Minimum Vertice Dominating Set

João Tiago Lacerda Rainho 92984

***Abstract* –This report provides the insight of what is the Minimum Dominating Set of an undirected Graph and why is an important topic, how to calculate that set in an exhaustive and greedy way and the methods created to improve our perception of how the graph is represented and tested against other methods. The greedy implementation for the solution for this subject studied this paper proved to be both fast $O(nlog(n))$ and return a very accurate results (for small Graphs, large graphs could not be compared with this implementation because of its high complexity).**

***Keywords* –Minimum Dominating Set**

I. Introduction

The Minimum Dominating Set is the smallest set in which there is no vertex in the graph that is not connected to this set. This is a well known NP-Complete Problem and has many practical applications for example Social Network, protein interaction networks and ad-hoc Networks optimizations [1]. Computing this problem with an exhaustive algorithm means that only graphs with a few vertices can be exploited, this is not an optimal algorithm, for that reason a greedy algorithm is implemented in this paper which exploits probabilistic factors in graphs in order to get better results without having to test every combination.

II. Methods

A. Graph Representation

A Graph is represented by:

- **Vertices**: Set of Vertices.
- **Edges**: Dictionary with a Vertice as Key and a Set of Vertices as Value.

The vertices are represented by a Set of Vertices (*Set[Vertice]*), this is optimal because we can not have the same vertice twice inside this Set and also provides us with $O(1)$ search complexity.

The edges are represented by a Dictionary containing a Vertice as a key and a Set of Vertices as the value (*Dict[Vertice, Set[Vertice]]*). This structure also provides $O(1)$ search complexity for both the values and the keys.

B. Generating the Graphs

There is a need to test the algorithms proposed by this project, to fulfill that objective, graphs need to be generated in order to check results. To generate graphs we first need to initialize a Graph object.

Then we can use the **generate()** method which generates the graph based on some user input such as seen in I:

The graph is constructed in **two main steps** by the **generate()** method:

1. Generate the vertices
2. Generate the edges

B.1 Vertices Generation

The vertices generation is of complexity $O(n)$ because iterates though the number of vertices to generate each one individually.

There are differences in this function based on if it is needed to generate the **width**, **height** and **minimum_distance_between_vertices** (for visualization purposes). However these are negligible when both the **width** and **height** are different than *None* because each vertex must be generated based on weather there is a vertex in the same place however, when **minimum_distance_between_vertices** is different than *None* then it also must guarantee that there are not any vertices in that range but for this confirmation it is necessary to either add auxiliary vertices representing the area ocupied by each vertex, or to iterate though all vertices when trying to add another vertice, this is even worse because in the worst case, the complexity is $O((\frac{(n-1)}{2})^2)$ which simplifies to $O(n^2)$.

B.2 Edges Generation

The edges are more complex to generate than the vertices based on the argument **connect_with_closest** because if **True**, it means we need to look to the closest vertices from all the others vertices to discover which are closer. This is very helpful when debugging the complex algorithms because even for graphs with just 100 vertices it is almost impossible to figure out what vertices the edges are connecting. The implementation of the generation is done by the following pseudo-code:

The **Get_closest_vertices()** method is implemented with a **priority queue** in order to achieve approximately $O(log(n))$ complexity instead of the python **sorted()** which runs in $O(nlog(n))$ [2]. When generating large graphs, this difference is very noticeable because this function runs for every vertex.

C. Auxiliary Features

Some features were implemented in order to improve graph visualization (**visualizer.py**) and results (**results.py**).

Algorithm 1: Generating Edges

```
1  GenerateEdges $(V, C)$
       inputs : Vertices, connect_with_closest
       output: Edges
2      foreach $v_i \in Vertices$ do
3          if connect_with_closest then
4              vertices_to_connect =
                Get_closest_vertices($v_i$)
5          else
6              vertices_to_connect =
                Random($Vertices$)
7          foreach $v_2 \in vertices\_to\_connect$ do
8              AddEdge($v_1$, $v_2$)
9          end
10     end
11     return Edges;
```

C.1 Visualization

This class is used to provide a visualization method named **show()** which takes in a Graph or a list of graphs and plots them in a new window. The vertices are represented by a white circle with the name of the vertex but for easier distinction, the dominating set vertices are represented at a different color as it can be seen in 1. The **connect_with_closest** argument in the **generate()** method is important here because it makes the graph much more easy to interpret, for example even a bigger graph as in 2 which would otherwise be impossible to decode as seen in 3

