Solving the Minimum-Weight Dominating Set Problem with Different Algorithms and Heuristics

Leandro Silva, 93446
leandrosilva12@ua.pt

***Abstract* –This main objective of this article is to analyse the Minimum-Weight Dominating Set Problem with different algorithms and for different sizes of the problem. Formal analysis of the computational complexity of each algorithm are made, and complemented with an experimental analysis based on their execution times and number of basic operations. There is also a helpful visualization of the given solution of the graphs under analysis.**

***Keywords* –Minimum, Weight, Graph, Dominating Set, Algorithm, Search, Complexity, Exhaustive, Greedy, A-Star, Heuristic, Bisect, Heap Queue, Bucket Queue**

I. INTRODUCTION

This problem was first studied in the 1950s, and since then, the fastest algorithm found has $O(1.5048^n)$ time complexity, due to the combination of two central techniques from the field of exponential time algorithms: inclusion/exclusion and branching with measure and conquer analysis. [2]

Dominating sets are of practical interest in several areas. In wireless networking, dominating sets are used to find efficient routes within ad-hoc mobile networks. [1]

This article will conduct an analysis to a few solutions to this problem as a contemplation for the first project of the course *Algoritmos Avançados*.

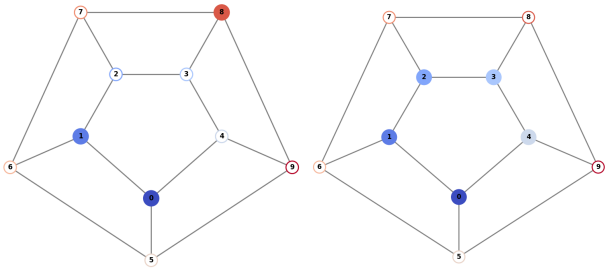
The results were obtained using a program made in python, which, despite being slower than other languages, is a simple language for making investigation in such topics.

II. DESCRIPTION OF THE PROBLEM

A. Minimum-Weight Dominating Set

A dominating set for a graph G = (V, E) is a subset D of V such that every vertex not in D is adjacent to at least one member of D. Then, the minimum-weight dominating set is the dominating set that has the lowest total weight sum, although in some variants it may be the set with fewer number of vertices. [1] The whole graph is also considered as a subset of itself, as it obeys the rules described above, it is also considered a dominating set. Therefore, there will be always at least this subset as a solution to any graph problem, which is the worst case solution.

Figures 1a and 1a show two examples of dominating sets for a graph with a visual format that will be used in further analysis throughout the article. In each example, each outlined vertex is adjacent to at least one filled vertex, and it is said that the outlined vertex is dominated by the filled vertex. The domination number of this graph is 2: the examples (b) and (c) show that there is a dominating set with 2 vertices, and it can be verified that there is no dominating set with a number of vertices fewer than 3. The colours used for each vertex result in a colour map scale to facilitate the visual representation of the vertex weight (which is used as its label). Vertices whose colours are closer to blue have lower weights, whereas vertices with colours closer to red have higher weights.



(a) Solution 1 (b) Solution 2

Fig. 1: The visualization of two different dominating sets for the same problem

B. Graph Data Structures

An **adjacency matrix** [3] was considered to represent the graphs, but eventually discarded due to its inferior efficiency to other data structures. The problem faced required the answer to two questions: what is the weight of one vertex; and what are the vertices connected to one vertex. In an adjacency matrix, the answer to the first question cannot be easily done, and the answer to the second question would take linear time, as it would have to traverse the entire row of one vertex to find its neighbours. An **incidence matrix** [4] is also off-topic, since it reckons edges, which are irrelevant to our problem.

With this in mind, the data structures used to represent the graphs were two **dictionaries**, one to store the weight of a vertex, and another to store the set of neighbours of a vertex. Likewise, both questions can be answered in $O(1)$ time.

