Monte Carlo Algorithm For Maximum Weight Clique

Hugo Gonçalves 98497

***Abstract* –This paper presents a detailed analysis of a Randomized Algorithm - Monte Carlo Algorithm - 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 second practical assignment of Algoritmos Avançados developed by the author. This paper first presents an overview of the problem the algorithm intends to address. Afterward, the paper presents the time complexity analysis for the algorithm developed using formal methods and the practical results obtained. There is a discussion over the obtained results comparing the formal results with the practical results achieved by running the algorithm. Later, we will discuss possible optimizations to the algorithm so we could get better results with a higher probability. We will predict the limit of the algorithm and how long would it take to reach that limit. Finally, We will also calculate the algorithm's accuracy by comparing the obtained results with the ones achieved in the first assignment.**

***Resumo* –Este artigo apresenta uma análise detalhada de um algoritmo "Randomized- Algoritmo de Monte Carlo - para obter o clique de peso máximo num grafo não direcionado cujos vértices têm pesos positivos. Trata-se de uma compilação dos resultados obtidos para o segundo trabalho prático de Algoritmos Avançados. Este artigo começa por apresentar uma visão geral do problema que o algoritmo pretende abordar. Em seguida, o artigo apresenta a análise da complexidade temporal para o algoritmo desenvolvido por meio de métodos formais e resultados práticos obtidos. De seguida, há uma discussão sobre os resultados obtidos comparando os resultados formais com os resultados práticos alcançados através da execução do algoritmo. Mais adiante, discutiremos possíveis otimizações para o algoritmo de forma a que possamos obter melhores resultados com uma maior probabilidade e também calcularemos a precisão do algoritmo comparando os resultados obtidos com os alcançados no primeiro trabalho prático. Por fim, vamos prever o limite do algoritmo e quanto tempo levaria para que ele pudesse atingir esse limite.**

***Keywords* –Clique, Randomized Algorithm, Combinatory, Max Weight Clique**

***Palavras chave* –Clique, Algoritmo Random, Combinatória, Clique de Peso Máximo**

I. Introduction

This paper presents the implemented solution for the second assignment of "Algoritmos Avançados", in which it is asked to implement a randomized algorithm to get the Maximum Weight Clique for a certain graph G. To resolve the assignment, an implementation of the Monte Carlo algorithm was developed.

In the next sections, we start by introducing the notion of Randomized Algorithms and discuss the main differences between these types of algorithms and the ones used in the first assignment. Next, we write an overview of the problem the algorithm intends to address. Afterward, there is a discussion about the time complexity of the developed algorithm based on the formal methods and the practical results obtained, comparing both of them to make sure they are consistent. Later, we will examine possible optimizations that were implemented, in the context of this problem, to the Monte Carlo algorithm so we could get better results with a higher probability. We will also calculate the algorithm's accuracy by comparing the obtained results with the ones achieved in the first assignment. Finally, we will predict the limit of the algorithm, i.e., for how big of a graph it could get a solution in util time by analyzing a fit for the results obtained, which will help us to predict the performance of the algorithm for bigger graphs.

II. Randomized Algorithms

A randomized algorithm is intended to find a solution for a certain problem using a certain degree of randomness as part of its logic. The difference between these algorithms and the brute force algorithms, like the ones implemented in the first assignment, is that for brute force algorithms the answer is always deterministic, and, with the correct algorithm, we can find the optimal solution, while for the randomized algorithms, the solution is not deterministic, i.e., different executions of the algorithm will, most likely, provide different results.

There are two major categories of randomized algorithms: Monte Carlo and Las Vegas. Monte Carlo algorithms are always fast, but they can only give the correct answer with a given probability. Las Vegas algorithms, however, will always give the correct answer, but they are not guaranteed to always be fast.

During the practical assignment on which this paper is based, as already declared before, the author implemented a Monte Carlo algorithm to solve the problem and get the maximum weight clique for a given graph. The Monte Carlo algorithm is guaranteed to be fast,

but it may or may not find the correct solution. However, it is assured that it will always find the correct answer with a certain probability. In some cases, the probability of error can be so low, even within efficient execution time, that it can be considered as negligible for practical purposes. Therefore, the execution time for these algorithms is deterministic, i.e., is always predictable, due to the stop conditions implemented, but the result may be wrong with a small probabilistic error.

