Solving the Minimum Edge Cover Problem using Exhaustive Search and Greedy Algorithms

Pedro Monteiro 97484
pmapm@ua.pt

***Abstract*—This article main objective is to analyse the Minimum Edge Cover with different algorithms approaches on graphs with different sizes. The computational complexity and number of basic operations are also made in each algorithm.**

***Keywords*—Minimum, Edge Cover, Graph, Exhaustive, Greedy**

I. Introduction

This article comes within the scope of the first advanced algorithms project, where the minimum edge cover of a graph will be demonstrated.

Along with the report it's possible to find the file `generate_graphs.py`, which uses a constant seed to generate graphs with **n** vertices and **p** probability of maximum edges, $p \in [12.5\%, 25\%, 50\%, 75\%]$.

Also, it's possible to find the files `min_edge_cover.py` and `min_edge_cover_greedy.py` where the code of the algorithms can be found.

In addition, the `utils.py` file contains all the functions necessary for graphs visualization, the results, as well as the adjacency and incidence matrices.

II. Description of the Problem

A. Minimum Edge Cover

According to graph theory, an edge cover of a graph is a set of edges such that every vertex of the graph is incident to at least one edge of the set.

The minimum edge cover problem is an optimization problem and consists of finding an edge cover of minimum size, and is denoted ρ(G).

Only connected graphs have edge cover, that is, without isolated vertices.

If a graph G has no isolated vertices, then $\nu(G) + \rho(G) = |G|$ where ν(G) is the size of a maximum independent edge set and n=$|G|$ is the vertex count of G.

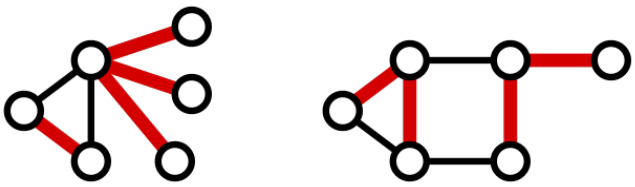

Fig. 1: Minimum Edge Cover Examples.

As can be seen in Figure 1, the graphs have 6 vertices and the minimum number of edges possible for all vertices to be connected to at least one edge is 4, which means that the minimum edge cover of the graphs presented is 4, ρ(G) = 4.

III. Graph Generation

To begin with, graphs are generated, as previously mentioned, through the function **generate_graph(n,p)**, where n is the number of vertices intended for the graph and p is the probability used to calculate the number of edges.

This function returns vertices and edges, the vertices being placed at random positions in the xOy plane with coordinates between [1,20]. To create the edges, vertices are chosen randomly, making sure that all vertices have at least one edge.

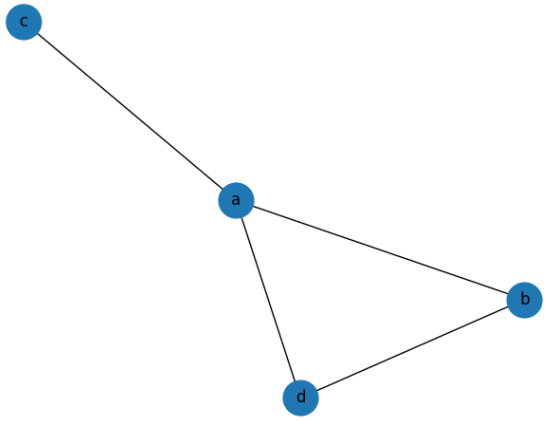

Fig. 2: Graph Example.

In the example of figure 2, the following vertices and edges were used, resulting from the generating function of the graphs:

```
Vertices - [['a', (17, 1)], ['b', (18,
18)], ['c', (11, 13)], ['d', (6, 15)]]

Edges - [('a', 'b'), ('a', 'c'), ('b',
'd'), ('d', 'a')]
```

After generating the graphs and saving them in separate files, the minimum edge cover problem was solved through the exhaustive search algorithm, and otherwise, with a method using greedy heuristics.