III. DESCRIPTION OF THE SOLUTIONS

In this problem, there were tested a few solutions. Even though their approaches are different, they follow the same line of thought to solve the problem: get the next candidate subset of the graph, check if it is a dominating

set, and update the global minimum set found according to the candidate's total weight sum. The different resides in the order they pick the next subset candidates, which relies on the way the search traversal is made and/or on heuristics.

TABLE I: Sequence of candidate subsets for different algorithms

Iteration / Basic Op.	Algorithm Sequence		
	Exhaustive / Branch-and-Bound	Greedy	A-Star
1	1	1	1
2	2	12	2
3	3	123	12
4	12	13	13
5	13	2	3
6	23	23	123
7	123	3	23

Supposing there is an ordered list of vertices according to some rules or heuristics, say [1 ... 3], the order from which they chose the next subset for the next iteration is demonstrated in Tab. I. So, for the exhaustive search, the candidate subset on the 4th iteration is the set with the 1st and 2nd vertex from the list of vertices. For each algorithm, a more in depth analysis is made in the following respective section.

A. Exhaustive Algorithm

The exhaustive search can also be labeled as a brute-force search. It iterates through all possible combinations of subsets in a graph, regardless of its order, and tests if they are dominating sets. If so, it calculates its total weight sum and updates its current solution, if required.

Its complexity is easy to come up with, it is the number of subsets in a graph, which can be formulated into $2^n - 1$. Thus, its time complexity is $O(2^n)$.

Tab. I shows an order of the candidates sequence, but it really does not matter. It has this aspect because it is in fact the way that the developed program iterates as, yet it is only like this as a convenience for the Branch-and-Bound algorithm (see Sec. Branch-and-Bound Algorithm).

B. Branch-and-Bound Algorithm

With a small improvement on the brute-force search, we can reduce significantly its complexity. The branch-and-bound algorithm follows this idea. The difference from the exhaustive algorithm is that it has an additional condition to stop the search when there is certainty that the search reached a global minimum. For this reason, the code method of the exhaustive was reused to this algorithm.

The method used to know when the algorithm can stop is straight forward. This algorithm generates the sequence of candidate solutions like the exhaustive search: first iterate through all subsets with only 1 vertex, then iterate through all with 2 vertices, and so on (check Tab. I). As it is straightforward to get the minimum-weight subset cost with n vertices, and as the minimum cost increases with the number of vertices, it is reasonable to stop the algorithm if the next subsets with n vertices have a minimum cost higher than the solution cost already found in all subsets with size lower than n.

The worst case complexity of Branch-and-Bound remains the same as that of the brute-force, because in the worst case, we may never get a chance to stop the algorithm. However, in practice, it performs very well depending on the minimum size of an existent dominating set.

C. Greedy Algorithm

The greedy algorithm chooses candidate sets according to one rule: at each iteration, add the next minimum weighted vertex to the previous candidate subset, which is demonstrated in Tab. I. This method is implemented in linear time, as, in the worst case, the algorithm will return the whole graph as a solution. Therefore, the number of basic operations is lower than the number of vertices in the graph, which can be translated into a $O(n)$ time complexity.

The simplicity and the efficiency of this algorithm pushes down its effectiveness in finding a good solution.

D. A-Star Algorithm

This algorithm can be seen as a variant of the Greedy algorithm, with the exception that it has in consideration, not only the next vertex, but also the previous vertices chosen. To assist in this, this algorithm normally uses a **bucket queue**. A bucket queue is a data structure that maintains a dynamic collection of elements with numerical priorities and allows quick access to the element with minimum (or maximum) priority. [5] Translated into our program, the A-Star Algorithm will keep a list of candidate nodes visited and continue the search on the nodes with higher priority (lower cost).