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

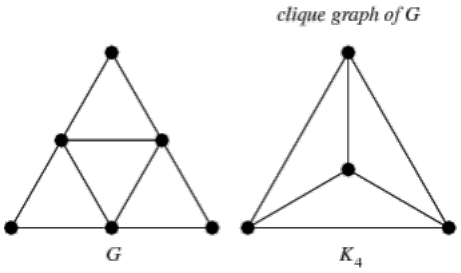

Fig. 1 - Example of a Clique K4 inside a Graph G.

Another fundamental 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]:

$$\binom{n}{2} = \frac{n!}{2! * (n-2)!} = \frac{1}{2} * n\,(n-1) \qquad (1)$$

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.

IV. Monte Carlo Algorithm For Maximum Weight Clique

For this assignment, a Monte Carlo algorithm was implemented to retrieve the problem solution, i.e., the maximum weight clique for a set of vertices with a certain edge probability. The graphs used were the ones already generated and used for the first assignment. This helps to calculate in a more reliable way the accuracy of the algorithm by comparing its results with the results obtained in the previous assignment.

A. Algorithm Strategy

The first step is to generate random vertices subsets with size n for each graph G with v vertices. The size (n) of the subsets will be a value in the interval 2...n, where n is at maximum equal to the total number of vertices in the graph.

Next, each one of the generated subsets is tested - to check if it is a clique or not - and added to a Python *set*, so we don't test the same subset more than once. A *set* structure was used instead of a *list* because the average complexity of the Python *in* operation is *O(1)* for a set while it is *O(n)* in a list. This means that by using a set instead of a list we can improve the performance of the operation to check if a subset was already tested or not.

```
# Convert to frozenset so it can be hashable
frozen_subset = frozenset(subset)

# If we already calculated this subset continue
if frozen_subset in tested_subsets:
    continue

# Mark the current subset as tested
tested_subsets.add(frozen_subset)
```

We keep generating subsets until one of the conditions is met:

- All subsets of size n were generated and tested;
- We reached a stop condition defined by the Monte Carlo algorithm.

When one of the above conditions is reached, we will give up on trying more solutions for subsets with size n and will start generating subsets with size *n+1* until n is equal to the number of vertices of the graph. When n is equal to the number of graph G's vertices we will assume that the current solution is the best solution overall and return it.

To check if all subsets of size n were generated we can compare the number of subsets in *tested subsets* with the maximum number of combinations of subsets with size n for a graph with v vertices. In order to know this last parameter, i.e., the maximum number of subsets with size n we can apply the mathematical formula of the Combinations, given by:

$$\binom{v}{n} \qquad (2)$$

By calculating this, we can know when it is no longer possible to keep generating subsets for a certain size n and when we should start generating new subsets with size *n+1*

The algorithm can be described, in a very simplified way, by the following steps:

```
# vis = vertices in subset count
# g.v = vertices in graph g
for vis in range(2, len(g.v)):

  tested = set()
  max_clique = None
  max_combs = get_max_number_of_subsets()

  while len(g.v) < max_combs and not
    stop_conditions_reached():

      rnd_subset = generate_rnd_subset(vis)
      if rnd_subset not in tested:
        tested.add(rnd_subset)
      if is_clique(rnd_subset):
        max_clique = Clique(rnd_subset)
```

The *get_max_number_of_subsets()* function is calculated using Combinations (2).

The *generate_rnd_subset(vis)* function just creates a random subset by picking one by one the vertices in the graph. The latter function is illustrated by the following code:

```
subset = set()
while len(subset) < vi:
  rnd_idx = random.randint(0, len(g.v)-1)
  rnd_v = g.v[rnd_idx]
  subset.add(rnd_v)
```

As you may notice, picking the vertices randomly, i.e., picking each one of the graph's vertices with the same probability, may not be the best or most efficient approach to this particular case. Therefore, some optimizations to make this process more efficient will be discussed in the next sections.