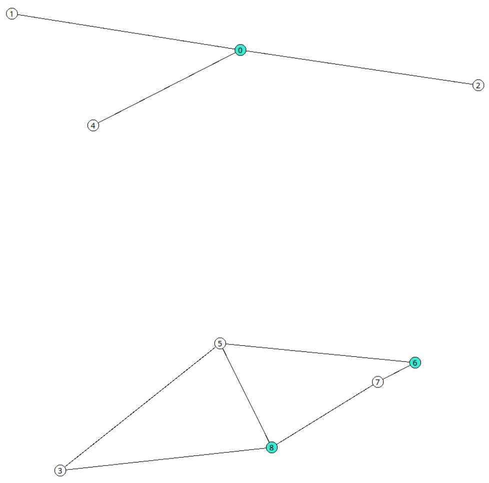

Fig. 1
Visualization example with 9 vertices and 3 max edges per vertex

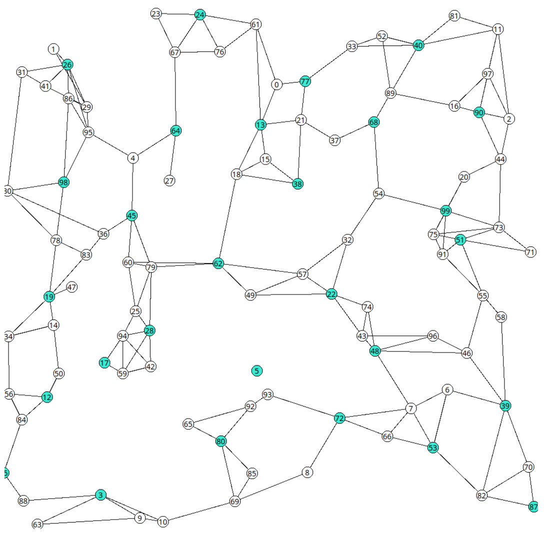

Fig. 2
Visualization example with 100 vertices and 5 max edges per vertex

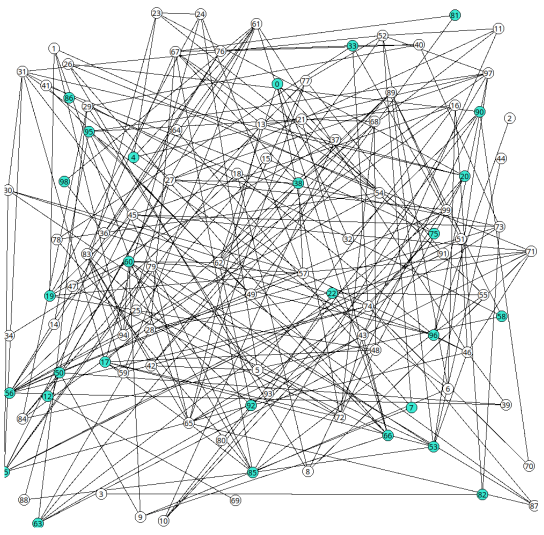

Fig. 3
Visualization example with 100 vertices and 5 max random edges per vertex

C.2 Comparison

In order to improve results extraction and to facilitate the comparison between multiple algorithms, 2 defined functions compare both the Custom and the Networkx implementation of the minimum dominating set:

1. **show_both_algorithms()**: given the inputs (*number of vertices*, *medium number of edges* and *seed*), returns the plots of both the implementations as seen in the picture 4.
2. **compare_both_algorithms_with_table()**: given a list of inputs (list of inputs of the previous function), returns a table with statistics which are used multiple times throughout this paper for example in II.

III. Results and Discussion

During the whole formal and empirical analysis the **basic operation** considered in these analysis will be the **retrieval of a vertex**.

A. Formal Analysis

In this subsection we will discuss the complexity of the algorithm by a mathematical expression.

A.1 Exhaustive Algorithm

In order to compute this problem in an exhaustive way, we need to run all possible combinations, check if that specific combination is a dominating set and then check if the new dominating set is smaller than the last one. The pseudo-code for this algorithm would be:

Algorithm 2: Exhaustive Min Dominating Set

```
1 MinDominatingSet ($G$)
      inputs : Graph
      output: Vertices
2     combinations = Combinations(Vertices($Graph$))
3     foreach $c_i \in combinations$ do
4         if $Length(c_i) < Length(min\_dominating\_set)$ then
5             if $Is\_dominating\_set(Graph, c_i)$ then
6                 min_dominating_set = $c_i$
7     end
8     return $Edges$;
```

Since we need to check every combination, the complexity of this algorithm will be

$$O(2^n)$$

However, if we count the **Is_domination_set()** method, because it is of complexity $O(n \times medium\_number\_of\_edges)$, we end up with a larger complexity: $O(2^n \times n \times medium\_number\_of\_edges)$ but since the smaller order terms are not taken into account, for that reason, from now on these will not appear again.