IV. Formal Computacional analysis

A. Exhaustive Algorithm

Exhaustive search, also known as brute-force search, consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

In this case, two different approaches were followed:

- analyzing all combinations of edges, smallest to largest subsets
- analyzing all combinations of edges, largest to smallest subsets

Exhaustive Search analyzing all combinations of edges, from smallest to largest subsets

All edge subsets are generated, the algorithm iterates through them all and returns when it finds all vertices in the edge subsets, since this will be the minimum value.

A candidate is a solution when all graph vertices are contained in the edges subset. As the smallest subsets are being generated first, when all vertices are present it is a solution, since is the minimum value, the minimum edge cover.

As can be seen and following the same graph example,

```
subset:  [('a', 'b')]
subset:  [('a', 'c')]
subset:  [('b', 'd')]
subset:  [('d', 'a')]
subset:  [('a', 'b'), ('a', 'c')]
subset:  [('a', 'b'), ('b', 'd')]
subset:  [('a', 'b'), ('d', 'a')]
subset:  [('a', 'c'), ('b', 'd')]
Minimum Edge Cover:  2
```

Fig. 3: Exhaustive Search Example.

Exhaustive Search analyzing all combinations of edges, from largest to smallest subsets

In this case, the edge subsets start to be generated from the highest values to the lowest possible value.

In each iteration the value of the minimum edge cover is updated, if the subset is valid.

Similar to the previous approach, a candidate is a solution when all graph vertices are present in the subset of edges.

This approach is the one that takes the longest to execute since it has to iterate through all the subsets of edges, and it is easy to see that as the number of edges increases considerably, the execution time also increases.

Since it is a problem where the exhaustive search becomes quite slow, the approach adopted was when reaching the solution, the minimum value, it is not necessary to evaluate the following values.

And so, the computational complexity to generate all subsets of edges in the worst case will be,

$$\sum_{n=1}^{n} \binom{n}{k} = 2^n - 1.$$

Exemplifying for the case of a graph with 4 vertices and 4 edges, there will be $2^n - 1$ edges subsets.

According to Pascal's triangle, the line for 2^4 corresponds to [1 4 6 4 1], which means a subset with 0 edges, which is ignored, four subsets with 1 edge, six subsets with 2 edges, four subsets with 3 edges and one subset with all 4 edges.

These subsets represent all the subsets that it is possible to create with the edges of the graph.

Since the complexity of visiting each vertex and each subset of these is O(n^2), it can be concluded that the complexity of the problem is O(n^2) * O(2^n) = O($n^2 2^n$)

B. Greedy Algorithm

One way to speed up a brute-force algorithm is to reduce the search space, that is, the set of candidate solutions, by using heuristics specific to the problem class.

With this algorithm it is then possible to reduce the time until a solution is found, defining a heuristic.

Unlike the exhaustive search algorithm, where all subsets of edges are generated, with greedy approach, that was developed, there was no edges generation. Instead, a graph was built from scratch, always trying to use the minimum number of edges and cover all vertices.

For this was used the adjacency list, represented by a dictionary in python, where the keys are the vertices and the values are the lists of neighbors.

Starting by sorting the adjacency list by the vertices with the fewest neighbors and then iterating to connect the vertex with the fewest neighbors to the neighbor with the fewest neighbors, repeating the process until all vertices are covered by at least one edge.

The computational complexity of sorting the vertices in the adjacency list, according to their neighbors, is O(n log n).

Iterating over the vertices of the graph is also O(n log n), so it can be concluded that the complexity of the algorithm, in the worst case, can be expressed through: $O(nlogn) * O(nlogn) = O(n^2 log^2 n)$.

V. Experiments

The vertices [3,4,5,7,8,9,11,12,13,14,15] were chosen in order to show how both algorithms work, a set of vertices was selected where experiments were performed.

A. Number of Basic Operations

Table I shows the number of basic operations related to exhaustive search and table II shows the number of basic operations related to greedy search, for graphs with different numbers of vertices and edge densities.

Exhaustive Search			
n	25%	50%	75%
3			4
4		9	16
5		55	154
7		1300	7050
8	294	5292	28371
9	1467	57852	483354
11	32630	2259414	31164059
12	79104	8013225	94342493
13	592420	130479024	2372274356
14	1807520	373277070	7498360544
15	18881356	7093094944	

TABLE I: Exhaustive Search Basic Operations Number.

