# Advanced Algorithms Assignment 1 – Topic #12

Diogo Bento 93391

***Resumo* – O presente artigo detalha tentativas de resolução do problema de cálculo do conjunto dominante de arestas de um grafo com peso mínimo no contexto da Unidade Curricular de Algoritmos Avançados da Universidade de Aveiro.**

**Este artigo apresenta duas soluções para o problema, includindo análise de complexidade computacional, cujos resultados serão comparados a dados obtidos ao correr o programa.**

***Abstract* – This paper details an attempt at solving the problem of calculating the minimum weight edge dominating set of a graph, tackled within the context of the Advanced Algorithms Curricular Unit of Universidade de Aveiro.**

**This article describes two different approaches to solving this problem including complexity analysis work that will be compared to real data obtained from running implementations of both solutions.**

## I. INTRODUCTION

The **dominating edge set** of a graph is a set of edges such that every edge not contained within the set is adjacent to an edge that is part of the set.

The **minimum weight dominating edge set** is a variation of this concept: given a weighted graph we must find the dominating edge set with the smallest sum of edge weights.

## II. ALGORITHMS

### A. Graph Generation

This project uses a graph generator (***graph.py***) that outputs a **networkx graph** as JSON.

Generated graphs will have nodes placed within a grid with x and y coordinates ranging from 1 to 9.

Nodes cannot be directly adjacent, including diagonally.

Every edge has weight equal to the Euclidean distance between their nodes:

$$w = \sqrt{(x1 - x2)^2 + (y1 - y2)^2}$$

The generator will always create a connected,undirected graph without any parallel edges as long as there is enough space to place all nodes and the edge count parameter is within acceptable bounds.

The generator has the following parameters that can be used to influence it:

- **--seed**: seed to be used for the generator, default is **93391**
- **--nodes**: how many nodes the algorithm must attempt to place,defaults to **5**
- **--edges**: how many edges to place, value defaults to **10** and must be in the **[n-1,n*(n-1)/2]** range, with n being the number of nodes
- **--output**: file to write the graph to, default is **graph.json**

### B. Solution Detection

Any given solution is a list of edges which contain information about the nodes that they bridge.

The algorithm used to validate a solution is as follows:

```
VALID = FALSE
SEEN_NODES = SET()
FOR EDGE IN SOLUTION:
    SET.ADD(EDGE.NODES)
MISSING = EDGES - SOLUTION
FOR EDGE IN MISSING:
    IF EDGE.NODES NOT IN SEEN_NODES:
        BREAK
    MISSING.REMOVE(EDGE)
IF MISSING IS EMPTY:
     VALID = TRUE
```

### C. Exhaustive Search

Exhaustive search iterates over all possible solutions and returns the cheapest valid solution found.

The implementation used to benchmark performance iterates all combinations ordered by their length.

The node set will always be computed but the implementation only finishes checking whether a solution is valid if it has less total weight than the best solution that has been seen so far.

Due to checking every possible solution exhaustive search will always provide the best solution, however being this thorough is very costly performance wise.

This method will always check

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$$

solutions, while the number of nodes added to the set is

$$2\sum_{k=1}^{n}\binom{n}{k} * k = 2^n n$$

with n being the number of edges.

### D. *Greedy Solution*

The greedy implementation pre-emptively sorts edges by weight and creates a solution queue that starts with said ordered edges.

Solutions are removed from the queue and validated, with the first valid solution being returned.

If a solution is not valid more solutions are generated based off it, by appending all edges that are more expensive and contribute nodes to the solution, and said solutions are appended to the queue.

This process guarantees that the algorithm will always return a minimum-length solution and that said solution will have the least weight of all valid solutions of the same length, with no duplicate solutions being generated.

If this queue were to be changed to a priority queue by positioning new items based on weight rather than just appending them at the end of the queue we would always obtain an optimal solution.