A.2 Greedy Heuristics

The greedy approache consists in 3 main steps:

1. starting the min dominating set with the vertices that contain 0 edges and if only contains 1 edge, then add the vertice which is connected
2. sort vertices from the number of edges it contains and start adding them to the dominating set until finds a dominating set
3. remove excess vertices from the dominating set by sorting them in decreasing order of number of edges and trying to check if the dominating set remains a dominating set without that vertice

The preprocessing complexity is $O(n)$ and can be seen as

Algorithm 3: Preprocessing Min Dominating Set

```
1  Preprocessing ($G$)
       inputs : Graph
       output: Vertices
2      vertices = Set()
3      foreach $v_i \in Vertices(Graph)$ do
4          vertices = Edges($Graph$, $v_i$)
5          if $Length(vertices) == 0$ then
6              vertices.add($v_i$)
7          else if $Length(vertices) == 1$ then
8              vertices.add(Pop(vertices))
9      end
10     return $vertices$;
```

In order to get a dominating set, it is more likely that highly linked vertices are more important as a member of the dominating set so that a larger number of other vertices can be included and therefore expand even more the set of included vertices. Having the previous said in mind, in order to find the dominating set we should start by ordering the vertices (which is $O(nlog(n))$) and then add the vertex and their adjacent vertices, following the block of pseudocode we can check that has a $O(n)$ complexity resulting in $O(2nlog(n))$ which simplifies to

$$O(nlog(n))$$

Algorithm 4: Calculating Min Dominating Set

```
1  MinDominationSet ($G, V$)
       inputs : Graph, Vertices
       output: Vertices
2      covered = Set()
3      domination_set = Set()
4      invert_sorted_vertices = Sort(Vertices(Graph))
5      foreach $v_i \in (invert\_sorted\_vertices)$ do
6          if $v_i \in covered$ then
7              continue
8          Add($covered$, Edges($Graph$, $v_i$))
9          Add($domination\_set$, $v_i$)
10         if $IsDominatingSet(Graph, domination\_set)$ then
11             break
12     end
13     return $domination\_set$;
```

And finally remove redundant nodes by ordering the resulting dominating set with $O(dominating\_set \times log(dominating\_set))$ complexity and $O(dominating\_set)$ complexity to remove the vertices which simplifies to

$$O(dominating\_set \times log(dominating\_set))$$

Algorithm 5: Optimize Min Dominating Set

```
1 RemoveRedundantVertices ($G, V$)
      inputs : Graph, dominating_set
      output: Vertices
2     sorted_dominating_set = Sort(dominating_set)
3     foreach $v_i \in$ sorted_dominating_set) do
4         Remove(sorted_dominating_set, $v_i$) if
            IsNotDominatingSet(Graph,
            domination_set) then
5             Add(domination_set, $v_i$)
6     end
7     return domination_set;
```

Therefore, the sum of each component complexity of this algorithm is $O(n + n \times log(n) + dominating\_set \times log(dominating\_set))$ which simplifies to

$$O(n \times log(n))$$

B. Experimental Results

All results are calculated using $seed = 100$.

B.1 Exhaustive Search

The exhaustive algorithm does not allow us to test the algorithm much larger than 22 vertices. This is explained by the formal analysis of the exhaustive search. The table capturing the extension of the time complexity can be found in II and the basic operation in III.

We can see as expected that the number of vertices returned by the exhaustive search is smaller than the Networkx version, however the elapsed time is much larger even for small increases in the size of the graph, for example in the last row, the increase in vertices was 10% and edges 11.8%, however, the elapsed time increased by 331.6

The exhaustive algorithm is therefore concluded experimentally to be exponential while both the Greedy and Networkx can only be concluded that they are not exponential.

B.2 Greedy Heuristics

The greedy algorithm has a much better ratio between being computationally fast and having a good solution, remaining always better than the Networkx however, being always slower as seen in IV.

When not applying the pós-processing we get worse but faster results as expected.

The table with the basic operations is presented in V.

IV. Conclusion

In conclusion the minimum dominating set greedy algorithm implemented in this paper while it is not as fast as the dominating set function of the Networkx library, it is still very fast and provides much better results while still providing $O(nlog(n))$ complexity. This can be used when trying to achieve a better solution when time is not a massive factor.

