Decision problem: Edge dominating set of a graph with k edges

Bruna de Melo Simões
103453 - brunams21@ua.pt

***Abstract* – This paper presents an overview of the edge dominating set problem of a graph, that is, the problem finding an edge dominating set with size k of a graph. An edge dominating set of a graph G consists of a set of edges D, such that every edge not in D is adjacent to at least one edge in D. We address this problem using two different approaches, one which considers every solution that there is, using an exhaustive search, and another that implements an heuristic, more specifically, a greedy algorithm, which tries to find a solution given certain criteria.**

***Resumo* – Este artigo apresenta uma visão geral do problema do conjunto dominante de arestas de um grafo, ou seja, o problema de encontrar um conjunto dominante de arestas com tamanho k de um grafo. Um conjunto dominante de arestas de um grafo G consiste num conjunto de arestas D, tal que cada aresta que não esteja em D seja adjacente a pelo menos uma aresta em D. Este problema foi abordado usando duas metodologias diferentes, uma que considera cada solução que existe, utilizando um algoritmo de procura exaustiva, e outro que implementa uma heurística, mais especificamente, um algoritmo voraz, que tenta encontrar uma solução dados determinados critérios.**

***Keywords* – k-edge dominating set, brute-force algorithm, exhaustive approach, heuristic approach, greedy algorithm**

***Palavras chave* – conjunto dominante de k arestas, algoritmos de força bruta, abordagem exaustiva, abordagem heurística, algoritmos vorazes**

I. Introduction

Given a graph G, and a subset D, an edge dominating graph of G is one such that every edge not in D is adjacent to at least, one edge in D. In other words, every edge in the graph is either part of the edge dominating set or adjacent to an edge in it.

Using a more formal mathematical approach, we can define the problem as such: let $G = (V, E)$ be an undirected graph. A set $D \subseteq E$ is an edge dominating set if, for every edge e, either $e \in D$ or there exists an edge $f \in D$ such that e and f share a common vertex. [1]

Finding whether a graph has a dominating set with a certain size is an NP-hard problem, that is, a decision problem, which can be reduced to polynomial time. [2]

In order to find the edge dominating set of a graph, two main implementations were developed: an exhaustive search, using brute-force, which always finds all possible solutions to a fixed value of k by searching thoroughly the given graph, and an greedy search, which uses an heuristic approach to find the most suitable edges to consider in the search of a solution.

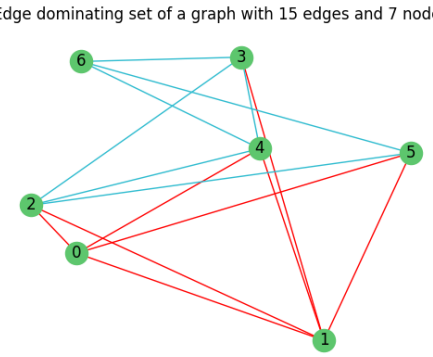

Fig. 1 - Edge dominating set of a graph with 7 nodes and 15 edges, where the edges in the dominating set, with $k = 8$, are shown with the red color

II. Exhaustive Search

A. Algorithm Analysis

The first approach to this problem was implementing an algorithm that would search for every possible solution to the problem exhaustively, using a brute-force approach and, therefore, iterating through all possibilities and checking if each respects the edge dominating set criteria. [3]

Therefore, in order to better analyse the results and test their accuracy, this algorithm was implemented in Python language:

```
def exhaustive_search(graph, k):
    solutions = []
    edges = list(graph.edges)

    for subset in itertools.combinations(edges
        , k):
        is_dominating = True
        for edge in graph.edges:
            if not any(is_adjacent(edge, e)
                for e in subset):
                is_dominating = False
                break
        if is_dominating:
            solutions.append(subset)

    return solutions
```

```
def is_adjacent(edge1, edge2):
    return len(set(edge1) & set(edge2)) > 0
```

This algorithm starts by going through each possible combination of edges of size k. Then, it checks if the subset is a dominating set or not, by going through each graph edge. If there isn't any edge who is adjacent to one edge in the subset, then it does not consist of one dominating set, and therefore, it does not consist of a solution.

