Solving the Minimum Edge Cover Problem using Randomized Algorithms

Pedro Monteiro 97484
Advanced Algorithms
Course Instructor: Joaquim Madeira
DETI, Aveiro University
Aveiro, Portugal
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, Randomized, Algorithm, Monte Carlo, Las Vegas**

I. Introduction

This article comes within the scope of the second advanced algorithms project, where a randomized algorithm will be developed to solve the minimum edge cover problem, taking into account graphs created through a previously available generator, but also larger graphs available on the Web.

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

Also, it's possible to find the file `randomized.py` where the code of the algorithm can be found [1].

For larger graphs, the file `larger_randomized.py` was used.

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 [2] 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.

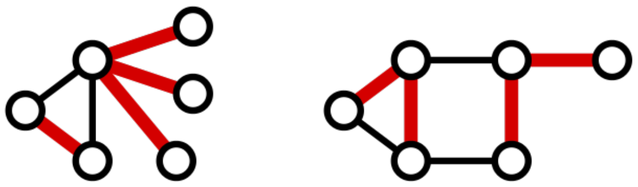

Fig. 1: Minimum Edge Cover Examples.

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.

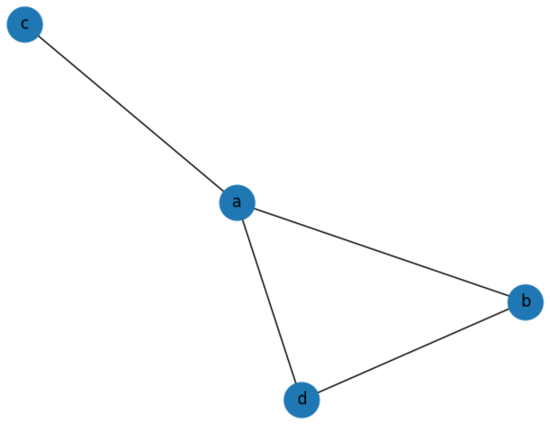

Fig. 2: Graph Example.

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. Randomized Algorithms

A randomized algorithm [3] is an algorithm that employs a degree of randomness as part of its logic or procedure. It is typically used to reduce either the running time, or time complexity; or the memory used, or space complexity, in a standard algorithm.

A randomized algorithm could help in a situation of doubt by flipping a coin or a drawing a card from a deck in order to make a decision. Similarly, this kind of algorithm could help speed up a brute force process by randomly sampling the input in order to obtain a solution that may not be totally optimal, but will be good enough for the specified purposes.

Randomized algorithms are usually designed in one of two common forms: as a **Las Vegas** or as a **Monte Carlo** algorithm.

A. Las Vegas Algorithms

A Las Vegas algorithm runs within a specified amount of time. If it finds a solution within that timeframe, the solution will be exactly correct. However, it is possible that it runs out of time and does not find any solutions.

The running time however is not fixed (not deterministic), that is it can vary for the same input. For eg. Randomized Quick Sort always gives a correctly sorted array as its output. However it takes O($nlogn$) time on average but can be as bad as O(n^2) in the worst case.

B. Monte Carlo Algorithms

On the other hand, Monte Carlo algorithms are probabilistic algorithms which, depending on the input, have a small probability of producing an incorrect result or failing to produce a result. Rely on repeated random sampling to achieve numerical results, and are often used in physical and mathematical problems.

The running time for these algorithms is fixed however. There is also generally a probability guarantee that the answer is right.

Most Monte Carlo algorithms can be converted to Las Vegas algorithms by adding a check to see if the answer given is right and if not running the Monte Carlo algorithm again till the right answer is produced.

V. Description of the Solution

The solution developed contains two different parts and starts by saving the vertices number of the graph, making a copy of the edges list and initializing a set that will contain vertices of the graph.