Greedy Search			
n	25%	50%	75%
3			8
4		5	5
5		17	22
7		27	59
8	9	20	62
9	20	50	105
11	33	91	167
12	33	107	175
13	42	110	274
14	37	146	281
15	68	194	421

TABLE II: Greedy Search Basic Operations Number.

Through the tables is possible to see that the number of basic operations increases in both algorithms, however it is worth noting the fact that in the exhaustive algorithm the values grow much more, reaching values that are too large, and for n = 15 the number of basic operations is already in the order of 10^5.

In the case of the exhaustive algorithm is notable that its growth is large and this is due to the fact that with the increase in the number of vertices, the number of edges will also increase. As this algorithm generates all possible subsets of edges for a given graph, iterating them in order to find the solution to the problem, an increase in the number of edges results in a drastic increase in the number of operations required.

And so, is important to note that for the same number of vertices, in denser graphs the number of basic operations is greater, being relatively smaller in sparse graphs.

On the other hand, for the greedy approach, as no subsets of edges are generated, and consequently, it is not necessary to iterate all these possible solutions, the algorithm is quite fast, and performs much less operations, since from the provided edges it seeks to build a graph from scratch, connecting vertices to their neighbors.

B. Execution Times

Table III and Table IV show the execution times for the exhaustive algorithm and for the greedy algorithm, respectively, for graphs with different numbers of vertices and edge densities.

Exhaustive Search			
n	25%	50%	75%
3			$8.6 * 10^{-6}$
4		$9.06 * 10^{-6}$	$8.8 * 10^{-6}$
5		$2.18 * 10^{-5}$	$5.17 * 10^{-5}$
7		$3.79 * 10^{-4}$	$2.06 * 10^{-3}$
8	$1.21 * 10^{-4}$	$1.56 * 10^{-3}$	$8.06 * 10^{-3}$
9	$4.4 * 10^{-4}$	0.0167	0.141
11	$6.05 * 10^{-3}$	0.656	9.039
12	0.0238	2.351	30.93
13	0.205	40.59	784.76
14	0.599	120.22	784.76
15	5.70	2508	

TABLE III: Exhaustive Search Execution Times.

Greedy Search			
n	25%	50%	75%
3			$1.90 * 10^{-5}$
4		$1.1 * 10^{-5}$	$1.31 * 10^{-5}$
5		$1.67 * 10^{-5}$	$3.48 * 10^{-5}$
7		$2.65 * 10^{-5}$	$3.93 * 10^{-5}$
8	$1.72 * 10^{-5}$	$2.50 * 10^{-5}$	$3.91 * 10^{-5}$
9	$2.29 * 10^{-5}$	$4.17 * 10^{-5}$	$5.20 * 10^{-5}$
11	$2.84 * 10^{-5}$	$5.53 * 10^{-5}$	$7.70 * 10^{-5}$
12	$3.15 * 10^{-5}$	$5.05 * 10^{-5}$	$9.11 * 10^{-5}$
13	$5.15 * 10^{-5}$	$9.06 * 10^{-5}$	$1.74 * 10^{-4}$
14	$5.03 * 10^{-5}$	$1.15 * 10^{-4}$	$1.72 * 10^{-4}$
15	$6.94 * 10^{-5}$	$1.39 * 10^{--4}$	$2.49 * 10^{-4}$

TABLE IV: Greedy Search Execution Times.

In a similar way to the number of operations, and as can be seen in the tables above, the execution times for the exhaustive algorithm are much larger compared to the greedy algorithm.

As the number of vertices increases, and consequently the number of edges, the execution times increase significantly for the exhaustive search, not the same for the greedy approximation, which executes much faster, in all cases.

On the other hand, is also possible to observe that, in the case of exhaustive search, the growth is more accentuated in the experimental values. It is then concluded that the complexity is even higher than that calculated previously in the formal analysis.

C. Number of Solutions/Configurations Tested

In the following table can be seen the number of tested configurations, for graphs with different numbers of vertices and edge densities.