Using the sequence example from Tab. Sequence of candidate subsets for different algorithms, the bucket queue would have in the beginning the following nodes:

[1, 2, 3]

Then, the node with higher priority, 1, would be popped out of the queue, undergo in a dominating set test, and after used to get more candidates, resulting in the following state:

[2, 3, 12, 13]

This queue would then have to be sorted according to the node's total weight sum. Assuming that the vertices have the same cost as its label ID, the queue would remain unchanged, and the cycle would continue on.

This results in perfect sequence where ordered according to the total weight sum of a subset, which means that this algorithm can stop on the first candidate that passes the dominating set test.

The complexity of the A-Star algorithm has to have in count the complexity of the python's sort method. On the average and worst case, the sort method has $O(n\ log\ n)$ time complexity. [6] For the best case scenario, when the list is already sorted (which is almost the case for every

time it is sorted by this algorithm), its complexity is reduced to $O(n)$. Therefore, considering k to be an approximation to the average bucket queue size, the A-Star algorithm would have a time complexity of $O(2^n\ k)$ for the worst case. It might seem an awful complexity comparing to the exhaustive search, but there must be the consideration that this is the worst case. In reality, this algorithm can perform better than the exhaustive search, depending on the problem (see Sec. Solution Costs).

D.1 Heap Queue

During the implementation of the A-Star algorithm, it was very noticeable the huge time complexity of this algorithm. However, there were still some optimizations that could be performed. The bucket list was being sorted every time without necessity, as all its members were already sorted, except for the new ones. For this reason, methods that can maintain a list in sorted order without having to sort the list after each insertion are highly rewarding.

Two python modules that had followed this idea were investigated, the **bisect** and the **heapq**.

The bisect module has a disadvantaged compared to the other: the $insort()$ functions are $O(n)$ because the logarithmic search step is dominated by the linear time insertion step. [8] The heapq module does not have this disadvantage. A heap is a binary tree in which every parent node has a value less than or equal to any of its children. This implementation uses arrays for which $arr[k] <= arr[2 * k + 1]$ and $arr[k] <= arr[2 * k + 2]$ for all k elements in the list's size range. An interesting property of a heap is that its smallest element is always the root, $arr[0]$. [7]

IV. RESULTS AND DISCUSSION

In order to analyse the performance of the algorithms implements, their number of basic operations, execution times, and solution costs are all used for comparison. The results shown were obtained in randomized graphs using the seed 93446.

A. Computational Complexity

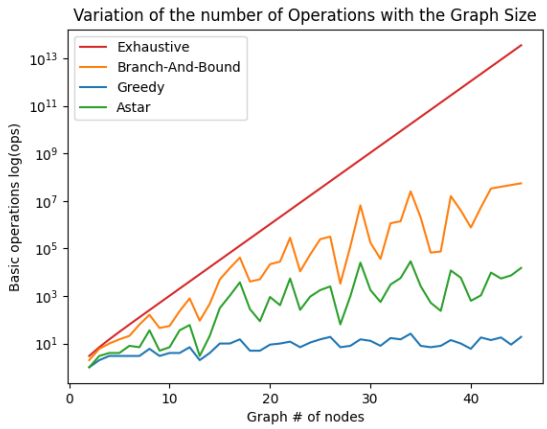


Fig. 2: Variation of the number of Basic Operations

When comparing all algorithms' number of basic operations per graph size, Fig. 2, there is a characteristic that is immediately spotted: the perfectly linear red line from the **exhaustive algorithm**. This is due to the fact that this algorithm has an exact time complexity of $O(2^n)$, has shown in Sec. Exhaustive Algorithm. It is important to notice that this graph as a **logarithmic scale** on the y-axis, to better display the wide range of data.

The **branch-and-bound** algorithm, has improved a lot the exhaustive search, although it is hard to predict its performance, as shown on its irregular yellow line. The reason for this matches with the explanation in Sec. Branch-and-Bound Algorithm.

As the first solution of A-Star algorithm is always the best solution, it is reasonable to conclude that the steps required to reach this solution are always lesser than branch-and-bound. This is why there is a similar green line to the yellow line that is always under the latter.