B. Stop Conditions

As already expressed in the previous sections, the Monte Carlo algorithm requires some stop conditions, i.e., conditions that, when met, will make the algorithm stop testing new solutions with the current conditions and either change the conditions or consider the current best solution as the global solution - this was explained in the latter section.

In the developed implementation, this is no different. Three different stop conditions were implemented and every time one of them is reached, the algorithm stops testing subsets with size n and starts testing subsets with size $n+1$.

- **Time Threshold**: 3 seconds.
- **Operations Threshold**: 500 000 operations.
- **Tested Solutions**: 75 000 solutions.

However, fixing static values for the stop conditions comes with a problem. If we set very high values for the stop conditions, for smaller problems we will be doing nothing more than an Exhaustive Search, as the stop conditions will hardly be reached. On the other side, if we set small values for the conditions that would be great for the smaller problems, for larger problems - larger graphs in our case -, that would naturally require more time to process, the found solutions will be significantly inaccurate.

To mitigate this problem some optimizations were implemented. These optimizations will be discussed in the next chapter.

C. Optimizations

As discussed in the previous sections, some optimizations were made to make this algorithm a bit more "smart" and efficient. When presenting the results we'll actually see if the optimizations were successful or not but for now, we'll focus on explaining the implemented optimizations and how they can improve our Monte Carlo algorithm.

C.1 Algorithm Strategy Optimizations

As discussed, generating subsets by picking each one of the vertices available with the same probability may not be the best strategy. As already glimpsed in the previous assignment, vertices with higher weight also have a higher probability of making part of the Maximum Weight Clique. This means that if we could attribute a higher probability of picking vertices with higher weights while generating each one of the subsets, that would probably increase the efficiency of our algorithm. By doing this, our first subsets - the ones generated before we reach a stop condition - would have vertices with higher weights, and therefore a higher probability of generating the Maximum Weight Clique. That is exactly what we have done!

In this optimization, our function *generate_rnd_subset(vis)* was replaced by (pseudo-code, not working):

```
subset = set()
sorted_vs = sort(g.v)
vs_count = len(sorted_vs)

while len(subset) < vi:
  rnd_p = = random.random()
  if rnd_p < 0.80:
    rnd_idx = rndint(0, floor(vs_count*0.6))
  else:
    rnd_idx = rndint(floor(vs_count*0.6),
                vs_count-1)

  rnd_v = g.v[rnd_idx]
  subset.add(rnd_v)
```

We are still creating a new subset with random vertices, however, 80% of the time we will use one of the vertices that are on top 60% of the vertices with higher weight. The other 40% vertices - the ones with lower weight - will only be used 20% of the time, which should be enough to not lock our algorithm into only using the heaviest vertices, as sometimes the Max.

Clique may contain some of the lighter vertices. This will, in theory, help to make sure that the subsets created will be the ones with a higher probability to generate the Maximum Weight Clique.

Besides this optimization, another one was implemented. This other optimization limits the number of maximum subsets that can be generated for each value of n (n being the number of vertices in each subset). As discussed, the amount of subsets generated is limited by the stop conditions or by the maximum amount of combinations that can be generated. With this optimization, we will never generate all the possible combinations of subsets, but a number of subsets that is determined by the function f (3) and that varies with the number of items in the subset n.

$$f(x) = \frac{-10 + \frac{50}{1.6 * \log n}}{100} \tag{3}$$

The above function is a logarithmic function that provides a higher value for a smaller value of n and a smaller value for a higher value of n. The value provided by the function is multiplied by the maximum combinations possible - calculated using formula (2) - which ensures that the higher the number of vertices in the subset, the lower the number of subsets generated.

This optimization takes into consideration that is easier to generate a clique with a smaller amount of vertices than with a higher amount of vertices. Therefore, we generate fewer subsets when n is higher allowing, in theory, to discard subsets that will not, most likely, result in a clique.

C.2 Stop Conditions Optimizations

As approached in section IV-B, setting static values for the stop conditions doesn't allow our algorithm to adapt and work efficiently while providing the best solutions possible for all graphs (from graphs with 5 vertices up to graphs with 200 vertices).

