Minimum Edge Dominating Set for a graph

Ricardo Rodriguez (98388)
Universidade de Aveiro - DETI
AA - Algoritmos Avançados
ricardorodriguez@ua.pt

***Abstract*—This document describes the strategies, methodologies used for retrieving the minimum edge dominating set of an undirected connected graph through greedy and exaustive search, alongside with the results.**

***Index Terms*—exaustive, greedy, edge, adjacency**

I. Introduction

Graph theory problems have always gotten the interest from mathematicians, computer scientists and a variety of professionals who use it to solve problems (*e.g.* the travelling salesman problem [1]) and to help improve day-to-day tasks (*e.g.* find the shortest path between two locations).

In this report, we are going to look further into the Minimum Edge Dominating Set problem, an NP-hard problem, solve it using greedy and exaustive search and present the final results according to the search method, number of vertices and number of edges.

II. Problem

For a given undirected graph *G(V,E)*, with n vertices and m edges, a edge dominating set of G is a subset D of edges, such that every edge not in D is adjacent to, at least, one edge in D. A minimum edge dominating set is an edge dominating set of smallest possible size.

Additionally, this project requires that the undirected graph G is connected, but not the subset D of edges, and the number of edges sharing a vertex is randomly determined.

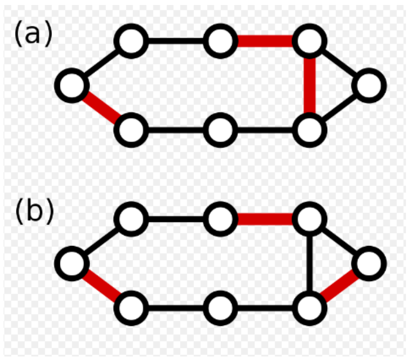

Fig. 1. Example of minimum edge dominating sets [2]

As we can see in *Fig. 1*, *(a)* and *(b)* are both minimum edge dominating sets. The edges with the color red constitute a minimum edge dominating set D, since the remaining edges in G, represented in black, are all connected to, at least, one of the edges belonging to D.

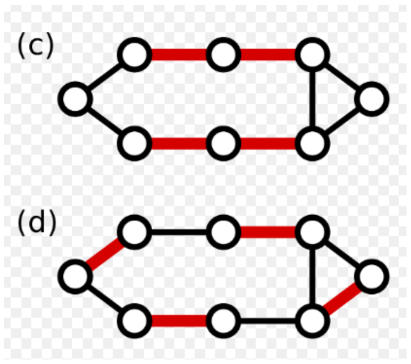

Fig. 2. Example of edge dominating sets [2]

In *Fig. 2*, *(c)* and *(d)* are also edge dominating sets. However, they are not the ones with the smallest possible size, since *(c)* and *(d)* subsets have both four edges while *(a)* and *(b)* subsets, represented in *Fig. 1*, have only three edges.

The minimum edge dominating set problem complexity augments when the number of edges in the undirected graph G increases. This being said, a graph G with 8 vertices and a 75% edge percentage has higher elapsed times to solve the problem than graph H with 9 vertices and a 50% edge percentage, since the number of edges in G is bigger than in H, although the number of vertices in H is higher.

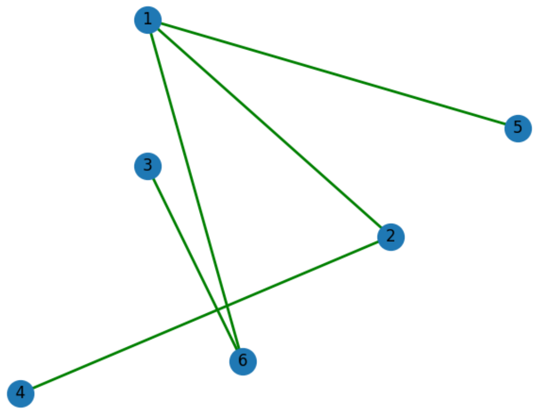

Fig. 3. Undirected graph with 6 vertices and a 25 edge percentage

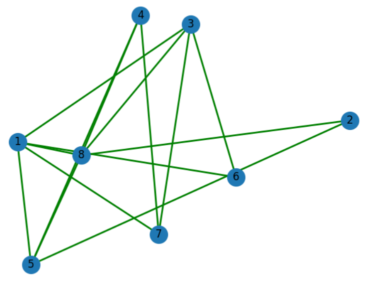

Fig. 4. Undirected graph with 8 vertices and a 50 edge percentage

III. Solution

The programming language used in this project was *Python3*. There are faster programming languages for this kind of computational problems like *C* or *C++* but, since the major goal of this project is to comprehend the difference between different algorithms and how to improve them, the programming language is indifferent.

