Maximum weighted matching: exhaustive and greedy search

Tiago Gomes Carvalho

***Resumo* - O problema do emparelhamento máximo no estudo de grafos consiste em encontrar um emparelhamento, ou seja, um conjunto de arestas não adjacentes entre si, para o qual a soma dos pesos das suas arestas seja o maior possível no grafo. Neste relatório, será detalhada a implementação de duas abordagens algorítmicas para resolver o problema: uma abordagem exaustiva (força bruta) simples e uma abordagem heurística gulosa. Será conduzida uma análise detalhada e explicação com o intuito de examinar os resultados obtidos pelos algoritmos.**

***Abstract* – The maximum weighted matching problem in graph theory comes down to finding the matching, i.e., a set of pairwise non-adjacent edges, for which the sum of the weights of its edges is as large as possible in the graph. In this report, it will be detailed the implementation of two algorithmic approaches to solve the problem: a simple exhaustive (brute-force) approach and a greedy heuristic approach. A detailed analysis and explanation will be conducted in order to examine the results produced by those algorithms.**

***Key-words* – matching, exhaustive search, greedy heuristic, undirected graph, algorithm analysis, algorithmic scale**

I. Introduction

A matching of an undirected graph *G (V, E)* is commonly described as a subset of edges that don't share any common vertex among themselves [1]. In this particular problem the challenging part is finding a matching in which the weight of its edges is maximum. Since the weight needs to be measured when searching the matching, the graphs must be edge-weighted, preferably using a decent set of possible weights.

This is also an optimization problem since it focuses on finding the optimal solution among all the possible solutions/matchings. It is also a combinatorial optimization problem, because the number of possible subsets in each graph is finite and the number of solutions can be directly calculated.

The two approaches in study, exhaustive and greedy heuristic, were developed in Python3.11 and represent two possible ways to solve this problem, as well as multiple similar problems.

II. Study Environment

The test environment consisted of 1040 graphs generated randomly using a seed and stored locally. The graph instances are defined by a set of 2D non-coincident vertices, with integer valued coordinates and represented numerically, and a set of edges also randomly determined.

A. Graphs for the computation experiments.

Each generated graph differs and has a different set of characteristics, number of vertices and edge density.

The graphs we generated including a range of vertices from 4 to 264 and for each number of vertices four distinct edge densities *d* were used: 12.5%, 25%, 50% and 75%.

$$\sum_{i}^{v} i \times \sum_{j}^{d} j = \text{numer of graphs}$$

In this case:

$$\sum_{i=4}^{264} i \times \sum_{j=1}^{4} j = 1044$$

B. Generating and storing the graphs.

The graphs were generated using Python's "*Networkx"* and "*Random*" packages and stored in a file to be used for the computation experiments.

As detailed before, this problem involves the usage of edge-weighted graphs. Therefore, when generating the graph in study, there were assigned weights to each edge in the graphs randomly determined by the seed provided.

```
for _ in range(num_vertices):
    while True:
        vertex = (r.randint(1, 100), r.randint(1, 100))
        if vertex_not_too_close(vertex):
            G.add_node(vertex)

setWeights(G{(u, v): r.randint(1, 10)})
```

C. Conducting the study.

The same experimental procedure was made for both algorithmic approaches and the results were saved in individual files for each graph. The files are name according to the graph they represent, for example, "5_0.25_results" representing the graph with 5 vertices and 25% edge density. The files are also group in two files, depending on the algorithm used to obtain the results.

III. Exhaustive Search

The exhaustive search algorithm consists of a brute-force approach usually used to solve combinatorial problems. In this approach, all possible combinations are searched in order to try to find the optimal one. Using this simple approach, it is guaranteed to obtain the correct maximum weighted matching, however it generally leads to unnecessary operations, high complexities and too inefficient to solve problems in larger scales [2]. The exhaustive search algorithm in study is a comprehensive approach that combines recursive depth-first search with pruning in order to try to solve those problems.

The recursive function explores all combinations of edges recursively and, in each call, represents a decision to either include or exclude a specify edge in the current matching. Indexing each edge in a set containing all the edge sorted by weight, it is possible to check if that edge's vertices were not already covered and if including that edge in the matching leads to a possible solution.

Upon reaching the end of the edge list, technically imposing a possible matching as solution, the function evaluates if the current weight is greater than the maximum weight previously registered and, if so, it updates the value. Then, the recursive function continues until it fully explores all the remaining edge combinations and consequently returns the maximum weighted matching and its weight.