The auxiliary `is_adjacent(edge1, edge2)` function checks if two edges are connected through a node, by checking if there is a connection between them or not.

All solutions are, upon discovery, added to a list. Therefore, the `solutions` list consists of a set of all possible solutions in which each one of them are of size k.

B. Time Complexity Analysis

The algorithm start by using the *itertools* library to iterate through each combination of edges of size k. Considering a graph with n number of graph edges and k the size of the dominating sets, the number of combinations is given by the formula $\frac{n!}{k!(n-k)!}$ Therefore, the complexity associated with these iterations is:

$$O\left(\frac{n!}{k!(n-k)!}\right) = O\left(\binom{n}{k}\right)$$

Next, its checked whether the iterating set is an edge-dominating one or not. In order to do that, all edges in the subset are iterated through and checked if they are adjacent to edges in the graph. In order to do so, we go through each edge in the searched edge subset of size k.

The loop that checks if the subset is a dominant one iterates through every graph edge, therefore, has a time complexity $O(n)$

Therefore, the overall time complexity can be reduced to:

$$O\left(\binom{n}{x} \times n\right)$$

We can conclude, therefore, the exponential nature of this algorithm, and confirm its inefficiency for larger problems with a greater number of edges, and for a greater value of k, since they would require great computational effort. However, this algorithm is useful in small problems, since it always delivers all correct solutions for a given graph and k value.

However, the problem of finding an edge-dominating set using less computational effort and increase its speed in finding a solution is still present. Therefore, another approach was implemented to deal with larger graphs and also greater k values.

III. HEURISTIC IMPLEMENTATION

The heuristic implementation is an implementation of an algorithm that tries to find a solution based on the graph's characteristics, namely, the edge density in a vertex.

A. Algorithm Analysis

```
def greedy_heuristic(G, k):

    dominating_set = set()
    covered_edges = set()

    while len(dominating_set) < k:

        edge_candidates = []

        for edge in G.edges():
            if edge not in dominating_set:
                adjacent_edges = set(G.edges(
                    edge[0])) | set(G.edges(
                    edge[1]))
                new_coverage = len(
                    adjacent_edges -
                    covered_edges)
                remaining = k - len(
                    dominating_set)
                efficiency = min(new_coverage,
                     remaining) / len(
                    adjacent_edges)
                edge_candidates.append((edge,
                    efficiency))

        if not edge_candidates:
            break

        edge_candidates.sort(key=lambda x: x
            [1], reverse=True)
        best_candidate = edge_candidates[0][0]

        dominating_set.add(best_candidate)
        covered_edges.update(G.edges(
            best_candidate[0]))
        covered_edges.update(G.edges(
            best_candidate[1]))

    return list(dominating_set) if len(
        dominating_set) <= k else None
```

This greedy implementation, contrary to the exhaustive search, does not try to find all edge-dominating sets in a graph. It implements an heuristic, that is, a simple strategy to try and find a solution. Therefore, it only looks for one solution, and returns it, if it exists.

However, it won't be able to return more than one solution to the problem, and tries to answer the problem of finding out if a graph has an edge-dominating set of size k or not. Additionally, the implemented heuristic might return a wrong answer, namely, for a given graph, returning no solutions when in reality there is one, but was not detected by the heuristic algorithm.

The algorithm firstly initializes the possible solution as empty, and while the solution, which contains a set of edges, has lesser size than k, iterates continuously until no more edge candidates are detected. The edge candidates are edges that have a high efficiency, more specifically, edges that are connected to vertexes with a larger edge density. This means that these edges are the ones which are connected to vertex which in turn have more edges connected to themselves, and therefore, are the best candidates to start looking for edge-dominating sets within the graph.

Simultaneously, edges that already have been explored are stored in a list in order for the algorithm to not analyse them again and create a loop within itself.