The graph vertices have integer valued 2D coordinates between 1 and 100, a big range to facilitate the its visualization. All of them should neither be coincident nor to close to each other, having more than one unit of euclidean distance separating them from each other.

Besides that, for each number of vertices the program calculates the minimum edge dominating set with 12.5%, 25%, 50% and 75% of the maximum number of edges.

The minimum (*min*) and maximum (*max*) number of edges of an undirected connected graph with *n* vertices are described in the equations below:

$$min = n - 1 \tag{1}$$

$$max = (n * (n - 1))/2 \tag{2}$$

This being said, a graph with 8 vertices and with a 50% edge percentage is going to have 14 edges, while a graph with 4 vertices and a 25% edge percentage is going to have 3, since the percentage of edges in the graph must be bigger or equal than the minimum number of edges of an undirected connected graph (*Fig. 5*).

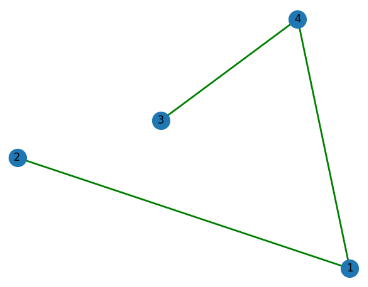

Fig. 5. An undirected graph with 4 vertices and with a 25% edge percentage (*max = ((4*3)/2)*25% = 6*0.25 = 1.5* does not satisfy the minimum number of edges requirement (*min = n-1 = 4-1 = 3*). Thus, the program must create a graph with 3 edges.

A. Adjacency List

In graph theory, an adjacency list is a collection of unordered lists used to represent a finite graph. Each unordered list within an adjacency list describes the set of adjacent neighbours of a particular vertex in the graph. [3]

On the other side, an adjacency matrix is a square matrix also used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent (1) or not (0) in the graph. [4]

Fig. 3 depicts an undirected graph, alongside with its adjacency matrix and adjacency list.

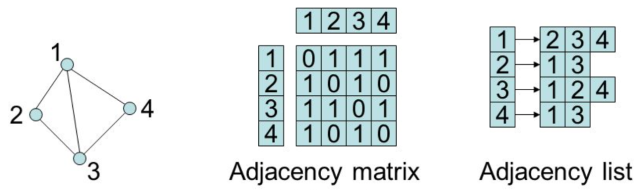

Fig. 6. Adjacency matrix and adjacency list for an undirected graph

For this problem, it's better to use an adjacency list in favor of an adjacency matrix, since it's faster to retrieve and iterate through all the edges connected to a vertex.

Using an adjacency matrix would be better in problems where we need to find if two vertices are connected with each other, which is not the case. Additionally, the space complexity is of an adjacency matrix is quadratic Ô(n²) whereas the space used in an adjacency list depends on the number of vertices and edges, usually saving more memory when the graph's edges are sparse.

B. Exhaustive Search

In computer science, exhaustive search or brute-force search is an algorithm method consisting of retrieving all possible combinations and verify, one by one, if the candidate matches the requirements to solve the given problem. Although it's easy to implement and will always find a solution if it exists, this method is slow when the candidates set is large.

The *exhaustive_search*() method implements in the project's code is presented below. Initially, the *generate_subsets*() is a generator function that behaves like an iterator and yields each subset of the given set (edges list). The *yield* keyword states that the used function is a generator and allows the return of a single value instead of the whole list of combinations, saving space in memory.

The brute-force method retrieves each edge combination set one by one and iterates through each edge of the subset, removing all the edges in the *remaining_edges* variable which are adjacent to the current edge we are analyzing (true if both edges share a common vertex) until there is no more remaining edges in the *remaining_edges* list, meaning we found a minimum edge dominating set.

```
# Exhaustive search - Return when minimum
    edge dominating set is found
def exhaustive_search(self):
   basic_operations = 0
```

```
    for subset in
        self.generate_subsets(self.graph.edges):
       remaining_edges = self.graph.edges
       for edge in subset:
          basic_operations += 1
          remaining_edges = [ e for e in
              list(remaining_edges) if e[0]
              not in edge and e[1] not in edge
              ]
          if not remaining_edges:
             return (subset, basic_operations)

# Generator of all subsets of a given set
    (edges)
def generate_subsets(self, edges):
    for subset in
        chain.from_iterable(combinations(edges,
        num) for num in range(len(edges)+1)):
       yield subset
```

The computational complexity of going through all the edge subset combinations is the number of k-combinations for all k where k is the number of subsets of graph *G(V,E). The respective equation is presented below:*

$$\sum_{k=2}^{n} \binom{n}{k} = 2^n \tag{3}$$

*In the worst case, the complexity of iterating all possible edge combinations is O(2^n), being n the number of edges of the graph. We also need to iterate through k edges of a subset (O(n)) and, for each edge of a subset, remove all the remaining neighbouring edges (O(n)). This being said, the complexity of the exhaustive search is O(2^n)*O(n^2)=O($n2^n$).*

C. Greedy Search

In opposition to the brute-force algorithm, greedy search uses heuristics at each stage of to find an optimal solution for the presented problem. It's faster than the traditional exhaustive method since it evaluates the best route we can redirect a given scenario to solve the problem.

