Analysing Minimum Vertex Cover On Undirected Graphs

Vasco Regal

Abstract **–Minimum vertex cover is an optimization problem that aims to find, given an undirected graph, the minimum sized set of vertexes which includes one or more endpoints of every edge of the graph. This paper aims to present a computational analysis of two algorithms: an exhuastive approach and an heuristic based aproximation, aided by experiments with a python implementation.**

Resumo **–Cobertura de vértices mínima é um algoritmo de otimização que procura encontrar, dado um grafo não direcionado, o conjunto de comrimento mínimo de vértices que inclui um ou mais pontos de todas as arestas do grafo. Este artigo apresenta uma análise computacional de dois algoritmos: um exaustivo e um de aproximação baseado em heurísticas, com recurso a experiências efetuadas numa implementação em python.**

Keywords **–graphs, optimization, min-vertex-cover, computational-analysis**

Palavras chave **–grafos, otimização, cobertura-minima, analise-computacional**

I. The Minimum Vertex Cover

A. Definition

Given an undirected graph, *G = (V, E)* and *V'* a subset of *V*, *V'* is a vertex cover if:

$$\forall\ u,v \in E \implies u \in V' \vee v \in V'$$

In other words, a subset of the graph's vertexes in which all edges of the graph are incident in at least one of the elements of the subset. The vertex cover of minimum size is called a **minimum vertex cover**.

B. Algorithms

Algorithm 1 Exhaustive Minimum Vertex Cover

```
1:  procedure MinimumVertexCover(V, E)
2:      current_best ← |V|
3:      for subset ∈ Subsets(V) do
4:          if IsVertexCover(subset) then
5:              if |subset| < current_best then
6:                  current_best ← |subset|
7:              end if
8:          end if
9:      end for
10:     return current_best
11: end procedure
```

$$score(v) = edgesUncovered(C) - edgesUncovered(C')$$

Fig. 1
Formula to calculate the score of a Vertex

Algorithm 1 consists in generating all possible combinations of vertexes for the given graph and analyse each combination. We then return the subset that contains the vertex cover of minimum length.

Algorithm 2 Greedy Heuristic Minimum Vertex Cover Approximation

```
1:  procedure MinimumVertexCover(V, E)
2:      C ← []
3:      best_candidates ← V
4:      while IsNotVertexCover(C) do
5:          v ← Random(best_candidates)
6:          AddVertexToSolution(v, C)
7:          UpdateScores(C)
8:          best_candidates ← Neighbors(C)
9:          RemoveVertexWithLowestScore(best_candidate
10:     end while
11: end procedure
```

As for algorithm 2, a score is calculated for each vertex to form a list of best candidates. In other words, given an initial random vertex from V, the next vertex to be expanded is the one with the highest score. This score is calculated according to Formula 1, where C is the current solution and C' is the solution with the new vertex v. This function was proposed on [1].

II. Formal Analysis

A. Exhaustive

A.1 Time Complexity

Since the exhaustive algorithm requires the analysis of every possible solution, it's an **NP-Complete** problem which means we can't solve it in polynomial time [3]. Although analysing one solution, in this case, verifying if the vertexes create a Vertex Cover, is a quick task, the procedure requires the computation of 2^V possible solutions, being V the total number of vertexes.

A.2 Basic Operations

Entrypoint:

- **Assignments**: 1

- **List appends** : 2^V

For each iteration:

- **Assignments** : 3 to 4
- **Comparisons** : 4
- **List appends** : 0 to 2
- **Sums** : 0 to 2

B. Greedy Heuristic

B.1 Time Complexity

For this algorithm the time complexity can be verified by looking at the different operations during the algorithm.

On each iteration, we need to calculate the total number of neighbors in the current solution 2, line 8. The time complexity for this step, being *avg* the average edges covered in each call is $O(|E|/avg) = O(|E|)$. Also to update the scores of each vertex and subsequent neighbors in current solution, the time complexity is $O(|V| + |Neighbors(C)|) = O(|V|)$. So the complexity of each loop is $O(|E| * |V|)$

B.2 Basic Operations

Entrypoint:

- **Assignments** : 2
- **Scores calculated** : V
- **List remove**: 1

For each iteration:

- **Comparisons** : 2
- **List fetch** : 1
- **List append** : 1
- **Assignments** : 1
- **Scores calculated** : C (current solution)
- **List remove** : 0 to 1

III. IMPLEMENTATION

For this project, three different python modules were developed: the problem, the solution and the analysis. All code is available on github.

A. Problem

In the implementation, graphs are represented as objects containing both a list of Vertex objects and a list of Edge objects. The Graph class provides functions to simulate adjacency and incidence matrices.