This algorithm, then, stops when the edge dominating set with size k has been found. When the list of solu-

tions reaches the size k, it returns this solution. Therefore, since the algorithm relies always on the graph's constitution and edge density, the solution presented is always the same, no matter how many times this algorithm is performed.

B. Time Complexity Analysis

The algorithm starts by repeating, in the worse case scenario, k times. This happens because upon the dominating set reaching size equal to the value k, the iteration ends, since each iteration supposedly adds one element to the dominating set. Therefore, the time complexity for this loop is $O(k)$.

Next, the algorithm iterates through each edge in the graph, and getting the best edge candidates. The operations to get the best edge candidates have a constant time complexity, $O(1)$, and therefore, are not considered in the end for the time complexity of the greedy algorithm.

Finally, the candidate edges are sorted based on their efficiency, that is, their density. This means that the best candidate edges are the ones attached to vertexes which in turn have a high number of edges attached to themselves. The default sorting mechanism in Python has a recorded time complexity of $O(n \log n)$, n being the amount of items to sort.

Therefore, we can conclude that the time complexity of this algorithm is defined as such:

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

In this notation, k corresponds to the size of the edge-dominating set we pretend to search, n is the number of graph edges and c is the list size of found edge candidates, which need to be sorted by efficiency.

IV. Result Analysis

Due to the great time complexity of the exhaustive search algorithm, and posterior time complexity analysis, it was possible to deduce that this algorithm would not be able to find efficiently edge dominating sets of size k of large graphs.

In order to infer about how time increases between nodes, some experiments were performed which help visualize the time exponentially increase with the number of nodes.

In Fig. 2, we can visualize the number of nodes related to the amount of time the greedy algorithm took to collect all possible solutions. In this experiment, different k values were calculated based on percentages of graph edges. For example, if a graph had 16 edges, the algorithm should look for the edge dominating sets of size 2, 4, 8 and 12, which correspond, respectively, to 12.5%, 25%, 50% and 75% of graph edges.

Even though the carried experiments for the exhaustive approach were only performed with four nodes, it is visible the amount of time it takes to find the solutions increases exponentially.

It is also visible that the k variable, in which a higher percentage corresponds to a greater size of k, and

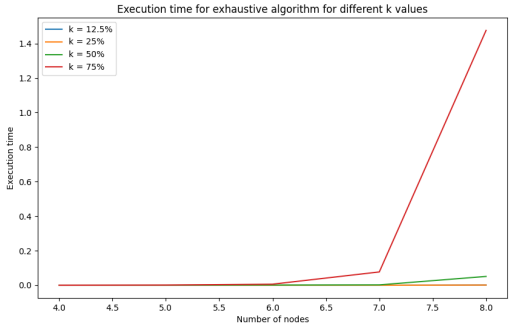

Fig. 2 - Number of nodes related to the amount of time needed to find an edge-dominating set of size k of a graph using a exhaustive approach

therefore, the number of edges in the edge dominating set increases in turn, translates in a higher processing time. This is to be expected, since the analysed time complexity of the exhaustive algorithm depends heavily on the number of combinations between k and the number of edges of the graph.

It is, therefore, expected an exponential increase in the time required to process the implemented algorithm with greater values of k.

As expected, the number of nodes also takes a role in the amount of time required to calculate the solutions, which is also to be expected, since in this project the amount of edges is calculated based on the number of nodes. In this Fig. 2, the analysed data uses a graph in which 75% of the maximum number of edges was used for the number of vertices.

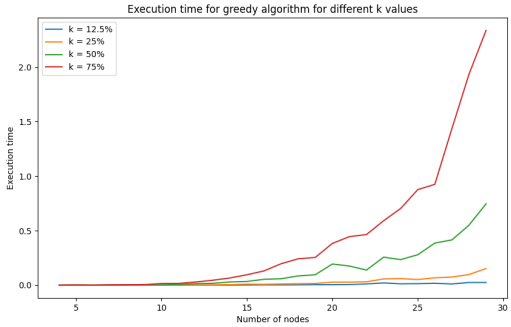

Fig. 3 - Number of nodes related to the amount of time needed to find an edge-dominating set of size k of a graph using a greedy algorithm

In Fig. 3, however, it was possible to perform a more detailed analysis, since the algorithm allows to find a solution in less time, and therefore, compute the edge-dominating set of size k for greater graphs more efficiently.

