Max Cut Problem - 2nd delivery

Rafael Gil

Abstract –This report presents a continuation of the study of algorithms aimed at solving the maximum cut problem. Previously, a study was conducted on a two algorithms: an exhaustive algorithm and a randomized greedy algorithm, were a formal and experimental analysis was done for both of them. The intention of this report was to conduct the same study but for a randomized algorithm, but seeing as the first report already includes a probabilistic algorithm, this report is going to be conducted on a purely greedy algorithm.

Resumo –Este relatório apresenta uma continuação do estudo de algoritmos destinados a resolver o problema do corte máximo. Anteriormente, foi efectuado um estudo sobre dois algoritmos: um algoritmo exaustivo e um algoritmo voraz aleatório, tendo sido feita uma análise formal e experimental para ambos. A intenção deste relatório era efetuar o mesmo estudo mas para um algoritmo aleatório, mas visto que o primeiro relatório já inclui um algoritmo probabilístico, este relatório vai ser realizado sobre um algoritmo puramente voraz.

Keywords –Max-Cut, Cut, Algorithm, Time Complexity, Greedy Heuristics

Palavras chave –Max-Cut, Corte, Algoritmo, Complexidade Temporal, Heuristicas Vorazes

I. Introduction

A. Problem Definition

Find a maximum cut for a given undirected graph $G(V, E)$, with n vertices and m edges. A maximum cut of G is a partition of the graph's vertices into two complementary sets S and T, such that the number of edges between the set S and the set T is as large as possible.
This is a NP-hard problem, meaning that there is no known solution to solve it in polynomial time. The best approximation, known to date, for the maximum cut problem, was proposed in 1994, by Goemans and Williamson, where a semidefinite programming relaxation was used to obtain a consistent approximation of .87856 times of the optimal solution.

B. Greedy Heuristic

This is an algorithmic approach that searches to select the optimal solution of an iteration of a problem, without worrying about the overall optimal solution of the given problem.
This technique might not always provide the best result for every problem, but it, usually, provides a good enough approximation to the optimal solution, in return of being much less time consuming than other algorithmic approaches.

II. Analysis of the Algorithms Implemented

A. Greedy Heuristic

There exist many different greedy heuristics that can be employed to get an approximate solution of the maximum cut problem.
The algorithm being analysed here, consists of starting with an arbitrary cut $C = (S, T)$, and proceed to swap vertices v between the sets, in order to maximize the cut C [1].
The arbitrary cut chosen is one where half the vertices are in set S and the other half are in set T. The process of creating the initial cut, seen in algorithm 1, consists on iterating through every vertex in the graph and attributing half of them to set S and the other half to set T. This process runs in linear time, $O(n)$, where n is the number of vertices in the graph.

Algorithm 1 Initialize Cut

```
1: function InitializeCut(vertices)
2:     partition ← {}
3:     for node in range(vertices) do
4:         partition[node] ← "S" if node < vertices/2
   else "T"
5:     end for
6:     return partition
7: end function
```

The next process in this algorithm, described in algorithm 2, is the calculation of the cut' size of the random partition that has been generated, which consists in iterating through every edge in the graph and checking, in the random partition, if the vertices of the edge are in different sets. This runs in linear time, $O(e)$, where e is the number of edges in the graph. This is the same process used in the randomized algorithm, detailed in the first report.

Algorithm 2 Calculate Cut Size

```
function Calculate_Cut_Size(graph, partition)
    cut_size ← 0
    for edge in graph.edges() do
        if partition[edge[0]] ≠ partition[edge[1]] then ▷ if the two vertices of the edge are in different sets
            cut_size ← cut_size + 1
        end if
    end for
    return cut_size
end function
```

Another process of this algorithm, seen in algorithm 3, consists in moving a given vertex v, from one set to the other. So, if $v \in S$, it will be moved to set T and vice versa. This is done to try to increment the number of edges that cross the two sets. This process runs in constant time, $O(1)$.