The Problem class contains a range of functions not only to generate the random graph but also to allow utilities like exporting the results to a file for later analysis or plotting the graph. In this class we also have the entrypoint to solve our problem, the **solve()** function which will call a method from a Solver object, discussed below.

B. Solution

B.1 Solver

The core of the developed solution is the **Solver** class, which accepts a Problem instance as argument. The class itself is in fact meant as an abstract class, from which the two algorithms' implementations will inherit. This super class has only one method defined, **solve()**, which returns a list of Vertex objects (the min vertex cover).

B.2 ExhaustiveSolver

The first subclass developed implements the algorithm presented on 1 as its **search()** method. The 2^V subsets are iterated and analysed for the presence of a vertex cover. The one with the smallest length is returned.

B.3 GreedySolver

For the greedy heuristic version, the 2 apporach was used. After creating the adequate functions to give costs to solutions and scores to Vertexes, a list of best vertex candidates is generated on each iteration based on the score they would obtain by joining the solution.

C. Analysis

C.1 Running experiments

To automate the experiments and respective analysis, a CLI was developed to make our solution script friendly. The main python script accepts a wide range of options, like specifying the number of vertexes to use, the % edges, which Solver subclass to implement the solution and export runs to a specific directory.

Having this interface setup, with a simple shell script, 20 runs were made with each implementation subclass, starting at 5 vertexes. All runs were exported to files.

C.2 Analysing runs

With the results stored in files, the analysis was conducted with the aid of **matplotlib** to generate plots, **pandas** module and other data science related python packages. The results of said study is presented below.

IV. RESULTS

For the experimentation, the runner script was used with following parameters:

- **Starting Vertexes** = 5
- **Max Vertexes** = 24
- **Edges** = 50% of max edges.
- **Seed** = 97636

After 24 vertexes, the exhaustive algorithm takes an impractical ammount of time to compute (2^{24}), hence the value selected for the maximum number of vertexes.

A. Runs

The generated files from the computations contain the data on table I (exhaustive) and II (greedy).

B. Time Complexity

Since the time complexity of the exhaustive algorithm is an exponential function to the power of V, as the number of vertexes grows, the time to calculate the solution is exponentially increased. On the other hand,

V	E	time(s)	solution
5	5	0.00017	2
6	7	0.00051	3
7	10	0.00146	4
8	14	0.00435	5
9	18	0.01261	5
10	22	0.03470	6
11	27	0.09328	7
12	33	0.25397	8
13	39	0.65047	8
14	45	1.60149	10
15	52	4.13139	10
16	60	9.61520	11
17	68	24.64056	12
18	76	59.04742	13
19	85	137.83108	14
20	95	317.53237	14
21	105	775.19329	16
22	115	1682.81999	16
23	126	3985.63708	17
24	138	8984.20006	18

TABLE I
Exhaustive Search results

V	E	time(s)	solution
5	5	0.00054	3
6	7	0.00098	3
7	10	0.00340	5
8	14	0.00967	5
9	18	0.02433	7
10	22	0.05038	8
11	27	0.10526	9
12	33	0.18935	10
13	39	0.34459	10
14	45	0.56059	11
15	52	0.94301	12
16	60	1.54294	13
17	68	2.38707	15
18	76	3.62520	15
19	85	5.57251	18
20	95	8.44024	17
21	105	11.57873	19
22	115	15.46557	19
23	126	20.61997	19
24	138	29.35762	23

TABLE II
Greedy Aproximation Results

and although also an exponential growth, such a spike is not seen, as expected, on the greedy algorithm. Figure 2 helps us to visually comprehend the evolution of both algorithms.

Using python's **scipy**, the data was fit to a function to predict future runs. For the exhaustive solver, the runs