In a connected, loopless graph it is guaranteed that if every node except one is connected to an edge in our set we have a edge dominating set. As each edge connects 2 nodes we only need **nodes/2** edges to achieve an edge dominating set in the worst case scenario meaning the worst possible tested solution count for this method is

$$\sum_{k=1}^{n/2}\binom{e}{k}$$

with n being the number of nodes and e being the number of edges and the number of set adds being

$$\sum_{k=1}^{n/2}\binom{e}{k} k$$

In practice the observed number of solutions and adds tended to be much lower than these upper bounds.

## III. ANALYSIS

### A. *On obtaining data and visualizations*

In order to obtain performance data the solver was run using graphs with the following parameters:

$$nodes \in [0,3]$$

$$edges \in [nodes - 1, nodes * \frac{nodes-1}{2}, \max 20]$$

The upper boundary to edge count had to be limited due to exhaustive search's exponential growth.

The obtained results cant be found at *results.csv* and the script used to be obtain then can be found at *datagen.sh*.

Any plots used in this report can be seen and interacted with by running *plots.py*.

### B. *Results*

The results confirmed that the formulas for both solutions seen and set adds performed for the exhaustive method are correct and that that the upper bounds calculated for the greedy method are not exceeded.

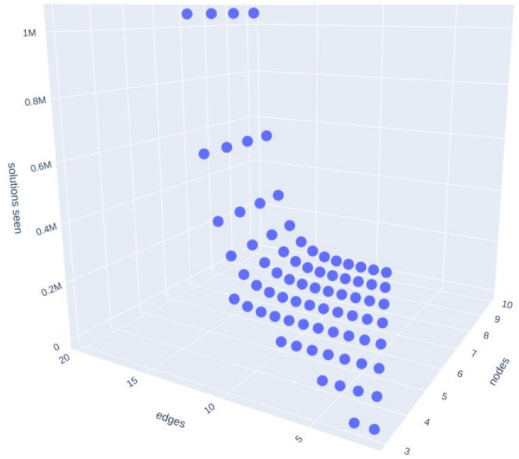


Fig. 1 - Solutions Seen, Exhaustive Method

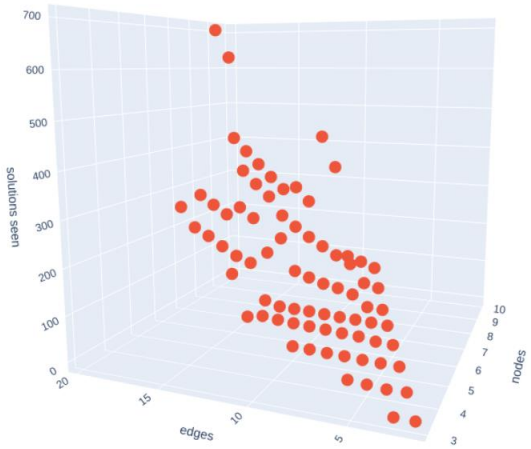


Fig. 2 - Solutions Seen, Greedy Method

As expected exhaustive search's solution count is independent from how many nodes the graph contains, while greedy's values vary with node count.

It is worth remembering that greedy search's solution count is not consistent across seeds and performance may vary up to the upper bounds described previously.