If instead of using static values for the stop conditions we used adaptive values, that would grow as the size of the problem grows, we could get the best of the two worlds, i.e., we can have smaller stop conditions for smaller problems and higher values for bigger problems (larger graphs). This allows us, in theory, to control the time and operations we spend on each problem while giving the algorithm more time and freedom to search for more solutions for larger problems.

This *adaptive* mechanism was implemented using three different quadratic functions. A different function was created for each stop condition to have better control over its value.

$$f(x) = 150x^2 + 100000 \tag{4}$$

$$g(x) = \frac{1}{1700}x^2 + 0.8 \tag{5}$$

$$h(x) = 80x^2 + 75000 \tag{6}$$

The function used for each one of the stop conditions was:

- **Operations Threshold**: Function $f(x)$ - (4).
- **Time Threshold**: Function $g(x)$ - (5) .
- **Tested Solutions**: Function $h(x)$ - (6).

These functions will determine a value for the stop conditions in function of x - the number of vertices the graph contains.

V. Monte Carlo Algorithm Results

To test the algorithm in practice we used the graphs generated for the previous assignment. These graphs have between 5 and 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 were, respectively, 12.5% 25%, 50%, and 75% of the maximum number of edges possible for the number of vertices.

The implemented Monte Carlo algorithm was run over each one of those successively larger graphs with and without the described optimizations.

A. Without Optimizations

In Table I we can check the results obtained for the Monte Carlo algorithm with no optimizations. In order to fit the table in this paper, only the results for 25% and 75% of the edges are presented. The remaining results can be consulted in the Appendix document.

Vertices	Edges	Solution	Ops.	Tested	Time
5	25,00%	69	16	26	0,0002
5	75,00%	117	56	26	0,0003
6	25,00%	80	48	57	0,0005
6	75,00%	103	176	57	0,0007
7	25,00%	97	160	120	0,0012
7	75,00%	149	480	120	0,0023
8	25,00%	97	448	247	0,0056
8	75,00%	137	1344	247	0,0056
9	25,00%	100	1152	502	0,0106
9	75,00%	179	3456	502	0,0146
10	25,00%	80	2816	1013	0,0266
10	75,00%	180	8448	1013	0,0361
11	25,00%	87	6656	2036	0,0635
11	75,00%	210	20992	2036	0,1095
12	25,00%	100	16384	4083	0,1447
12	75,00%	222	50176	4083	0,2634
13	25,00%	96	38912	8178	0,3860
13	75,00%	166	118784	8178	0,6647
14	25,00%	100	90112	16369	0,9724
14	75,00%	194	278528	16369	1,6558
15	25,00%	109	212992	32752	2,1722
15	75,00%	242	453308	27807	3,0004
16	25,00%	140	295696	50457	3,0009
16	75,00%	208	477487	34175	3,0009
17	25,00%	136	322310	63228	3,0028
17	75,00%	253	485623	40018	3,0013
18	25,00%	140	261084	62882	3,0030
18	75,00%	228	398875	38314	3,0005
19	25,00%	136	249517	65932	3,0020
19	75,00%	229	402435	42982	3,0062
20	25,00%	128	190038	60381	3,0129
20	75,00%	250	365078	41759	3,0038
...	...	...	...	...	...

Vertices	Edges	Solution	Ops.	Tested	Time
...	...	...	...	...	...
195	25,00%	97	1362	5362	3,0018
195	75,00%	98	1327	1786	3,0029
196	25,00%	97	1277	5267	3,0015
196	75,00%	97	1323	1787	3,0036
197	25,00%	96	1286	5206	3,0020
197	75,00%	97	1308	1744	3,0036
198	25,00%	97	1318	5204	3,0012
198	75,00%	98	1303	1739	3,0027
199	25,00%	97	710	2942	3,0029
199	75,00%	96	769	976	3,0054

TABLE I
SAMPLE OF THE RESULTS OBTAINED FOR MONTE CARLO ALGORITHM WITH NO OPTIMIZATIONS

B. With Optimizations

In Table II we can check the results obtained for the Monte Carlo algorithm with the described Optimizations in the previous sections. In order to fit the table in this paper, only the results for 25% and 75% of the edges are presented. The remaining results can be consulted in the Appendix document.