Exhaustive Search			
n	25%	50%	75%
3			3
4		7	10
5		25	63
7		385	1940
8	119	1470	7546
9	381	12615	101583
11	4095	397593	5358577
12	14892	1391841	16122225
13	94183	19311487	346188399
14	280599	54910659	1090188223
15	2533986	909574393	

TABLE V: Exhaustive Search Number of Configurations Tested.

As discussed in section II, the exhaustive algorithm used returns when it finds a value that is a solution, i.e. the minimum edge cover value, so it is not necessary to continue running the algorithm, since all the following values will not be a solution, because are greater than the minimum value.

The number of configurations tested will be equal to the number of subsets of edges analyzed, being therefore, in the best case 1, if the solution corresponds to the first subset, and in the worst case, corresponding to the last subset of edges generated.

As the table shows, with the increase in the number of vertices, it is expected that the number of configurations tested will also increase, since more edges are needed to connect them, so it is necessary to analyze more subsets of edges.

Regarding the greedy algorithm, the number of tested configurations is always one, since the solution is built from scratch connecting the vertices, using as few edges as possible. When all vertices are connected the algorithm ends.

VI. Experimental and formal analisys comparison

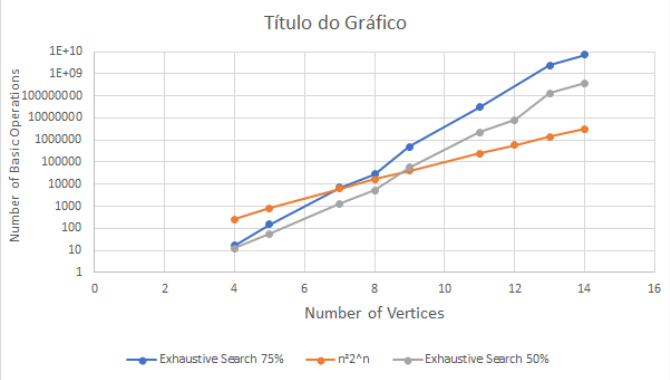

Fig. 4: Exhaustive Search Comparison of Number of Basic Operations with Formal Analisys.

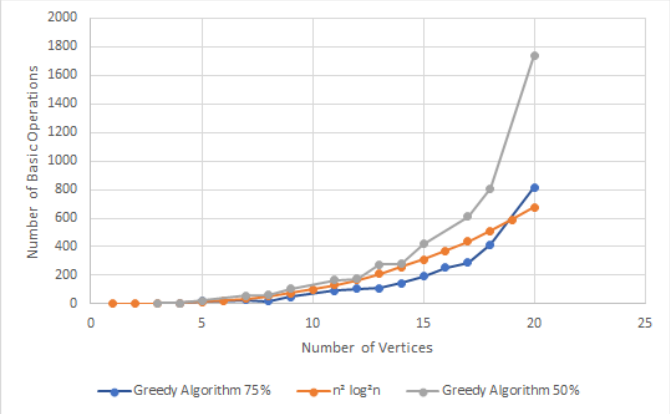

Fig. 5: Greedy Algorithm Comparison of Number of Basic Operations with Formal Analisys.

Looking for the graphs that compare the number of operations with the formal analysis is possible to see that the behavior of greedy algorithm is similar to that described in the formal analysis, which means, it correctly describes the complexity of the algorithm, so it can be said that it is a good starting point to estimate execution times and resources for graphs with higher vertices number.

VII. Larger graph processed

Regarding the exhaustive search, and as it was expected to be a slower algorithm, the largest graph that was processed had 15 vertices and 50% edge density.

For this graph, the algorithm already took about 40 minutes, which is a high execution time, also due to the fact that its computational complexity is high.

With the greedy algorithm, unlike brute-force, it was possible to test larger graphs with more edge density, since it is an extremely fast algorithm, due to the simple fact that there is no generation of edges, nor the need to iterate them.

Therefore, it was possible to process a graph with 51 vertices and 75% edge density, that is, about 1000 edges. In contrast to the previous algorithm, which for 15 vertices already took a considerable time, for 51 vertices, the greedy algorithm only took 0.03s.

