from 5 to 200. For each number of vertices, there are 4 different versions of the graph. Each one of those versions has a different number of edges - 12.5%, 25%, 50%, and 75% of all possible edges totalizing 780 graphs in total.

The Greedy Algorithm was run over each one of the generated graphs, the results were stored and can be consulted in Table 2.

Vertices	Edges	Max Weight Clique	Operations Count	Tested Solutions	Search Time
5	12,50%	68	8	1	3,505E-05
5	25,00%	69	8	1	1,192E-05
5	50,00%	100	17	1	3,099E-05
5	75,00%	117	19	1	3,481E-05
6	12,50%	80	7	1	8,106E-06
6	25,00%	80	9	1	1,001E-05
6	50,00%	83	19	1	3,481E-05
6	75,00%	103	31	1	9,394E-05
7	12,50%	79	10	1	1,001E-05
7	25,00%	97	11	1	1,216E-05
7	50,00%	100	22	1	3,886E-05
7	75,00%	149	35	1	9,394E-05
8	12,50%	68	15	1	1,073E-05
8	25,00%	97	13	1	1,526E-05
8	50,00%	128	26	1	5,198E-05
8	75,00%	137	51	1	1,862E-04
9	12,50%	97	10	1	1,097E-05
9	25,00%	97	15	1	1,597E-05
9	50,00%	128	30	1	5,293E-05
9	75,00%	179	57	1	2,160E-04
[...]	[...]	[...]	[...]	[...]	[...]
191	12,50%	127	2280	1	4,584E-03
191	25,00%	156	4556	1	1,428E-02
191	50,00%	281	9128	1	8,351E-02
191	75,00%	560	13790	1	4,245E-01
192	12,50%	152	2312	1	6,630E-03
192	25,00%	177	4604	1	1,305E-02
192	50,00%	284	9210	1	6,309E-02
192	75,00%	570	13992	1	5,647E-01
193	12,50%	135	2328	1	4,776E-03
193	25,00%	137	4644	1	8,933E-03
193	50,00%	325	9336	1	1,080E-01
193	75,00%	566	14106	1	4,923E-01
194	12,50%	138	2352	1	4,471E-03
194	25,00%	175	4700	1	1,496E-02
194	50,00%	259	9416	1	8,638E-02
194	75,00%	586	14250	1	4,958E-01
195	12,50%	151	2384	1	7,730E-03
195	25,00%	180	4748	1	1,472E-02
195	50,00%	313	9513	1	8,528E-02
195	75,00%	497	14368	1	4,535E-01
196	12,50%	152	2408	1	6,761E-03
196	25,00%	200	4807	1	1,946E-02
196	50,00%	313	9611	1	9,733E-02
196	75,00%	571	14542	1	4,977E-01
197	12,50%	141	2433	1	7,101E-03
197	25,00%	180	4846	1	1,524E-02
197	50,00%	322	9725	1	1,110E-01
197	75,00%	612	14719	1	6,012E-01
198	12,50%	144	2449	1	4,458E-03
198	25,00%	155	4895	1	1,363E-02
198	50,00%	271	9823	1	1,102E-01
198	75,00%	481	14783	1	3,911E-01
199	12,50%	138	2474	1	4,887E-03
199	25,00%	186	4945	1	1,416E-02
199	50,00%	295	9906	1	8,820E-02
199	75,00%	492	14985	1	5,117E-01

Table 2: Experimental data for Greedy Algorithm.

With the Greedy algorithm, we were capable of finding solutions for graphs with 199 vertices and 75% of edges in useful time – approximately 0.5 seconds. Therefore, we can understand that this algorithm is way more efficient than the Exhaustive Search algorithm; a more detailed comparison between both algorithms is done in the next sections.

Using MatLab to analyze the results obtained, namely to find a function that fits the data, we were able to find a quadratic function (polynomial function with degree 2) that fits the *Operations Count* metric with an $r^2 = 0.999$. This confirms the results obtained in the formal analysis, i.e., the Greedy Algorithm has a **quadratic complexity** for its *average case*.

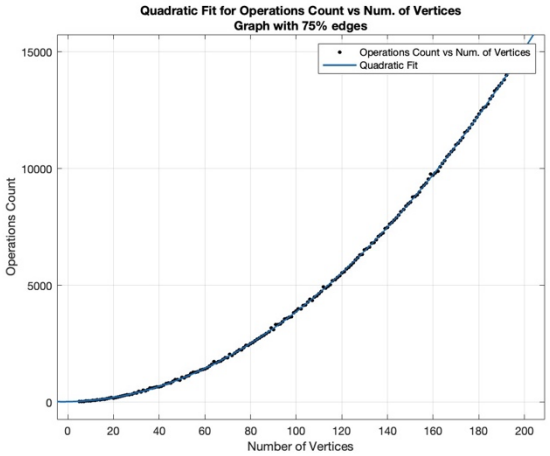

Fig. 4 - Quadratic Fit (f(x) = $0.37x^2 + 1.53x + 18.99$) for Operations Count in algorithm with Greedy Heuristic.