References

[1] Minh Hai Nguyen, Minh Hoàng Hà, Diep N. Nguyen, and The Trung Tran, "Solving the k-dominating set problem on very large-scale networks", *Computational Social Networks*, vol. 7, no. 1, pp. 4, Dec. 2020. **URL:** `https://computationalsocialnetworks.springeropen.com/articles/10.1186/s40649-020-00078-5`

[2] "Timsort", Dec. 2021, Page Version ID: 1058845510. **URL:** `https://en.wikipedia.org/w/index.php?title=Timsort&oldid=1058845510`

Appendix

Argument	Type	Function
number_of_vertices	Integer	Number of vertices inside the graph
medium_number_of_edges	Integer	Medium number of edges connecting each vertice
width	Integer	Width of the canvas where the graph is represented
height	Integer	Height of the canvas where the graph is represented
minimum_distance_between_vertices	Integer	Mininum distance between each vertice
connect_with_closest	Boolean	Connect each vertice with the **n** closest vertices
verbose	Boolean	Prints information about the operation

TABLE I
Generate Interface

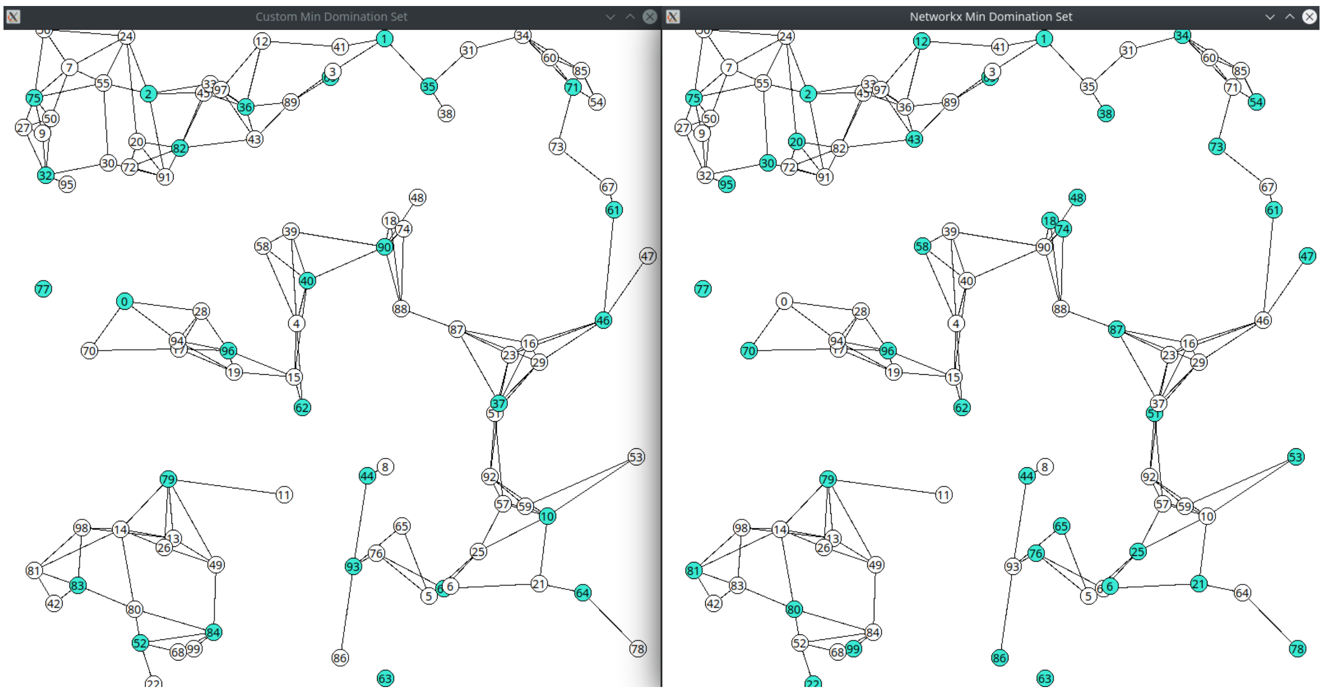

Fig. 4
Visual algorithms comparison

Vertices	Edges	Exhaustive	Greedy	Networkx
4	2	2 vertices 0.04 ms	2 vertices 0.03 ms	3 vertices 0.02 ms
8 + 100.0%	6 + 200.0%	4 vertices 0.76 ms + 1961.2%	4 vertices 0.04 ms + 34.2%	5 vertices 0.02 ms - 18.2%
12 + 50.0%	11 + 83.3%	5 vertices 12.59 ms + 1565.8%	5 vertices 0.04 ms + 0.2%	7 vertices 0.03 ms + 36.4%
16 + 33.3%	11 + 0.0%	9 vertices 235.21 ms + 1767.9%	9 vertices 0.07 ms + 88.1%	10 vertices 0.03 ms + 15.5%
20 + 25.0%	17 + 54.5%	9 vertices 4962.32 ms + 2009.8%	9 vertices 0.09 ms + 23.1%	12 vertices 0.03 ms + 13.5%
22 + 10.0%	19 + 11.8%	11 vertices 21415.71 ms + 331.6%	11 vertices 0.12 ms + 37.8%	14 vertices 0.04 ms + 4.5%

TABLE II
Time comparison of all the algorithms for small graphs

Vertices	Edges	Exhaustive	Greedy w pós-processing	Greedy w/out pós-processing
4	2	2 vertices 15 b/o	2 vertices 23 b/o	3 vertices 16 b/o
8 + 100.0%	6 + 200.0%	4 vertices 255 b/o + 1600.0%	4 vertices 54 b/o + 134.8%	5 vertices 38 b/o + 137.5%
12 + 50.0%	11 + 83.3%	5 vertices 4095 b/o + 1505.9%	5 vertices 75 b/o + 38.9%	5 vertices 59 b/o + 55.3%