In order to achieve the results quicker, avoid unnecessary operations and tests redundant solutions, techniques of pruning are applied to the approach [3]. The pruning depends on the searching current state and the remaining edges to iterate. Essentially, it checks if the sum of the current weight and the maximum possible weight from the remaining edges is less than or equal to the current value of the maximum weigh. If this checks out, searching the remaining edges won't lead to a solution and the function goes back to explore new combinations.

The algorithm will always present a solution, i.e., all the graphs have a maximum weighted matching, since at worst the matching simply consists of the edge with the greatest weight. However, if the input graph doesn't contain any edges or if those edges are not weighted the algorithm won't be able to return a maximum weighted matching. It will instead return an empty list and a maximum weight equal to zero.

In Python *pseudocode*, the code structure of the recursive function looks similar to the following one:

```
def exhaustive_recursive_func(weight, matching):
    if remaining_possible_weight < max_weight:
        return
    if num_remaing_edges == 0:
        update(max_matching, max_weight)
        return

    edge = all_edges[current_index]
    if edge.vertices not covered_vertices_list:
        update(matching, weight, covered)
        exhaustive_recursive_func()
        update(matching, weight, covered)

    exhaustive_recursive_funct()
```

A. Formal analysis.

The exhaustive search algorithm complexity is $O(2^n)$, which means it increases exponentially with the number of edges n. This is due to the recursive searching of all the possible matchings, but the pruning techniques can reduce the actual number of combinations explored.

Excluding the possibility of pruning, 2^n edges will be explored in total, performing a number of operations equally exponential to the number of edges in graph. The number of solutions doesn't have any evident correlation with the number of edges.

This complexity doesn't change regardless of the input graph and the recursive function can, at worst, search the optimal at the last call.

B. Experimental analysis.

To actually verify the analysis explained above, the algorithm was applied to the graphs randomly generated, which corresponded to the following characteristics:

- Number of vertices $v \in [4, 20]$
- Possible edge density $e \in \{0.125, 0.25, 0.5, 0.75\}$
- Edge weight's randomly assigned $w \in [1, 10]$

In the study, three fundamental metrics were used to test the efficiency of the algorithm: the number of operations executed, the number of possible solutions detected and the execution time it took the algorithm to iterate the graph. The number of operations consists of the sum of all the considerably basic operations like value updates and conditions checked. The number of possible solutions describes the number of times the algorithm effectively found a new maximum weighted matching. The process time started counting from the first function's operations to the one preceding the final result return.

Running the code multiple times is not expected to produce results with values very different from those used in the analysis.

The Figure 1 represents the number of operations over the number of basic operations in a logarithmic scale over the number of vertices. Examining the growth of number of

operations in each edge density, and since the scale is logarithmic, it's possible to affirm that the number of operations increases exponentially with the number of vertices, because in the visualization it is noticeable a linear behaviour regardless of the edge density in the graph. Besides that, the difference in the slope growths between the edge densities is also relevant. As expect by the computational complexity of the problem, greater edge density will result in an exponential increase in the number of operations registered.

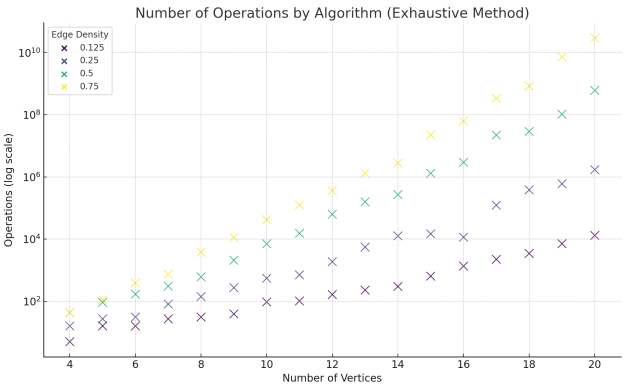

Fig. 1 - Number of basic operations over the number of vertices in the exhaustive search.

The Figure 2 presents the correlation between the execution time (in seconds) on a logarithmic scale and the number of vertices for each graph. The behaviour is essentially similar to the one previously mentioned, showing an exponential growth for each edge density. However, this time it's possible to see the main reason why the analysis of the exhaustive search was restricted to graphs with a maximum of twenty vertices. Slightly increasing the number of vertices means a tremendous increase of the time it will take the algorithm to iterate all the edges.

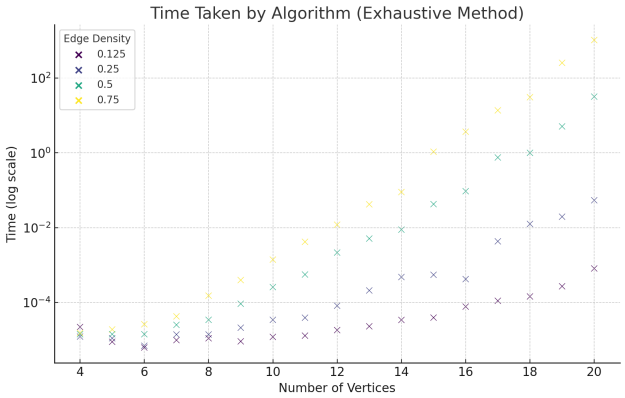

Fig. 2 - Execution time over the number of vertices in the exhaustive search.