However, as was expected through the computational analysis of the greedy algorithm, the algorithm has a greater time complexity depending on the number of nodes, and similarly, greater k value.

As mentioned before, the analysed heuristic implementation took in consideration the size of the edge-

dominating set, number of graph edges and list of found edge candidates.

The graphs analysed also had the maximum number of edges as a percentage of the number of vertices. Therefore, higher percentages in the graph relate to higher graph edges. Both in Fig. 2 and Fig. 3, 75% the maximum number of edges was used for the number of vertices, which means that a greater number of vertices results in a greater number of edges in all cases.

Since there is a direct relation between the number of edges and the number of vertices, it is to be expected the greater computation effort in cases with greater number of vertices.

Additionally, greater values of k also result in the growth of processing time needed to find a viable solution.

Changing solely the number of the most efficient edges should also display a logarithmic growth. However, tests were not made regarding this aspect since the most efficient edges are calculated through private properties of the graph and are problem-specific.

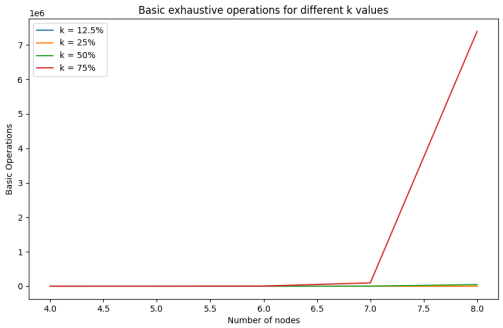

Fig. 4 - Number of nodes related to the amount of basic operations made while finding an edge-dominating set of size k of a graph using an exhaustive algorithm

The number of basic exhaustive operations consist on the number of iterations performed. This should be indicative of how many iterations were done when finding a solution.

In Fig. 4, we can relate the number of nodes to the number of basic operations. Similarly, the lines describe an identical relation comparing to the graphic that relates the number of nodes to processing time. This is to be expected, since, admitting that each single calculation takes a fixed amount of time, the relation between the basic operations and calculations should be almost constant.

In Fig. 5, the same phenomena happens, where the number of basic operations is tightly related to the amount of graph nodes. Expectantly, the number of basic operations increases significantly for greater amount of k values, which again translate to higher edge numbers.

Since the greedy algorithm is less demanding in terms of computational power, the relation between the number of nodes and the basic operations are more visible, as well as the differences in growth for different k

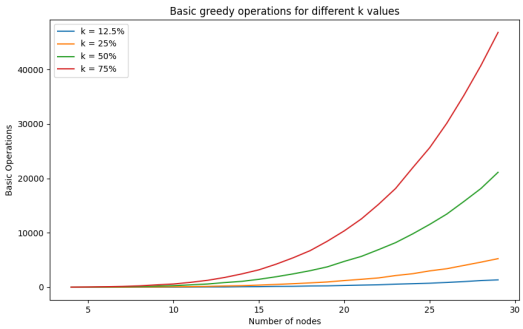

Fig. 5 - Number of nodes related to the amount of basic operations made while finding an edge-dominating set of size k of a graph using an exhaustive algorithm

values, with a slower growth for low k and respective higher growth for higher k.