As mentioned earlier, the minimum edge cover of a graph corresponds to the minimum number of edges where all vertices are connected to at least one. For this, a random edge of the graph was chosen, using **random.choice**, and its vertices were stored in the set, so that there were no repeated vertices. To avoid choosing the same edge, it was eliminated from the list with the copy of all edges. This whole process takes place while the size of the set is not equal to the number of vertices. That is, the process ends when all vertices have been chosen, which means that all are connected to at least one edge. In the end it is possible to obtain an edge cover of the graph.

The second part of the algorithm, in order for this value to be the minimum, the algorithm is run a certain number of times, according to its size. This avoids running the algorithm a high number of times for relatively small graphs. Also the solutions found were validated in order not to test a solution already tested.

Being the possible number of solutions $2^n - 1$, where n is the number of edges of the graph, since it corresponds to the number of combinations of edges for a graph. For example, a graph with 4 vertices and 3 edges has $2^3 - 1$ sets of possible edges, that is 7, E1, E2, E3, E12, E13, E23, E123.

As it is easily noticed, this number tends to grow a lot with the increase in the size of the graphs and the consequent increase in the edges number. To combat this increase, the number of simulations to be carried out is obtained through:

$$max([100, round(log2(2^{num\_edges}) * num\_edges * vertices * 10)])$$

Through this formula is possible to obtain a more reasonable number of the set of possible solutions, through log2. In order for this value of simulations to be variable with the size of the graphs, the number of edges and the number of vertices was also added. After some tests and after running the algorithm several times, was noticed that a little more simulations would be necessary, thus multiplying by 10. For smaller graphs, 100 simulations are used.

Also, a Monte Carlo approach was implemented. The algorithm is run according to this number of simulations, but to ensure that there is a solution at the end, although it may not be the optimal solution, a computational resource limit has been set. That said, the algorithm only runs for a maximum of 5 minutes [4], returning the best solution found until then.

VI. Formal Analisys

In order to study the complexity of the developed algorithm, first is necessary to understand how it works.

As previously mentioned, to begin with, the necessary variables are defined, namely the min_edge variable, which will contain the value that solves the problem, the edges_copy

Algorithm 1 Randomized Algorithm

```
$edges\_copy \leftarrow edges[:]$        ▷ Copy edges list from graph
$unique\_vertices \leftarrow set()$
$min\_edge \leftarrow 0$
**while** $len(vertices) \neq len(unique\_vertices)$ **do**
    $selected\_edge \leftarrow random.choice(edges\_copy)$
    **if** $selected\_edge$ in $edges\_copy$ **then**
        $edges\_copy.remove(selected\_edge)$
        $unique\_vertices.add(selected\_edge[0])$
        $unique\_vertices.add(selected\_edge[1])$
        $min\_edge \leftarrow +1$
    **end if**
**end while**
**return** $min\_edge$
```

which contains a copy of all the edges of the graph and the unique_vertices which is an initialized empty set and which will contain the vertices belonging to the chosen edges.

According to the pseudocode shown above:

Then the algorithm will make a loop, where an edge of the graph is randomly chosen. This loop will continue until the number of vertices is equal to the size of the set containing the vertices of the randomly chosen edge. That is, it will occur v times, where v is the number of vertices, O(v).

In order to guarantee that the same edge is not chosen more than once, and the algorithm is in a dead lock, the edge is deleted from the list (edges_copy). The complexity to remove the edge from the list is O(e) [5]. So the complexity of the algorithm is O(v) * O(e) = O(ve).

Should be noted that this complexity is only relative to the search for an edge cover, which is subsequently run to obtain better values.

VII. Experiments

Experiments were performed using [3,4,5,7,8,9,11, 12,13,14,15,17,19,20] as number of vertices.

A. Number of Basic Operations

Table I shows the number of basic operations related to the algorithm, for graphs with different numbers of vertices and edge densities.