Vertices	Edges	Solution	Ops.	Tested	Time
5	25,00%	69	13	20	0,0002
5	75,00%	117	39	20	0,0003
6	25,00%	80	27	38	0,0006
6	75,00%	100	102	38	0,0009
7	25,00%	97	81	67	0,0007
7	75,00%	148	227	67	0,0012
8	25,00%	97	168	122	0,0017
8	75,00%	133	511	122	0,0026
9	25,00%	100	468	218	0,0026
9	75,00%	179	1227	218	0,0048
10	25,00%	80	970	400	0,0051
10	75,00%	180	2735	400	0,0099
11	25,00%	87	2220	732	0,0116
11	75,00%	210	6464	732	0,0242
12	25,00%	100	4854	1353	0,0258
12	75,00%	222	14424	1353	0,0535
13	25,00%	96	10408	2522	0,0556
13	75,00%	164	31818	2522	0,1336
14	25,00%	100	23484	4739	0,1131
14	75,00%	194	70295	4739	0,2958
15	25,00%	109	54505	8961	0,2347
15	75,00%	242	133786	8420	0,5723
16	25,00%	140	119358	17026	0,5558
16	75,00%	208	138407	10730	0,6349
17	25,00%	136	143357	24488	0,7364
17	75,00%	253	143364	12506	0,7031
18	25,00%	136	148604	29618	0,8327
18	75,00%	229	148612	14133	0,7742
19	25,00%	136	154153	33182	0,9123
19	75,00%	232	154162	16243	0,9006
20	25,00%	128	160001	39438	1,0362
20	75,00%	250	160002	17636	0,9737
...	...	...	...	...	...
195	25,00%	143	18422	35993	23,1706
195	75,00%	98	10896	14494	23,1702
196	25,00%	140	17824	35356	23,4009
196	75,00%	98	10937	14597	23,4011
197	25,00%	133	18852	36826	23,6318
197	75,00%	98	11025	14712	23,6306
198	25,00%	142	18606	36854	23,8646
198	75,00%	98	11333	15060	23,8627
...	...	...	...	...	...

Vertices	Edges	Solution	Ops.	Tested	Time
...	...	...	...	...	...
199	25,00%	134	18675	36724	24,0977
199	75,00%	98	11286	15060	24,0976

TABLE II
SAMPLE OF THE RESULTS OBTAINED FOR MONTE CARLO ALGORITHM WITH OPTIMIZATIONS

C. With Optimizations vs Without Optimizations

As discussed in the section Optimizations IV-C, the optimizations developed by the author for the implemented Monte Carlo algorithm should, in theory, improve its efficiency and the results obtained. In this section, we will compare the results of the optimized version with the results of the not optimized to actually check if the optimizations were successful or not. All the comparisons and chart analysis from now on will be made for graphs with 25% of edges.

In Fig. 2, we've used *Matlab* to draw the solutions found in the function of vertices in the graph for both algorithm versions - the Optimized and the Not Optimized version.

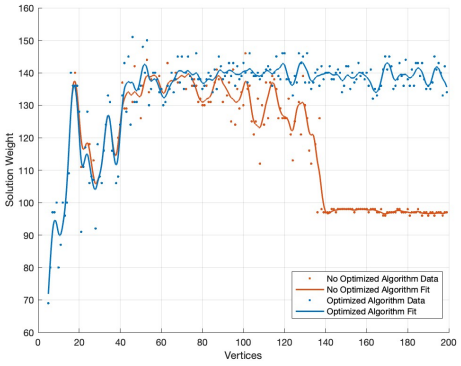

Fig. 2 - Comparison between Optimized (in blue) and Not Optimized (in orange) relatively to Solutions found.

As we can see, for smaller graphs (up to 40 vertices), both algorithms are more or less equivalent, i.e., both of them can find almost the same solutions. However, for larger graphs (mostly from 70 vertices on-wards) the Optimized version of the algorithm can almost always find better solutions than the Not Optimized version, and for very large graphs (more than 130 vertices) the Optimized version may be the one with acceptable results, due to the large difference in the values found for the solutions. This behavior can be explained due to the optimizations performed in the Stop Conditions - the stop conditions are dynamic and adapt to the size of the problem (see more details in section IV-C.2).