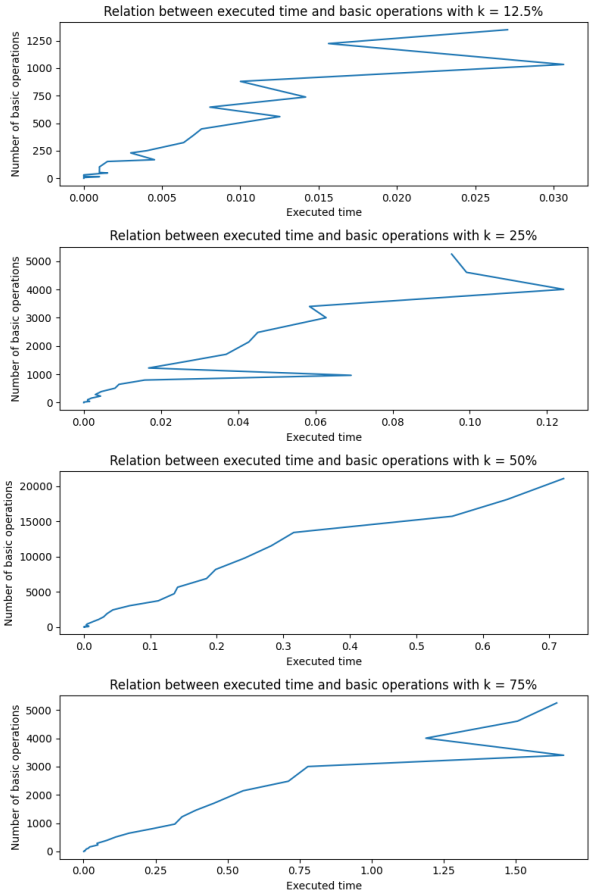

Fig. 6 - Relation between executed time and made operations within the greedy algorithm

The relation between basic operations and executed time should be, in both cases, related. Considering that each basic operation takes a fixed amount of time, the overall execution time should be proportional to the number of operations.

In Fig. 6 we can observe more closely this relation, using the greedy algorithm. Even though some of the values do not follow the main linearity, and therefore, might consist of outliers, it is clearly visible the relation

between both.

These outliers might be indicate of the sorting implementation performed in the heuristic approach, in which edges connected to high density vertexes should be searched first and foremost. Since, in Python, the function `sort()` is native to its language, the counted basic operations do not consider this sorting mechanism, and therefore, one of the reasons for the existence of these outliers might be because an increased time in performing this sort and therefore, increase the overall execution time of the algorithm, while the number of basic operations stays the same, since it considers only the number of edges and k value.

Therefore, since the execution time is dependent on the performance of the `sort()` function, and the number of basic operations does not take into consideration this sort mechanism, it is expected that with greater edge numbers, the list of efficient edges also has a larger size, and therefore, takes a greater time to sort them, which explains why the greater number of basic operations results in non-conforming values of execution time.

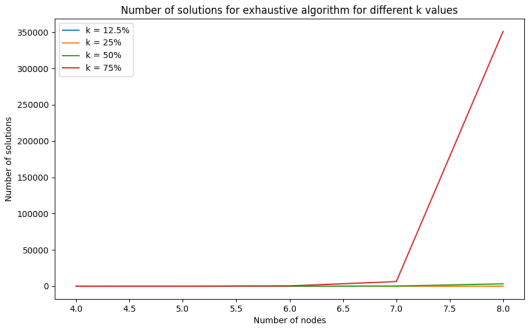

Fig. 7 - Relation between number of solutions and graph node size for exhaustive search

In Fig. 7 we can check the number of solutions for different k values. This graphic is specific to the exhaustive implementation since the greedy one only returns one solution, if it finds it. For higher k values it is possible to confirm the exponential growth of the number of solutions found, for higher numbers of nodes. This is due to the fact that the number of nodes and number of vertices are directly proportional, as well as for higher k values, the number of possible edge dominant sets is also higher, since the number of combinations of edges of size k also increases.

However, in order to check the accuracy of the greedy implementation results, the number of correct solutions was checked. To do so, the list of solutions of the exhaustive algorithm are all considered correct and there is no other solution to a specific graph. Then, a set of comparisons between the greedy algorithm solution and the solutions of the exhaustive one was performed, and if the greedy solution consisted of one of the solutions in the brute-force implementation, then the number of overall correct greedy algorithm guesses increased by one. Every time a comparison was made,