Randomized Search			
n	25%	50%	75%
3			1500
4		7077	13443
5		32103	66570
7		289443	813420
8	163512	876546	1952526
9	350718	1723440	4216272
11	1240224	6439026	15597084
12	2210736	12736958	27442536
13	3913944	20857660	47588228
14	6560262	32540126	75634010
15	10814136	53725774	117386754
17	28244844	116809852	270011690
19	53188390	253725154	345961729
20	82259496	33371329	354087599

TABLE I: Randomized Search Basic Operations Number.

Looking at the table, there is a high number of operations. This is due to the fact that the algorithm needs to be run several times, in order to obtain an increasingly better solution, since the choices are made randomly and the optimal solution may not appear. It is also necessary to ensure that there are no deadlocks.

Note the fact that these operations are relative to the algorithm to be run during several simulations.

B. Time Execution

As is visible in the table II, the random algorithm presents good execution times, for relatively smaller graphs, since up to about 12 vertices takes approximately 10 seconds at most.

With the growth of the graphs, that is, the increase in the number of edges and vertices, the algorithm becomes more difficult to execute, starting to slow down.

This happens due to the increase in the number of simulations, since for larger graphs it is necessary to perform more simulations to obtain a correct solution. However, this increase is very high, making the number of simulations impossible to run.

To work around this, an execution time limit was implemented, and the algorithm can only iterate for a maximum of 5 minutes, 300 seconds.

Randomized Search			
n	25%	50%	75%
3			0.0041
4		0.0137	0.0261
5		0.0618	0.1327
7		0.5954	0.3227
8	0.3317	1.2636	0.5293
9	0.7314	0.4632	1.1497
11	1.8471	1.9282	6.8327
12	1.4372	4.5315	12.8145
13	2.1145	7.1298	20.3927
14	4.1192	11.8531	36.1075
15	3.3164	20.9241	60.4408
17	9.8727	53.7415	185.3330
19	20.4121	146.7402	300**
20	39.9407	230.6340	300**

** - timeout value (5 minutes)

TABLE II: Randomized Search Execution Times.

C. Configurations Tested

Table III shows the solutions/configurations tested by the algorithm.

Randomized Search			
n	25%	50%	75%
3			1
4		1	7
5		17	77
7		348	15711
8	6	9137	35280
9	45	29160	65610
11	12403	80190	184910
12	30601	130680	288120
13	46868	197730	437320
14	67711	283500	647360
15	101400	405600	912600
17	196520	786080	1768680
19	335160	1372750	1952181
20	441800	1805000	1878225

TABLE III: Randomized Search Configurations Tested.

It is important to ensure that all these solutions are different, that is, the same solution is not tested more than once, thus avoiding wasting time running code that could fall into an infinite loop.

VIII. Another Graphs Experiments

In addition to the graphs already created in the previous project, several graphs were also used through the pointers provided by the teacher.

Among all, highlight the Sedgewick & Wayne graphs and the graphs present in the Stanford Large Network Dataset Collection [6] that were used for testing.

It was necessary to create a new function to read the graphs, since the way they were stored was different.

For these graphs, since most are relatively much larger, the way the algorithm runs has been significantly changed. Instead of looking for the number of possible solutions, $2^n - 1$, where n is the number of edges, and then running the algorithm for a part of that number, in these cases a fixed number was used. The number of simulations was then fixed at 1000, avoiding calculating extremely high numbers.

Starting with Sedgewick & Wayne graphs, 2 different types were tested, tiny and medium. The tiny graph has 13 vertices and is therefore very similar to those created by the generator. Medium has 250 vertices and about 1270 edges.

Randomized Search			
graph	operations	time	solutions
SW tiny	0.024	2.35	30.9
SW medium	17	1671	21973

TABLE IV: Comparison Between SW Tiny and Medium Graph

When looking at the table, it is easy to see that the algorithm becomes costly as the number of vertices, and consequently the number of edges, increases.

It should also be noted that when using a fixed number for the number of simulations, the result for larger graphs is not optimal and may be a little distant from it, since the number of possible combinations increases drastically.

In this case, for the medium graph, the result obtained for the minimum edge cover value was around 390, and the value for 250 vertices is close to 125 [7].