The greedy algorithm seems to relatively maintain a constant number of basic operations, yet this is not true. For a higher number of nodes, the increase of the basic number of operations would be more noticeable. The logarithm scale, alongside with some favourable problems, may lead to this misleading analysis.

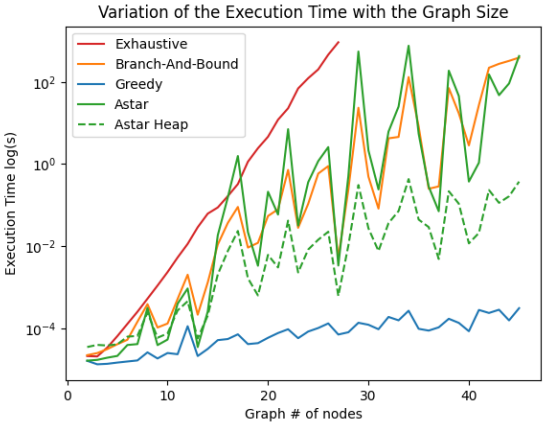


Fig. 3: Variation of the Execution Times

Switching to the execution time comparison, we can see the similarity on the relation with the number of basic operations. The only exception is the A-Star line. This is not strange at all, since the operations required to sort the bucket queue are not taken into account. It can be seen that the a-star almost always surpasses the branch-and-bound search time, and can even have higher execution times than the exhaustive search. In contrast, the performance of the a-start optimized with a heap queue is noteworthy. The basic operations remained unchanged, but the execution times decreased significantly, leading to results always better than the branch-and-bound method.

B. Solution Costs

In Fig. 4, the solution costs of the greedy approach is compared to the best and worst case scenario. The remaining algorithms always give the best solution, represented in the green line, whereas the greedy algorithm have the best or the slightly best cost. Sometimes it can even output an exaggerated cost for the problem, as shown in the blue

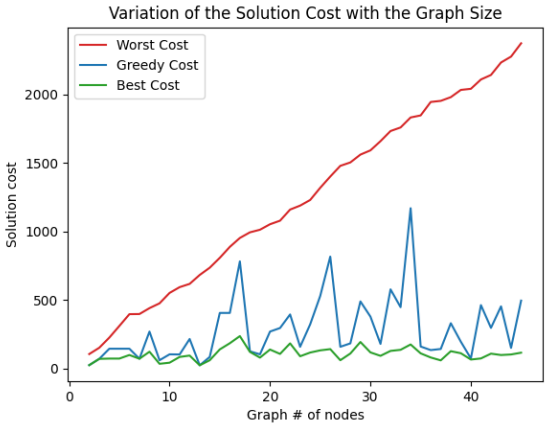


Fig. 4: Relation of the Greedy Solution Costs with the bound solution costs

line spikes. Nevertheless, its linear time complexity justifies the lack of effectiveness for the larger the problem is.