In this implementation below, when the greedy_search() method is first called (counter == 0, the items of the adjacency list are sorted in a reverse order, meaning the vertices with the higher number of neighbours come first and the ones with lower values are the last. In the same way, the same procedure occurs with the vertices in a vertex adjacency list, which are also ordered in a descending way by their number of neighbours.

When the adjacency list, represented in a dictionary data structure where keys are vertices and values are a list of neighbouring vertices, is sorted, the edges variable is responsible for storing all the edges of the graph.

Since we want to find an optimal solution, the program will start looking at first edge in the edges list and removing it. The first item on the list contains the edge with the current biggest number of neighbouring edges, since those vertices together share the most number of neighbouring edges.

This is the best way to find a solution to the minimum edge dominating set problem, since it always starts checking if the current edge, which is the most connected to other edges, satisfies the conditions of the problem.

The function responsible for verifying if a candidate is the solution for the question is the verify_edge_dominating_set() one. It needs two parameters: the remaining edges to analyze (edges) and the first element popped from the list (edge). The remaining_edges variable only stores edges which are not adjacent to the edge and the remaining adjacency list is then created and returned in the end.

After the function is called, the result is stored in the adjacency_list variable. If the returned adjacency list is empty, that means the function has found a solution where all the subset edges are adjacent to the other remaining edges, returning the edge and the number of basic operations (counter.

If the solution has not been found yet, the recursive function is called again with an updated adjacency list that doesn't contain the neighbouring edges of the last verified edge.

This way, when a solution is found the edges are grouped together by the recursive calls and the minimum edge dominating set is returned, alongside with the number of basic operations the method has done.

The computational complexity of going through all combinations of edges is mathematically presented in the following equation:

$$\sum_{k=2}^{n} \binom{n}{k} = 2^n \tag{4}$$

*In the worst case, the complexity of iterating all possible edge combinations is O(2^n), being n the number of edges of the graph. We also need to iterate through k edges of a subset (O(n)) and, for each edge of a subset, remove all the remaining neighbouring edges (O(n)). This being said, the complexity of the exhaustive search is O(2^n)*O(n^2)=O($n2^n$).*

```
def greedy_search(self, adjacency_list,
    counter=0):

    if counter == 0:
       adjacency_list =
           dict(sorted(adjacency_list.items(),
           key=lambda x : len(x[1]),
           reverse=True))
       for data in
           list(adjacency_list.items()):
          adjacency_list[data[0]] =
              sorted(data[1], key = lambda x :
              len(adjacency_list[x]),
              reverse=True)

    edges = []
    [ edges.append((v1,v2)) for v1, edge_list
        in adjacency_list.items() for v2 in
        edge_list if (v2,v1) not in edges ]

    while edges:
       counter += 1
       edge = edges.pop(0)
       adjacency_list =
           self.verify_edge_dominating_set
                                (edges,edge)
       if not adjacency_list:
          return ([edge], counter)
        greedy =
            self.greedy_search(adjacency_list,
            counter)
        return ( [edge] + greedy[0], greedy[1])

def verify_edge_dominating_set(self, edges,
    edge):

    remaining_edges = [ e for e in edges if
        e[0] not in edge and e[1] not in edge ]
    remaining_adjacency_list = { e[0] : [e[1]]
        for e in remaining_edges }
    for e in remaining_edges:
        if e[0] not in remaining_adjacency_list:
            remaining_adjacency_list[e[0]] =
                [e[1]]
        else:
            remaining_adjacency_list[e[0]] +=
                [e[1]]
    return remaining_adjacency_list
```

For the greedy search, the worst case complexity is $O(2^n)$, n being the number of edges of the graph. However, the greedy search is very fast comparing to the exhaustive search because of the heuristics. The methods always starts analyzing the smallest possible combination of edges with the most number of neighbours until it finds an optimal solution.

D. Results

In Fig. 7 and Fig. 8 are presented results from the exhaustive and greedy search methods, each one of the graphs having different numbers of vertices and edges.