VIII. Larger problem instances

As already mentioned, for the exhaustive search, as the number of vertices increases, and consequently the number of edges, the execution time increases dramatically.

For graphs with 12.5% and 25% the algorithm still manages to solve the problem in a reasonably good time. As the graphs become denser, for the 50% and 75% cases, there is an increased difficulty in executing the algorithm.

For example, for a graph with 15 vertices and 52 edges, the algorithm takes approximately 2555s.

With this, it is concluded that in the case of exhaustive search, the number of edges greatly affects the performance of the algorithm, running well in sparse graphs, but being very expensive in denser graphs.

It is possible to estimate, approximately, the time that the algorithm needs to run in graphs with larger sizes, through the expression calculated in section II.

Note the fact that the density of edges in a graph affects performance a lot, for that, different times were chosen, corresponding to different densities for a graph with 12 vertices. In the case of a 25% density, with 12 vertices the algorithm takes about 0.03s to run, but with a density of 75%, the algorithm takes about 32s, so for larger graphs the execution time can be approximated by the following expression, where t is the time, in seconds, that the algorithm takes to execute for a graph with 12 vertices:

$$exhaustive\_execution\_time(n) = \frac{n^2 2^n}{12^2 2^{12}} * t$$

Unlike the previous algorithm, the greedy algorithm proves to be quite fast and efficient in solving the problem, with the execution times, with graphs up to 20 vertices, in the order of

10^{-5}, regardless of edges number to the performance of the algorithm.

Similarly to the exhaustive search, is possible to approximate the execution times for graphs of larger sizes, using the expression calculated previously. For this, a graph with 20 vertices was chosen, where the execution time, t, in seconds, also varies according to the edges density, although the variation is much smaller:

$$greedy\_execution\_time(n) = \frac{n^2 log n^2}{20^2 log 20^2} * t$$

Through the approximation functions calculated above it is possible to estimate the time required for the algorithms in relatively larger graphs, as shown in table VI.

Exhaustive Search			
n	25%	50%	75%
12	0.024	2.35	30.9
20	17	1671	21973
50	$1.14 * 10^{11}$	$1.12 * 10^{13}$	$1.48 * 10^{4}$
100	$5.16 * 10^{26}$	$5.05 * 10^{28}$	$6.6 * 10^{29}$
250			
500			
1000			
10000			
100000			

TABLE VI: Exhaustive Search Larger Graphs Execution Times

Greedy Algorithm				
n	12.5%	25%	50%	75%
12	$1 * 10^5$	$2 * 10^5$	$4 * 10^5$	$9 * 10^5$
20	$5.3 * 10^5$	$9.87 * 10^5$	0.00025	0.00039
50	0.00056	0.00105	0.0027	0.00415
100	0.00313	0.00583	0.01477	0.02304
250	0.0281	0.05238	0.13269	0.20701
500	0.14255	0.2655	0.672	1.049
1000	0.7045	1.3120	3.323	5.184
10000	125	233	591	921
100000	$1.96 * 10^4$	$3.6 * 10^4$	$9.2 * 10^4$	$1.4 * 10^5$

TABLE VII: Greedy Algorithm Larger Graphs Execution Times

Comparing the results obtained, is observed that the exhaustive search is extremely expensive, and the execution time for a graph with 20 vertices is relatively high.

In the case of the greedy algorithm, the same does not happen, it has a very efficient behavior, being able to easily process graphs with vertices in the order of 10^3.

IX. Conclusion

To conclude this article, as expected, exhaustive search, also known as brute force, is an algorithm that requires some time to run. It can be very useful for small problems, as it is quite simple to implement and allows you to easily achieve a correct solution.

On the other hand, the greedy algorithm is extremely fast, has a small computational cost and can be useful to get an approximate solution quickly.

References

[1] Edge Cover https://en.wikipedia.org/wiki/Edge_cover

[2] Minimum Edge Cover https://mathworld.wolfram.com/MinimumEdgeCover.html

[3] Time Complexity https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

[4] Python Time Complexity https://wiki.python.org/moin/TimeComplexity