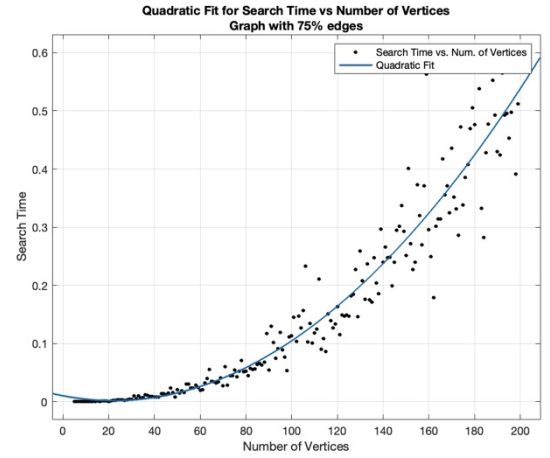

Fig. 5 - Quadratic Fit (f(x) = $1.696 * 10^{-5}x^2 - 0.000750x + 0.0102$) for Search Time in algorithm with Greedy Heuristic.

In Fig. 4 and Fig. 5 we can find graphical representations of the functions that best fit the data – *Operations Count* in Fig. 4 and *Search Time* in Fig. 5 for the Greedy algorithm.

C. Analysis of the algorithm's limits

Once we have the function that best fits our Search Time vs Number of Vertices correlation, then we can use it to predict how much time would take to find the maximum weight clique using the Greedy algorithm for an arbitrary number of vertices (with 75% edges). We can also use this function to know how far we could go, i.e., how big of a graph we could process/get the solution for in useful time.

To get the time, in seconds, the algorithm would take to get the maximum weight clique for a graph with 500 vertices and 75% of edges we can do:

$$f(500) = 1.696 * 10^{-5} * 500^2 - 0.000750 * 500 + 0.0102 = 3.875 \quad (8)$$

With 500 vertices, we can still use the algorithm to solve the problem in useful time as it would take only approximately 3.8 seconds to run (8).

$$f(5000) = 1.696 * 10^{-5} * 5000^2 - 0.000750 * 5000 + 0.0102 = 420.258 \quad (9)$$

With 10 times more vertices – 5000 vertices – it would take approximately 420 seconds ≈ 7 minutes (9), which is already a huge amount of time to compute a solution and therefore, cannot be accepted as something that could be done in useful time.

V. COMPARISON BETWEEN EXHAUSTIVE SEARCH AND GREEDY HEURISTIC

During the execution of this practical assignment, two different algorithms were implemented. Each algorithm has its own advantages and disadvantages. In this section, we will discuss the main differences between both algorithms and some use cases for each algorithm.

Some of the principal advantages of the Exhaustive Search algorithm are its simplicity – it is easy and very straightforward to implement once it uses combinations of all possible solutions to get the best solution overall – and its capability to always find the global optimal solution. These advantages contrast with some of the Greedy algorithm's disadvantages such as its inability to find the global optimal solution as it only takes into account the local optimal solution in each step [3].

However, the Exhaustive Search algorithm also has many disadvantages such as its huge temporal complexity – we have proven using formal and practical analysis that it has an exponential time complexity – and the high amount of memory used to store all the generated combinations for all graphs that contrast with the main advantage of the Greedy algorithm, i.e., its low temporal complexity – using the formal and practical analysis we defined its complexity as quadratic, which is way lower than the exponential complexity from Exhaustive search and allow us to find solutions (even if they are not optimal) for graphs with a high amount of vertices and edges.

Consequently, the Exhaustive Search algorithm is best suited for cases where we want to find the maximum weight clique for small graphs and we absolutely need to find the optimal solution while the Greedy algorithm is perfect for cases where we are dealing with huge and dense graphs and finding a good solution, even though not the best, may be acceptable. These conclusions also apply to other kinds of problems and data, i.e., the Exhaustive Search should be used when the problem is small but we need to find the optimal solution whereas the Greedy algorithm should be used when the problem is way bigger and an approximate solution can be acceptable.

VI. CONCLUSIONS

There are a huge amount of different strategies for optimization problems, each one with its own advantages and disadvantages. During this paper, we have explored two different approaches to get the maximum weight clique for a graph: a brute-force approach (Exhaustive Search), and an algorithm using a Greedy Heuristic – to get the best local vertice (the one with maximum weight) at each step and build a clique that was considered to be the global optimal solution or at least close to the global optimal solution.

Each approach was tested and compared with the other in terms of time complexity and results obtained using formal and practical methods. However, although we concluded the Greedy algorithm has lower complexity than the Exhaustive Search we cannot say it is the definitive solution for all use cases or that it can be a safe replacement for the brute-force algorithm as it may not always find the optimal solution [3].

Therefore, although the time complexity of an algorithm is a very important metric, it is not always sufficient to decide which algorithm should be used for a particular use case. In order to make this kind of decision, other metrics should be considered like the obtained results (if they are optimal or not) and the amount of memory used (although this metric is getting less important nowadays, it should still be considered) so we can find the algorithm that best adapts to our needs.

REFERENCES

[1] D. Ceccon, "Denny Ceccon," IA Expert Academy, 13-Jul-2020. [Online]. Available: https://iaexpert.academy/2020/07/13/algoritmos-de-otimizacao/. [Accessed: 02-Nov-2022].

[2] 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.

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

[4] "Clique (graph theory)," Wikipedia, 06-Jul-2022. [Online]. Available: https://en.wikipedia.org/wiki/Clique_(graph_theory). [Accessed: 02-Nov-2022].

[5] MikeFHay and mdp, "How many edges are in a clique of n vertices?," Mathematics Stack Exchange, 01-Dec-1959. [Online]. Available: https://math.stackexchange.com/questions/199695/how-many-edges-are-in-a-clique-of-n-vertices. [Accessed: 02-Nov-2022].

[6] 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.

[7] "Binomial theorem," Wikipedia, 09-Oct-2022. [Online]. Available: https://en.wikipedia.org/wiki/Binomial_theorem. [Accessed: 02-Nov-2022].

[8] 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.