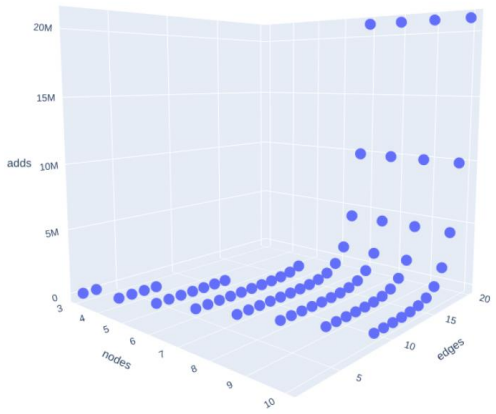


Fig. 3 - Set adds attempted, Exhaustive Method

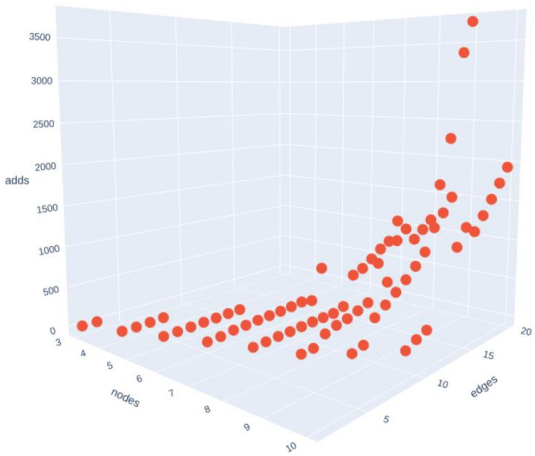


Fig. 3 - Set adds attempted, Greedy Method

When it comes to total solution costs the greedy algorithm generally obtained results close to those the exhaustive method.

The worst cost ratio between the greedy solution and exhaustive solution was 165%, meaning a +65% weight increase in the greedy solution, the median cost ratio was 100.43%, and 75% of observed data points had less than 14.5% increased weight.

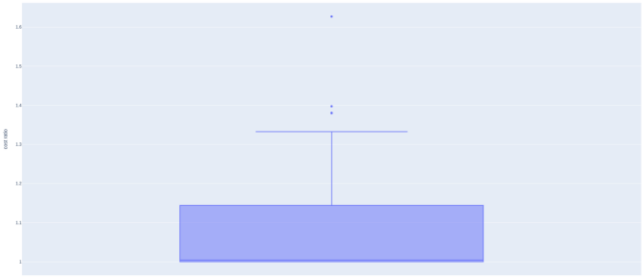

Fig. 5 - Solution Cost Ratio Distribution

These costs were distributed as follows:

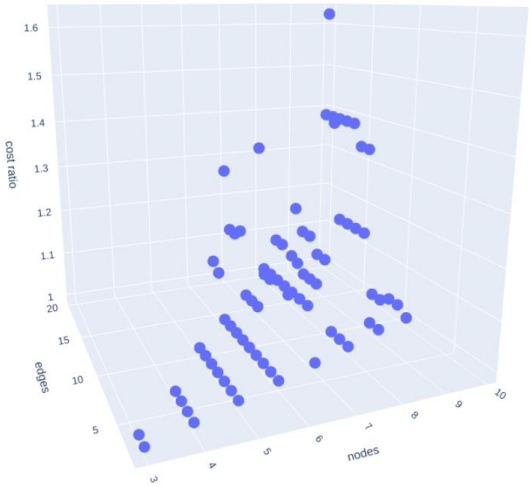


Fig. 6 - Solution Cost Ratio Distribution (Scatter Plot)

There seems to be a tendency for increased added cost when dealing with larger, more complete graphs.

This is probably due to the greedy method always picking the first, shortest solution it finds.

Larger edge counts are more likely to have longer, cheaper solutions that are ignored, leading to worse cost ratios.

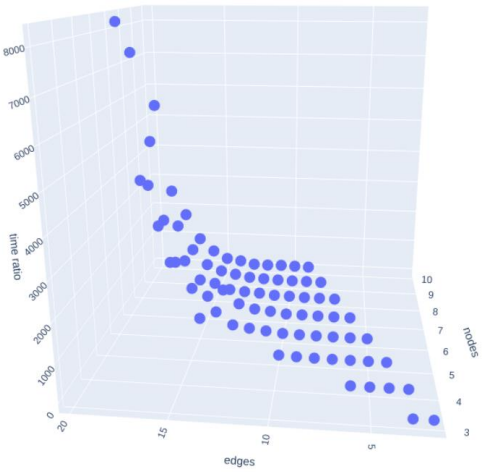


Fig. 7 - Solve Time Ratios

As edge count increases the solve time ratio goes up steeply.

For a given node count increasing the number of edges by 1 often leads to the solve time ratio increasing by a factor of 1.5 to 2 times.