Algorithm 3 MoveVertex

```
function MoveVertex(vertex, partition)
    partition[vertex] ← if partition[vertex] == "S" then "T" else "S"
end function
```

Lastly, the main execution process, seen in algorithm 4, consists on iterating through every vertex and performing the actions described previously. This way, it is possible to obtain a good enough approximation to real solution. This makes it so that the overall algorithm runs in polynomial time, $O(n * m)$, where n corresponds to the number of vertices and m to the number of edges.

Algorithm 4 OptimizeCut

```
made_move ← True
while made_move do
    made_move ← False
    initial_cut_size, _ ← CalculateCutSize(graph, partition)
    for v in range(vertices) do
        MoveVertex(v, partition)
        new_cut_size, operations ← CalculateCutSize(graph, partition)
        number_operations += operations
        if new_cut_size > initial_cut_size then
            made_move ← True
            current_cut ← new_cut_size
            current_partition ← partition
            if current_cut > best_cut then
                best_cut ← current_cut
                best_partition ← current_partition
            end if
        else
            MoveVertex(v, partition)
        end if
    end for
end while
```

III. Results

A. Impact of the vertices on the algorithms

This experimental analysis was conducted with the same graphs used in the first report, in order to test this algorithm under the same circumstances that the other two algorithms were tested. The graphs used had a successively larger number of vertices, starting as low as 4 vertices and ending at a highest of 25 vertices, and a constant probability of 30% for edge creation.
After running the experiments, a comparison between the results obtained in the first report and the results obtained for the algorithm being analysed here was made. We can see, in Fig. 1 and Fig. 2, that both the greedy algorithm and the randomized algorithm have the same rate of growth, which was predictable as they both have a polynomial time complexity.

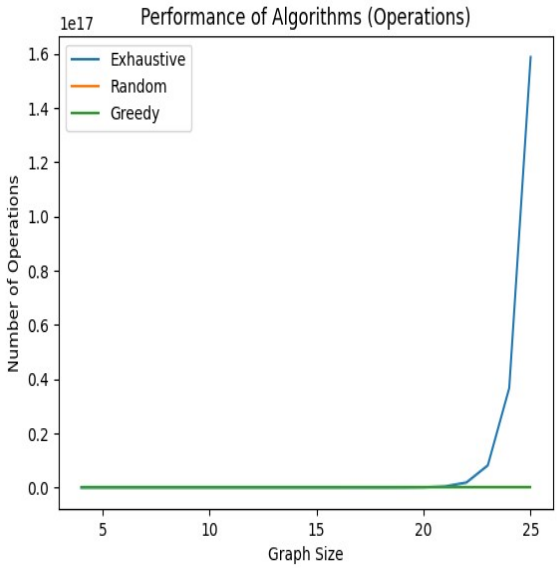

Fig. 1 - Impact of number of vertices in operations per Algorithm

N. Vertex	Exhaustive Time(s)	Random time(s)
4	0.0001389	0.0045344
5	0.0002294	0.0081493
6	0.0002922	0.0075021
7	0.00068979	0.009448
8	0.0015438	0.0097511
9	0.0049507	0.0120359
10	0.0095575	0.0109243
11	0.0229472	0.0134678
12	0.0453573	0.0148012
13	0.1089745	0.0153767
14	0.2192243	0.0192595
15	0.7063999	0.0276222
16	1.1780833	0.0295291
17	2.5152729	0.0236615
18	6.2236802	0.0450665
19	12.5995052	0.0322919
20	26.0108515	0.0360631
21	56.2371403	0.0327581
22	120.9686704	0.0364963
23	395.7680206	0.1048222
24	855.8324934	0.0706066
25	1366.1357973	0.0385439