Nevertheless, by looking at Fig. 2 we are persuaded into thinking that for smaller problems both algorithms are equivalent, once both can find the same solutions. This is not true. For smaller graphs, the Optimized version of the algorithm is preferable to the Not Optimized if we need to save time, while the Not Optimized is better if we need more precise results.

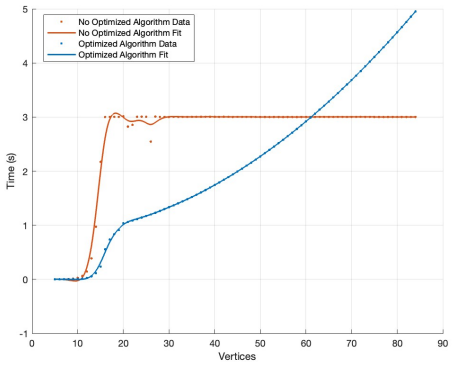

Fig. 3 - Comparison between Optimized (in blue) and Not Optimized (in orange) algorithms relative to Search Time.

As we can see in Fig. 3, for smaller to medium graphs (5 up to 60 vertices), the Optimized version of the algorithm outperforms the Classic or Not Optimized version in time matters. The conclusion is simple, even for smaller graphs, if we need to save time, it is better to use the Optimized version once it will find the same solutions as the Not Optimized version (with some small variations) but faster!

This behavior in the Optimized version - being able to find solutions faster for small graphs - may be explained by the optimization performed to reduce the maximum amount of subsets calculated and by the optimization performed in the algorithm to get samples of vertices - in the optimized version heavier vertices have a higher probability of being chosen. More details about these optimizations were already provided in the previous section IV-C.1.

VI. Monte Carlo Algorithm Complexity

A. Formal Analysis

Once this implementation is iterative, to make a proper formal analysis of its complexity we must start by defining the size of the input [2]. 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 [2]. In this case, the basic operation would be the test to check if a generated subset is a clique.

In this problem, it is not possible to check if we reached the global solution in runtime. This means that we cannot interrupt the algorithm based on the local solution we have. Therefore, the complexity of the algorithm depends on the values of the stop conditions and on whether they were reached or not.

If the stop conditions are too high, then we are beyond the worst case as we will generate all (or almost all) the possible subsets for the graph. In this case, the worst-case complexity will be of $O(2^n)$ accordingly to the Binomial Theorem [3] which states that the total number of combinations from a set with v elements is $2^v - 1$.

However, if the stop conditions are low, then we will be against the best case as we will generate a constant number of solutions and perform a constant number of operations. Therefore, the best-case complexity is a constant complexity - $\Omega(1)$.

Again, the complexity for the average case will depend on the stop conditions we implement and therefore, is different for the two different versions of the algorithm implemented.

For the Classic (Not Optimized) version, we will have exponential complexity until we reach the stop conditions. From that point onwards, we should have a constant complexity as the time and operations performed will be constant (equal to the ones defined by the stop conditions). Therefore, we can affirm that the complexity of this algorithm for the average case can be defined by $O(2^n)$.

For the Optimized version of the Monte Carlo algorithm, we are using adaptive Stop Conditions that will increase its value with the size of the problem. The functions (4), (5), and (6) control how much the value of the Stop Conditions is increased. Once these functions are quadratic functions, then we are expecting this algorithm to have a quadratic complexity - $O(n^2)$.

B. Practical Analysis

The complexity of the algorithm depends on the version implemented. The algorithm with No Optimizations has a complexity, and the Optimized algorithm has a different complexity.

B.1 Not Optimized Monte Carlo algorithm

For the Classic (Not Optimized) algorithm, in the Formal Analysis, we've come to the conclusion that its complexity is exponential until one of the stop conditions is reached. With the practical results, we can confirm or refute this theory. We know from Fig. 3, that the algorithm first reaches the time stop condition, approximately, at the graph with 16 vertices.