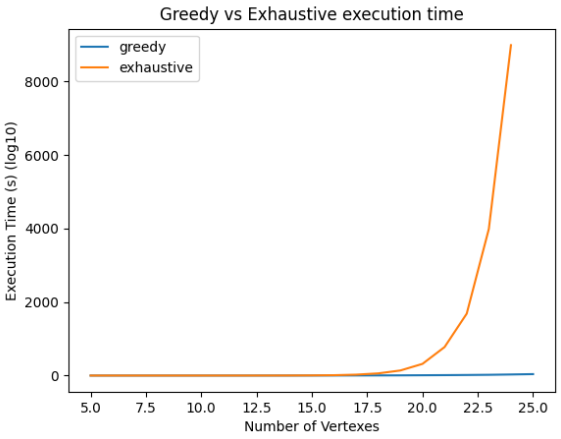

Fig. 2
Exhaustive vs Greedy execution time comparison

$$t(V) = 0.00002 * e^{0.82321*V} - 2.59394$$

Fig. 3
Exhaustive Algorithm exponential fit

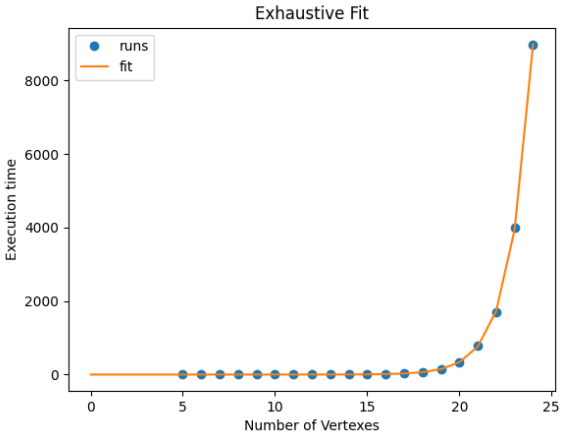

Fig. 4
Exhaustive fit vs actual data

were fit into an exponential function, which expression is presented in 3 and plotted in 4

The same procedure was applied to the greedy runs. 5 shows the function's expression and 6 its respective plot. We can easily confirm that the exponent on the greedy algorithm is much smaller than the one of the exhaustive, hence the less sharp growth.

With both these functions, we can predict the time the algorithms would take for a graph with 100 vertexes. For the exhaustive algorithm,

$$t(100) = 3.7e^{27}s$$

$$t(V) = 0.01343 * e^{0.32084*V} - 0.35222$$

Fig. 5
GREEDY ALGORITHM EXPONENTIAL FIT

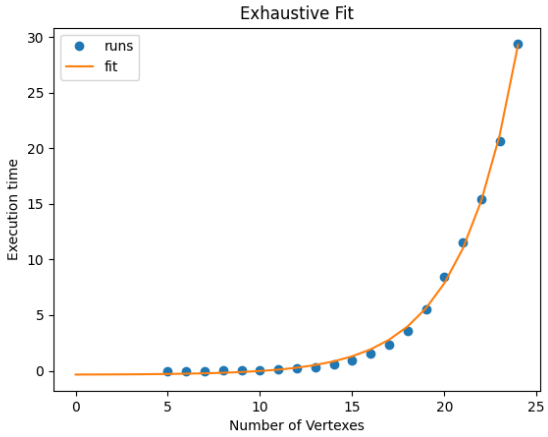

Fig. 6
GREEDY FIT VS ACTUAL DATA

V	exhaustive	greedy	absolute error
5	2	3	0.50
6	3	3	0.00
7	4	5	0.25
8	5	5	0.00
9	5	7	0.40
10	6	8	0.33
11	7	9	0.29
12	8	10	0.25
13	8	10	0.25
14	10	11	0.10
15	10	12	0.20
16	11	13	0.18
17	12	15	0.25
18	13	15	0.15
19	14	18	0.29
20	14	17	0.21
21	16	19	0.19
22	16	19	0.19
23	17	19	0.12
24	18	23	0.28

TABLE III
APPROXIMATION ERROR

And for the greedy:

$$t(100) = 320285635.94s$$

C. Approximation error

Based on the values on tables I and II, the greedy solution doesn't always compute the best possible value (the exhaustive procedure analyses every solution, so it always computes the minimum). With this in mind, the absolute error of each run was calculated, present on table III. With these values, we can infer that the algorithm implemented has an average error of **22.13** %. To reduce this error, after constructing the initial solution, a local search could be conducted in attempt to optimize the final solution.

V. CONCLUSION

With this analysis, it can be concluded that both algorithms have their pros and cons. Although in terms of computational power iterating every possible solution is very costly, if the goal is finding an optimal cover, this method should be used. Also important to note that up to a certain number of vertexes (in this particular case, around 15), the execution time between algorithms is almost negligible.

REFERENCES

[1] Yongfei Zhang et al., An Efficient Heuristic Algorithm for Solving Connected Vertex Cover Problem, Haipeng Peng, https://www.hindawi.com/journals/mpe/2018/3935804/

[2] GeeksForGeeks, *Vertex Cover Problem — Set 1 (Introduction and Approximate Algorithm)*, publicly available online, https://www.geeksforgeeks.org/vertex-cover-problem-set-1-introduction-approximate-algorithm-2/

[3] Wikipedia, *NP-completeness*, publicly available online, https://en.wikipedia.org/wiki/NP-completeness.