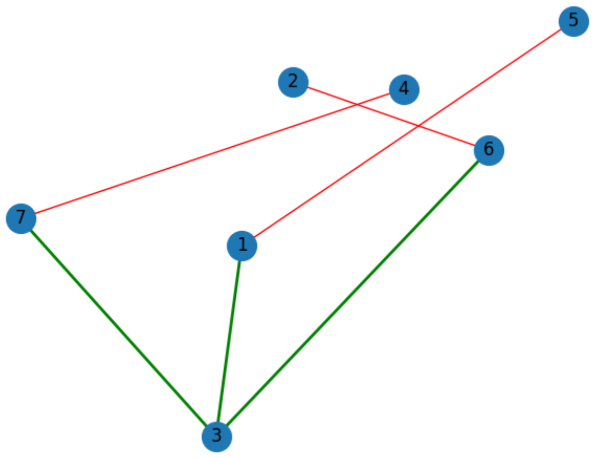

Fig. 7. Minimum edge dominating set (*in green*) of an undirected graph with 7 vertices and a 12.5% edge percentage

Both Table I and Table II display the number of basic operations and the elapsed time of graphs with different number of vertices and edges for greedy and exhaustive methods, respectively.

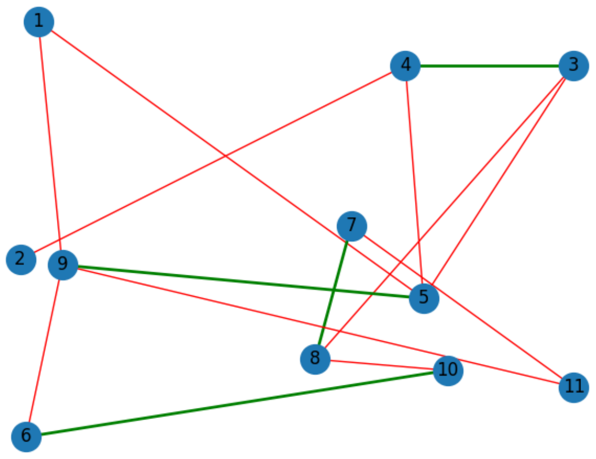

Fig. 8. Minimum edge dominating set (*in green*) of an undirected graph with 11 vertices and a 25% edge percentage

TABLE I
GREEDY SEARCH RESULTS

Vertices	Edge Percentage	Basic Operations	Time (s)
5	25	2	1.74e-05
5	75	2	3.98e-05
7	50	3	5.48e-05
10	25	4	4.24e-05
10	75	4	9.56e-05
15	25	7	0.000112
15	25	7	0.00028
25	50	12	0.00128
50	50	25	0.03044
100	50	50	0.530033
200	50	99	13.958
250	50	125	40.6416
270	50	134	58.710

TABLE II
EXHAUSTIVE SEARCH RESULTS

Vertices	Edge Percentage	Basic Operations	Time (s)
5	25	8	1.57e-05
5	75	18	4.10e-05
7	50	22	6.63e-05
9	25	114	0.00011
9	75	10272	0.03086
11	25	319	0.00041
11	75	67361	0.22999
14	25	20750	0.06555
14	75	59234144	187.673
16	25	191705	0.39644
16	75	419497272	1777.095
17	12.5	46917	0.04995
17	25	1755516	3.5929

E. Conclusion

According to the obtained results, we can conclude the greedy search is the best method when looking for a fast and an efficient way to find a minimum edge dominating set, although this doesn't always find the best solution but the optimal one.

The exhaustive search is the best option for situations where the number of vertices and edges is small, since the elapsed time is very low. However, the number of basic operations exponentially increases when the number of edges grows with this method.

This project promotes the reflection on best practices when implementing a complex algorithm that needs to be refactored to handle large amounts of basic operations in order to reduce its computational complexity and be more time and space efficient.

References

[1] "Discrete mathematics:" Traveling Salesman Problems. [Online]. Available: https://www.jiblm.org/mahavier/discrete/html/chapter-6.html. [Accessed: 31-Oct-2022].

[2] Edge dominating set (2021) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Edge_dominating_set (Accessed: October 31, 2022).

[3] Adjacency list (2022) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Adjacency_list (Accessed: October 31, 2022).

[4] Adjacency matrix (2022) Wikipedia. Wikimedia Foundation. Available at: https://en.wikipedia.org/wiki/Adjacency_matrix (Accessed: October 31, 2022).

[5] Fundamentals of data structures- graph. Available at: https://bournetocode.com/projects/AQA_A_Theory/pages/graph.html (Accessed: October 31, 2022).