At the Stanford Large Network Dataset Collection [6] it is possible to find many and different types of graphs, where for this project two of them were chosen, the graph about LastFM Asia Social Network [8] and the graph General Relativity and Quantum Cosmology collaboration network [9].

Both graphs contain a large number of vertices and edges, having 7624 vertices and 27,806 edges for Asia Social Network and 5242 vertices and 14496 edges for Facebook.

Randomized Search			
graph	operations	time	solutions
Social Network	83397	1203.24	138
Quantum Cosmology	86529	1210.07	125

TABLE V: Comparison Between Graphs Available on Web

IX. Algorithms Comparison

In order to understand which algorithm best behaves, a comparison was made for the results obtained between all of them.

Recalling the previous project, an exhaustive search algorithm was developed, which creates a list of solutions and exhaustively tests them all, and a greedy algorithm, where there is no generation of solutions, on the contrary there is the creation of a solution from scratch according to some rules, in order to improve performance.

50% edge density			
n	Exhaustive	Greedy	Randomized
4	9	5	7077
5	55	17	32103
7	1300	27	289443
8	5292	20	876546
9	57852	50	1723440
11	2259414	91	6439026
12	8013225	107	12736958
13	130479024	110	20857660
14	373277070	146	32540126
15	7093094944	194	53725774

TABLE VI: Comparison Between Graphs with 50% Edge Density

Looking at the results there is a marked difference between the values of the exhaustive and randomized algorithms with the greedy algorithm.

For the greedy algorithm the number of operations does not reach 200, while for the exhaustive search and for the greedy algorithm the values go up to 10^9. Between these last two algorithms, the randomized one starts with high values being overtaken by the exhaustive search as the number of vertices increases.

75% edge density			
n	Exhaustive	Greedy	Randomized
3	4	8	1500
4	16	5	13443
5	154	22	66570
7	7050	59	813420
8	28371	62	1952526
9	483354	105	4216272
11	31164059	167	15597084
12	94342493	175	27442536
13	2372274356	274	47588228
14	7498360544	281	75634010
15		421	117386754

TABLE VII: Comparison Between Graphs with 75% Edge Density

Similar to graphs with 50% density, these also present a very high number of operations for the random and exhaustive algorithms.

On the other hand, even though the graphs are denser, the greedy algorithm remains very efficient, suffering only a slight increase in the number of operations.

Comparing the results obtained for all algorithms in relation to the generated graphs, as can be seen in tables VI and VII, it's easy to see that the exhaustive search is extremely costly.

The number of edges consequently increases with the number of vertices, which implies that there are more possible solutions. As the exhaustive search is a brute force algorithm, it tries to test them all, noting a high increase in the number of basic operations.

In terms of execution time, and as you can see in tables VIII, IX and X, the greedy algorithm is the one with the best times.

Regarding the exhaustive search, and as mentioned earlier that it is a very time consuming algorithm, and from 15 vertices the execution time is already very high.

The randomized algorithm is quite efficient for graphs with relatively few vertices. As the number of vertices increases, and similarly to the exhaustive search, the number of possible solutions increases, which implies that the number of simulations in the randomized algorithm also increases, which makes it more time consuming. Even following a Monte Carlo approach, limiting the algorithm to just some running time, ensuring that there is always a solution, this solution may not always be the optimal solution.

On the other hand, the greedy algorithm is efficient and performs well overall. This is easily explained by the fact that in this algorithm there is no generation of possible solutions, since a graph is built from scratch.

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 VIII: 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 IX: Greedy Search Execution Times.

Randomized Search			
n	25%	50%	75%
3			0.0041
4		0.0137	0.0261
5		0.0618	0.1327
7		0.5954	0.3227
8	0.3317	1.2636	0.5293
9	0.7314	0.4632	1.1497
11	1.8471	1.9282	6.8327
12	1.4372	4.5315	12.8145
13	2.1145	7.1298	20.3927
14	4.1192	11.8531	36.1075
15	3.3164	20.9241	60.4408

TABLE X: Randomized Search Execution Times.