TABLE I
Results for execution time

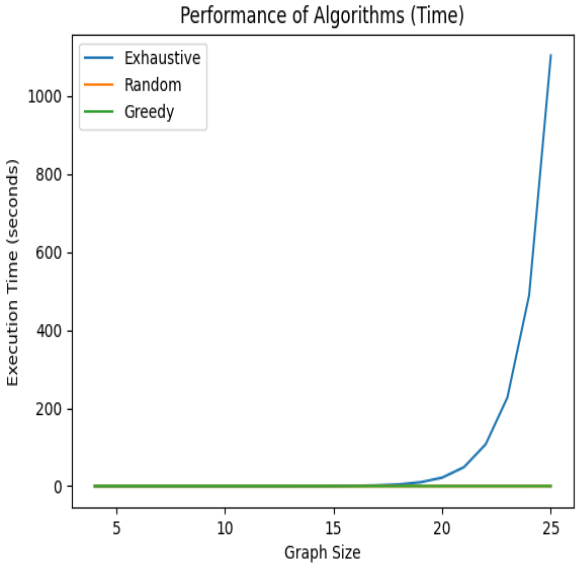

Fig. 2 - Impact of number of vertices in execution time per Algorithm

N. Vertex	Greedy Time(s)
4	0.00000517
5	0.00000684
6	0.00000707
7	0.0001103
8	0.0001843
9	0.0001614
10	0.0001299
11	0.0003236
12	0.0006039
13	0.0004843
14	0.0017958
15	0.0011512
16	0.0013337
17	0.0014015
18	0.0031677
19	0.0018781
20	0.0052803
21	0.0028254
22	0.005311
23	0.0053609
24	0.0084889
25	0.0097374

TABLE II
Results for execution time of greedy algorithm

We can see, in table I an table II, that the greedy algorithm, being analyzed here, performs much better than the other two, in terms of execution time.

In the case of the number of operations carried out by the algorithms, we can see that the greedy algorithm performs considerably less operations than any of the other two algorithms.

N. Vertex	Exhaustive Oper.	Random Oper.
4	1184	8522000
5	6240	11037000
6	37056	14922000
7	172800	17532000
8	886528	21411000
9	4329472	25524000
10	20457472	29437000
11	100683776	34893000
12	528531456	43505000
13	2516680704	50645000
14	11274584064	55882000
15	51540099072	62958000
16	244814249984	73026000
17	1108103790592	81585000
18	5153964687360	93107000
19	22677435187200	101445000
20	102254601306112	113003000
21	442003724697600	121451000
22	1899956206043136	130030000
23	8233143320444928	140091000
24	36732484947279872	154520000
25	158751888005660672	165918000

TABLE III
Results for operations carried out

N. Vertex	Greedy Operations
4	51
5	113
6	138
7	279
8	409
9	537
10	481
11	1279
12	2659
13	2242
14	5617
15	5937
16	7510
17	7652
18	15574
19	10802
20	15161
21	14452
22	24454
23	27717
24	35228
25	47271

TABLE IV
Results for operations carried out for the greedy algorithm

B. Impact of the edges on the algorithms

Now, to carry out these experiments, the same graphs as the first report were used, which have a successively larger probability of edge creation, starting at 10% and going all the way up to 90%, and a constant number of vertices of 18.
Similarly to the previous results, both the random algorithm and the greedy algorithm have the same performance, as we can see in Fig. 3 and Fig. 4.