16 + 33.3%	11 + 0.0%	9 vertices 65535 b/o + 1500.4%	9 vertices 131 b/o + 74.7%	10 vertices 88 b/o + 49.1%
20 + 25.0%	17 + 54.5%	9 vertices 1048575 b/o + 1500.0%	9 vertices 172 b/o + 31.3%	11 vertices 123 b/o + 39.8%
22 + 10.0%	19 + 11.8%	11 vertices 4194303 b/o + 300.0%	11 vertices 183 b/o + 6.4%	11 vertices 134 b/o + 8.9%

TABLE III
Basic operations comparison of all the algorithms for small graphs

Vertices	Edges	Greedy w pós-processing	Greedy w/out pós-processing	Networkx
125	289	38 vertices 1.41 ms	45 vertices 0.57 ms	49 vertices 0.11 ms
250 + 100.0%	622 + 115.2%	74 vertices 5.66 ms + 299.9%	84 vertices 1.92 ms + 236.8%	95 vertices 0.17 ms + 58.5%
500 + 100.0%	1274 + 104.8%	139 vertices 22.35 ms + 295.2%	167 vertices 7.67 ms + 299.6%	188 vertices 0.35 ms + 98.3%
1000 + 100.0%	2562 + 101.1%	290 vertices 90.96 ms + 306.9%	328 vertices 32.63 ms + 325.4%	353 vertices 0.64 ms + 84.1%
2000 + 100.0%	5063 + 97.6%	570 vertices 345.44 ms + 279.8%	647 vertices 130.22 ms + 299.0%	744 vertices 1.32 ms + 108.1%
4000 + 100.0%	10020 + 97.9%	1147 vertices 1428.48 ms + 313.5%	1303 vertices 513.64 ms + 294.4%	1482 vertices 2.88 ms + 117.3%
8000 + 100.0%	20063 + 100.2%	2319 vertices 6024.52 ms + 321.7%	2671 vertices 2164.56 ms + 321.4%	2958 vertices 5.84 ms + 102.8%

TABLE IV
Time comparison of the greedy algorithms for large graphs

Vertices	Edges	Greedy w pós-processing	Greedy w/out pós-processing
125	289	38 vertices 1410 b/o	45 vertices 1118 b/o
250 + 100.0%	622 + 113.5%	74 vertices 3104 b/o + 120.1%	84 vertices 2484 b/o + 122.2%
500 + 100.0%	2551 + 104.6%	139 vertices 6873 b/o + 121.4%	167 vertices 5473 b/o + 120.3%
1000 + 100.0%	1274 + 100.9%	290 vertices 15019 b/o + 118.5%	328 vertices 11950 b/o + 118.3%
2000 + 100.0%	5063 + 97.6%	570 vertices 32628 b/o + 117.2%	647 vertices 25905 b/o + 116.8%
4000 + 100.0%	10020 + 97.9%	1147 vertices 70594 b/o + 116.4%	1303 vertices 55809 b/o + 115.44%
8000 + 100.0%	20063 + 100.2%	2319 vertices 152698 b/o + 116.3%	2671 vertices 119623 b/o + 114.34%

TABLE V
Basic operations comparison of the greedy algorithms for large graphs