Regarding the graphs available on the web, tables XI and XII show the results for the greedy and randomized algorithms. The exhaustive search was not considered due to the fact that is not possible to run and obtain results for these larger graphs.

Basic Operations on graphs Available on Web		
graph	Greedy	Randomized
SW tiny	23	12.425
SW medium	8859	1200
Asia Social Network	8859	15859
Quantum Cosmology		126529

TABLE XI: Basic Operations Comparison on Graphs Available on Web

Execution Times on graphs Available on Web		
graph	Greedy	Randomized
SW tiny	$3.6954 * 10^{-5}$	12.425
SW medium	0.0043	1200
Asia Social Network	1.8067	1203.24
Quantum Cosmology		1210.068

TABLE XII: Execution Times Comparison on Graphs Available on Web

Looking at the results obtained, the greedy algorithm clearly stands out, being the one that presents the best results, with the best times and the lowest number of operations performed, only testing one solution. Furthermore, the value obtained for the minimum edge cover is very close to the optimal value.

The fact that the exhaustive algorithm searches all the candidates and checks if they are a solution makes it quite time consuming, and therefore a slower algorithm.

In the case of randomized algorithm, since the edge is chosen randomly, it's necessary to run the algorithm several times, so that the result obtained is optimal or very close to it. The number of times the algorithm should run is obtained through the number of possible solutions, this number being easily scalable due to the increase in the number of edges (2^n, where n is the edges number). Furthermore, this algorithm is guaranteed to have a solution.

X. Larger graph processed

As the graphs created in project one are relatively small compared to those available on the web, the largest graph tested was the Asia Social Network graph, which has 7624 vertices and 27,806 edges [8].

This number of edges is quite high, which leads to a high number of combinations of possible solutions. As previously mentioned, in these cases a fixed number of simulations was used.

The randomized algorithm for this graph did not perform well, despite returning a solution. This is due to the fact that after 20 minutes of execution it returns a solution, even if it may not be optimal.

XI. Conclusion

To conclude this article, randomized algorithms can prove to be relatively fast in solving problems, however there may not always be a guarantee of a right solution.

They are typically used to reduce either the running time, or time complexity, or the memory used.

In the case demonstrated in this article, the randomized algorithm was effectively more effective than the exhaustive search, with better execution times and better performance. On the other hand, the greedy algorithm surpassed the randomized one, both in terms of execution time and in terms of basic operations performed.

Acknowledgment

I would like to thank Professor Joaquim Madeira, regent of AA course, for being available to answer questions to discuss the project and for the flexibility to change the delivery date.

References

[1] Github https://github.com/pedromonteiro01/AA/tree/main/project2
[2] Edge Cover https://en.wikipedia.org/wiki/Edge_cover
[3] Randomized Algorithms https://en.wikipedia.org/wiki/Randomized_algorithm
[4] Break loop after Timeout https://stackoverflow.com/questions/13293269/how-would-i-stop-a-while-loop-after-n-amount-of-time
[5] Python Time Complexity https://wiki.python.org/moin/TimeComplexity https://snap.stanford.edu/data/
[6] Stanford Large Network Dataset Collection https://snap.stanford.edu/data/
[7] Python Calculate Minimum Edge Cover https://www.geeksforgeeks.org/program-to-calculate-the-edge-cover-of-a-graph/
[8] LastFM Asia Social Network https://snap.stanford.edu/data/feather-lastfm-social.html
[9] General Relativity and Quantum Cosmology collaboration network https://snap.stanford.edu/data/ca-GrQc.html
[10] Minimum Edge Cover https://mathworld.wolfram.com/MinimumEdgeCover.html
[11] Time Complexity https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7
[12] Python Time Complexity https://wiki.python.org/moin/TimeComplexity
[13] Randomized Algorithms https://brilliant.org/wiki/randomized-algorithms-overview/