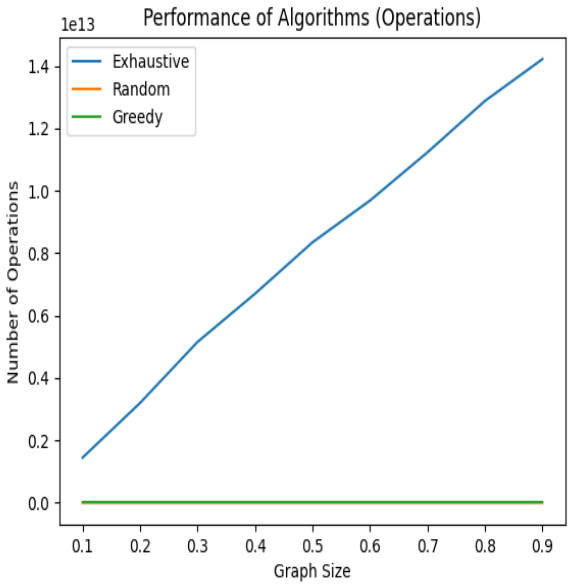

Fig. 3 - Impact of number of edges in operations per Algorithm

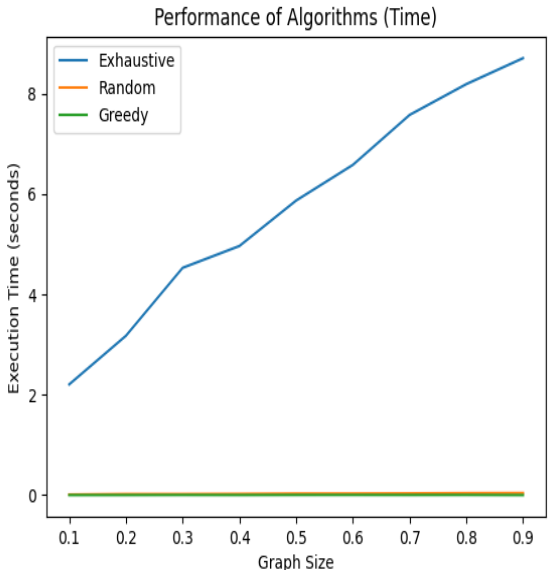

Fig. 4 - Impact of number of edges in execution time per Algorithm

C. Cut

When comparing the result of the approximation to the value of the maximum cut of a graph, we can see that this algorithm is, actually, more efficient than the random algorithm, detailed in the first report. It's possible to observe that the greedy algorithm, loses accuracy for the smaller graphs, when compared to

the random algorithm, but, as the graphs grow larger and larger, the approximations are more accurate, while even being able to actually achieve the value of the maximum cut for several medium-sized and large graphs, seen when comparing with the results obtained from the exhaustive search algorithm.

N. Vertex	Exhaustive	Random	Greedy
4	3	3	3
5	4	4	3
6	5	5	5
7	6	6	6
8	8	8	8
9	10	10	10
10	12	11	10
11	15	15	13
12	18	17	16
13	21	21	18
14	24	22	24
15	27	24	27
16	30	27	30
17	33	31	32
18	38	36	38
19	42	40	41
20	46	43	46
21	50	46	47
22	53	49	53
23	58	51	58
24	64	59	62
25	69	62	67

TABLE V
Calculated Cut per Algorithm

IV. Estimating the cost for larger graphs

Similarly to what was done in the first report, we can do a spacial extrapolation of the results, to try to make a rough estimate of the time required for this algorithm to perform, when using larger graphs. The fitting of the results can be seen in Fig. 5.

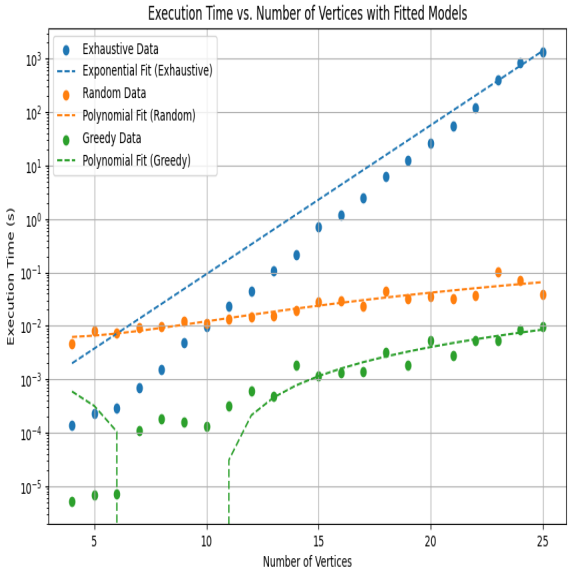

Fig. 5 - Fitting mathematical models

With the tests conducted, approximations to the execution time of this algorithm were obtained, which are seen in table VI.

N. Vertex	Greedy Cut ($\approx s$)
30	0.014617
35	0.022368
40	0.031749
45	0.042760
50	0.055401

TABLE VI
Approximate estimations on execution time

V. Conclusion

To conclude this report, it is safe to say that a better algorithm, than the ones detailed previously, was found as it not only provides better approximations to the real values, it also has a considerable less time consuming execution.
With this, a better comprehension, of how different algorithms have different executions and, therefore, a careful study should take place prior to using them, was gained, which will, most definitely, prove useful in future endeavours.

References

[1] Allen Xiao, “Maxcut problem”, COMPSCI 532: Design and Analysis of Algorithms, 2015.