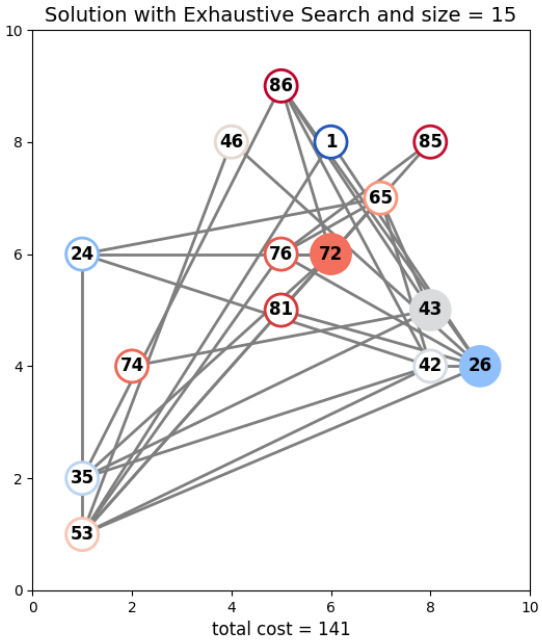


Fig. 5: Solution given by the Greedy Search on a Graph with size = 15

For the problem with $size = 15$, the solutions of the exhaustive and greedy approach are shown in Figs. 5 and 6, respectively. This problem was specifically chosen for the poor effectiveness of the greedy approach, that gave a solution with a total cost higher than 2 times the cost of the best solution. The problem resides in the fact that the greedy approach reckons the vertex with weight 72 and coordinates (6, 6) too late, for being considered a bad vertex according to its heuristic.

V. CONCLUSION

To conclude this article, I would want to remark how the formal analysis of the computational complexity of each

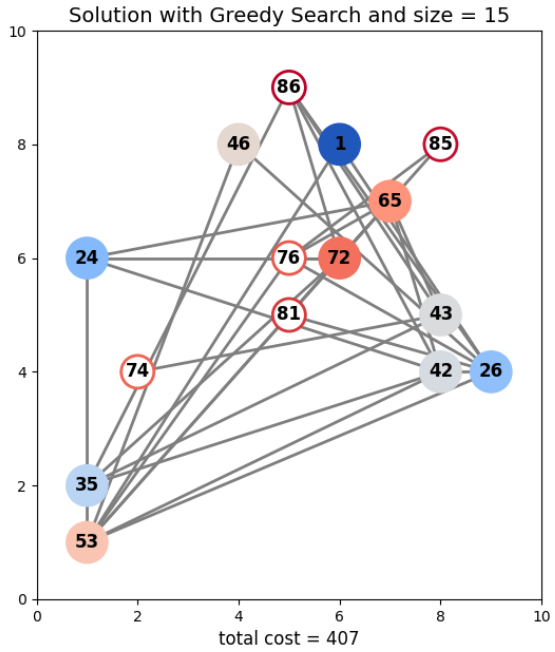


Fig. 6: Solution given by the Greedy Search on a Graph with size = 15

algorithm were coincident with the experimental analysis, which helped me better understand the investigated algorithms and the reasoning of its performances.

For the algorithms based in heuristics, such as the greedy and a-star, the heuristic used relied on the weight of the vertices. However, as future work, I think it would be a good idea to examine different approaches to the heuristic, or even additional conditions that aim to improve the effectiveness (in the greedy case) or the efficiency (in the a-star case) in finding the solution.

REFERENCES

[1] Dominating set. (2021, November 27). In *Wikipedia*. https://en.wikipedia.org/wiki/Dominating_set

[2] van Rooij, J. M. M., Nederlof, J., van Dijk, T. C. (2009). Inclusion/Exclusion Meets Measure and Conquer. *Lecture Notes in Computer Science*, 554–565. https://doi.org/10.1007/978-3-642-04128-0_50

[3] Adjacency matrix. (2021, October 4). In *Wikipedia*. https://en.wikipedia.org/wiki/Adjacency_matrix

[4] Adjacency matrix. (2021, November 2). In *Wikipedia*. https://en.wikipedia.org/wiki/Incidence_matrix

[5] Bucket queue. (2021, August 7). In *Wikipedia*. https://en.wikipedia.org/wiki/Bucket_queue

[6] *TimeComplexity - Python Wiki*. (2020, August 18). Python - Time Complexity. https://wiki.python.org/moin/TimeComplexity

[7] *heapq — Heap queue algorithm — Python 3.10.0 documentation*. (2021, December 4). Heapq — Heap Queue Algorithm. https://docs.python.org/3/library/heapq.html

[8] *bisect — Array bisection algorithm — Python 3.10.0 documentation*. (2021, December 4). Bisect — Array Bisection Algorithm. https://docs.python.org/3/library/bisect.html