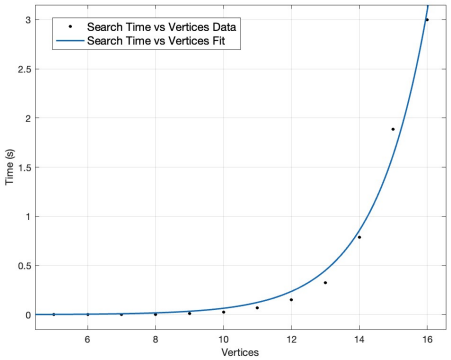

Fig. 4 - Quadratic Fit for Search time in function of Vertices - Not Optimized Algorithm.

Using Matlab we created an exponential fit for the function that gives the time spent in the search of the Max Weight Clique in the function of the number of vertices in the graph. The results were the following:

```
General model Power:
     f(x) = a*x^b
Coefficients (with 95% confidence bounds):
       a =     1.3e-11 (-2.65e-11,5.251e-11)
       b =       9.446 (8.34, 10.55)

Goodness of fit:
  SSE: 0.07287
  R-square: 0.9927
  Adjusted R-square: 0.992
  RMSE: 0.08536
```

We were able to obtain a fit with $r^2 = 0.992$. Consequently, we can confirm that the complexity of the Not Optimized Monte Carlo algorithm up to 16 vertices is exponential - $O(2^n)$. From 16 vertices onwards, the complexity is $O(1)$ - constant because all the subsequent solutions will be limited in the amount of time they can spend and the operations they do by the stop conditions imposed. This linear behavior can be witnessed in Fig. 3.

B.2 Optimized Monte Carlo algorithm

As seen in the Formal Analysis section, we are expecting to have a quadratic complexity - $O(n^2)$ - for the Optimized version of the algorithm, based on the stop conditions defined.

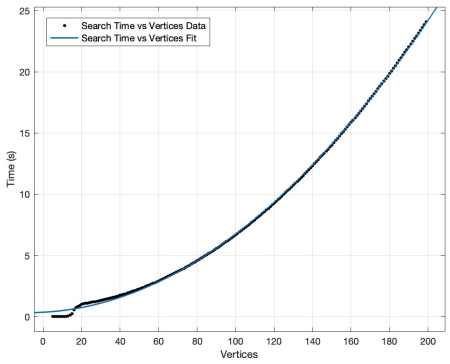

Fig. 5 - Quadratic Fit for Search time in function of Vertices - Optimized Algorithm.

Using Matlab we created a quadratic fit for the function that gives the time spent in the search in the function of the number of vertices in the graph. The results were the following:

```
Linear model Poly2:
     f(x) = p1*x^2 + p2*x + p3
Coefficients (with 95% confidence bounds):
       p1 =   0.0005551 (0.0005477,0.0005625)
       p2 =    0.008245 (0.006699,0.00979)
       p3 =        0.36 (0.2915,0.4285)

Goodness of fit:
  SSE: 4.183
  R-square: 0.9996
  Adjusted R-square: 0.9996
  RMSE: 0.1476
```

We were able to obtain a fit with $r^2 = 0.9996$. Consequently, we can confirm that the complexity of the Optimized Monte Carlo algorithm is, as expected, quadratic - $O(2^n)$. It is important to notice that if we wanted to reduce the complexity of the algorithm we could do so by just changing the functions that determine the adaptive Stop Conditions making them logarithmic, for example.

VII. Monte Carlo Algorithm Limits

A. Not Optimized Version

For the not Optimized Version of the algorithm, we limited the Search for solutions to a maximum of 3 seconds. Therefore, we can use the algorithm to solve problems as big as we want and it will only take, at most, 3 seconds to find a solution to those problems.

Obviously, as we increase the size of the problem, we will also be decreasing the probability of finding the optimal solution for the problem. This means that although we could, in theory, find solutions for huge problems (graphs with a tremendous amount of vertices), beyond 140 vertices the solutions start getting very poor, i.e., exceedingly far from the optimal solution - as we can see in Fig. 2.

B. Optimized Version

For the Optimized version of the algorithm, its complexity is limited by the adaptive Stop Conditions defined. As discussed in the previous sections, the adaptive Stop Conditions were defined with a resort to quadratic functions, namely the functions (4), (5), and (6). If we consider the function that defines the limit of time the algorithm can spend finding solutions (5), then we can use it to predict how much time the algorithm would take to find solutions for a graph with v vertices.