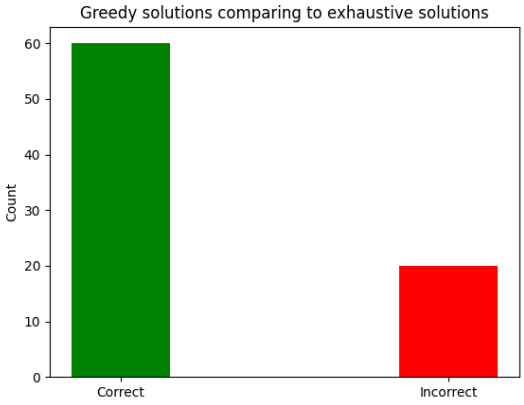

Fig. 8 - Relation between number of solutions and graph node size for exhaustive search

the number of overall comparisons increases by one as well, and so we can infer about the incorrect guesses by analysing the correct ones with the total ones.

Since the exhaustive solutions took too long to calculate, these results only infer in solutions from smaller graphs, that is, the number of graph nodes would go from four to eight.

However, we can analyse that most results are accurate for lower graphs, with a rate of success for finding a correct edge dominating set of 75% for the conducted experiments.

More computer power exigent tests would need to be performed to further check the greedy algorithm accuracy.

V. Result Discussion

The conducted research was implemented through a series of trials, with the purpose of implementing an algorithm capable of returning an edge dominating set which would be able to be more efficient than a brute-force approach, and therefore, be able to process larger graphs and greater edge dominating sets.

The amount of incorrect solutions may also not be simply due to the algorithm not always being able to detect a solution, but also the fact that the set returned might not be a dominant one. In order to address this issue, more tests should be performed to test the accuracy of the greedy algorithm and reliability in the solutions presented

The time consumed for processing graphs with a great number of nodes using exhaustive algorithm is also too extensive to better confirm and infer about the accuracy of some results, even though the main conclusions are mostly reliable and a pattern checks out.

The greedy algorithm, even though is able to process larger graphs and find a solution, might not always return a correct solution, or find no solutions when, in fact, there are some. This is due to the fact that the greedy algorithm settles on the fact that higher density nodes, which contain more edges, are the best to start searching for an edge dominating set, since its

believed they have a greater influence in the structure of a graph, which expands to edge dominating sets, which are sets of edges that, in a way, cover the entire graph.

Additionally, the amount of time consumed by the greedy algorithm, even though is definitely lesser than the exhaustive one, and is able to deal with larger graphs, could still be optimized, since the amount of time required to find a solution is still somewhat exponential, and should be optimized for even larger graphs and larger k size.

VI. Conclusion

This project allowed for greater graph comprehension, as well as learning how implementing heuristics has its ups and downs, since for a lesser execution time and higher graph density it sometimes returns a solution, and might not return any even when, in reality, it exists.

This also is indicative of the fact that the utilization of heuristics is problem-specific, depending on whether the results should rely on performance or reliability. Using a heuristic, depending on the type of problem, might not be the best approach to a problem that relies heavily on the solutions found.

By analysing the present results, we can confirm that the implemented heuristic is mostly reliable. However, it should be taken into consideration that this heuristic could be better in terms of finding better rules to get more reliable and less time consuming results.

This paper also demonstrates how heavily graph characteristics affect the results, their operations and the time it takes to find edge dominating sets. This is even more evident in graphs where some characteristics are the same. For example, in this paper, results in graphs with the same number of nodes are variant depending on edge density within the nodes.

References

[1] Diep N. Nguyen The Trung Tran M. Nguyen, Minh Hoàng Hà, "Solving the k-dominating set problem on very large-scale networks", *Computational Social Networks*, 2020.

[2] M. Yannakakis and F. Gavril, "Edge dominating sets in graphs", *SIAM Journal on Applied Mathematics*, vol. 38, no. 3, pp. 364–372, 1980.
URL: http://www.jstor.org/stable/2100648

[3] Bor-Liang Chen and Hung-Lin Fu, "Edge domination in complete partite graphs", *Discrete Mathematics*, vol. 132, no. 1, pp. 29–35, 1994.
URL: https://www.sciencedirect.com/science/article/pii/0012365X94902291