For 200 vertices it would take approximately 24 seconds as we can see in Table II.

$$g(500) = \frac{1}{1700}500^2 + 0.8 = 147.86 \tag{7}$$

For 500 vertices, using the formula (7) we can predict that it would take around 147 seconds to search for solutions, which is already more than 2 minutes.

$$g(1000) = \frac{1}{1700}1000^2 + 0.8 = 589.04 \tag{8}$$

For 1000 vertices, using formula (8) we estimate that it would require 589 seconds to return a solution to the problem - almost 10 minutes!

Therefore, we can affirm that beyond 1000 vertices is impractical to use this algorithm as it is. It is worth noting that the time the algorithm takes is directly correlated with its complexity which, as already discussed before, depends on the functions used for the Stop Conditions. If we want we can replace these functions with logarithmic functions making the algorithm faster but sacrificing precision in the solutions obtained.

VIII. Monte Carlo Algorithm Accuracy

In order to check our Monte Carlo algorithms' accuracy we compared the results obtained with the ones acquired from the previous assignment. We started by comparing with the Exhaustive Search results that, as we know, always provide the optimal solution.

Exhaustive S.	M. C. Optimized	M. C. Not Optimized
68	68	68
69	69	69
100	100	100
117	117	117
80	80	80
80	80	80
83	83	83
103	100	103
79	79	79
97	97	97
100	100	100
149	148	149
68	68	68
97	97	97
128	128	128
137	133	137
97	97	97
100	100	100
148	148	148
179	179	179
73	73	73
80	80	80
180	180	180
87	87	87
87	87	87
140	140	140
210	210	210
88	88	88
100	100	100
136	136	136
79	79	79
96	96	96
127	127	127
88	88	88
100	100	100
146	146	146
194	194	194
96	96	96
109	109	109
242	242	242
96	96	96
140	140	140
208	208	208
136	129	136
136	136	136
253	253	253
140	140	140
140	136	140
173	173	173
231	229	228
136	136	136
136	136	136
198	194	198
249	232	229
128	128	128
128	128	128
Total	Misses	Misses
55	8	2

TABLE III
Comparison between all solutions obtained for Exhaustive Search vs Monte Carlo Optimized vs Monte Carlo Not Optimized.

As we can observe in Table III, the different Monte Carlo versions (Not Optimized and Optimized) have different accuracy results. The Not Optimized version finds the same solutions as the Exhaustive Search 53 out of 55 times, i.e., has an accuracy of approximately 96% while the Optimized version finds the optimal solution - the best solution possible - 47 out of 55 times, and therefore has, in this set of solutions, has an accuracy of 85%.

With such a small set of solutions, it is hard to correctly identify the accuracy of the developed algorithms, however, we were only able to obtain optimal solutions for up to 20 vertices in the previous assignment, and, therefore, this comparison is the only we can do.

IX. Conclusion

By implementing the Monte Carlo algorithm we were able to better understand how this algorithm behaves and which are the general benefits and disadvantages of randomized algorithms. Randomized algorithms are a great choice when we need to obtain an approximate solution in the less possible amount of time. However, they may miss the optimal solution with a certain probability, so, if we really need the optimal solution for the problem they may not be the best possible choice. It is also important to note that the Monte Carlo algorithm is not a deterministic algorithm, therefore, different successive executions can provide different results [1].

We were also able to perform some probabilistic optimizations that improved the efficiency of the algorithm allowing it to use less time and find better solutions for large problems. However, these optimizations proved to decrease the accuracy of the algorithm for small problems. This could be mitigated by adjusting the Stop Conditions functions in the optimized version.

Finally, we were able to compare the accuracy of the Monte Carlo algorithm against the Exhaustive Search algorithm concluding that, at least for small problems, the Monte Carlo algorithm can be a great alternative to the Exhaustive Search as it can find the optimal solutions in less time.

References

[1] D. Vrajitoru and W. Knight, "Algorithms and Probabilities," in Practical Analysis of Algorithms, Cham: Springer, 2014, pp. 325–328.

[2] 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.

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