The Figure 3 represents the number of solutions checked over the number of vertices. This time it's not possible to see a clear pattern between the two objects being quantified. Since the objective of algorithm is finding the maximum weighted matching, the number of solutions being counted in each run is only determined the number of matchings searched that have equal weight to the matching returned by the algorithm.

IV. Greedy Algorithm

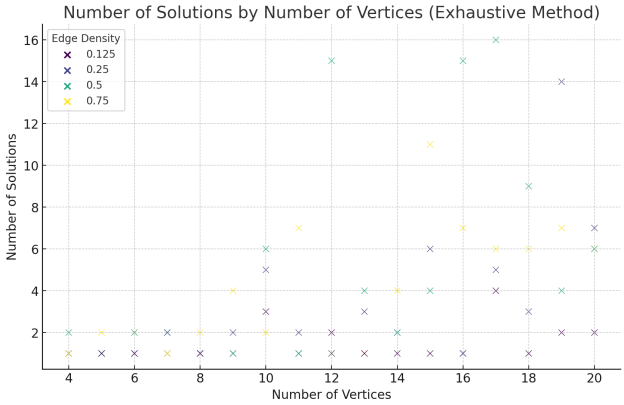

Fig. 3 – Number of solutions checked over the number of vertices in exhaustive search.

One way to simplify the complexity of an exhaustive search is applying a heuristic way to reduce the number of operations or restricting the solutions checked. Implementing these heuristics sometimes implies not obtaining an optimal solution as result. In this case, the greedy heuristic applied significative reduces the complexity of the search and restricts the solutions explored to just one single solution [4].

Essentially, it starts by sorting the edges by their weight in a descending order and subsequently adds the edge with the next greatest weight to the matching if none of its vertices were already covered by any other edge. After iterating through all the edges, it will produce a valid matching containing the maximum weight calculated by that iteration [5].

Assuming the algorithm iterates through every edge in a graph with more than one edge, it's possible that the maximum weighted matching is solely constituted by the first edge searched. However, it's impossible to find a matching that consists of the last edge searched, since that would have meant that the ones with greater weight were not included.

The following Python *pseudocode* represents the implementation of the described function:

```
def greedy_heuristic_search(graph):
    matching = set()
    weight = 0
    v_covered = set()
    sorted_graph = sort(graph, edge.weight)

    for edge in sorted_graph:
        if edge.vertices not in vertices_covered():
            update(matching, weight, v_convered)

    return matching, weight
```

The algorithm automatically excludes the edges sharing vertices with all previously selected edges without checking their weights.

This algorithm is considered to be based on a greedy heuristic approach since all of its conditional choices are locally restricted. Upon reach the last edge is officially considers the matching produced to be the maximum weighted matching, regardless of the edges that were discriminated by have common vertices with the ones previously included.

A. Formal analysis.

The computational complexity of the heuristic approach is dominated by the sorting step, which is O (n log n). The iteration over edges and set operations contribute with a complexity of $O(n)$, making the final complexity of the algorithm O (n log n). This lower computation complexity makes it a valuable approach in scenarios when the approximate solution is acceptable or where the graph structure favours greedy choices.

B. Considering the worst case.

Besides the advantages of using such this heuristic, it's also necessary to fully explain the unexpected results that might be associated with heuristic approaches.

As mentioned above, once an edge is selected as the edge with the next greatest weight, it means that all the edges sharing vertices with the selected one will be excluded from being part of the matching. Unfortunately, using this kind of heuristic it is impossible to understand if this decision was a smart one, according to the final objective of finding the theorical maximum weighted matching. The decision of discriminating edges is not well defined and needs to be refined to order to be completely reliable.

To understand the undesired result that can be produced, it will be taken, as an example, the following graph:

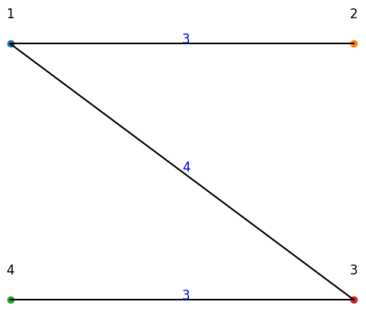

Fig. 4 – Undirected graph with weighted edges.

The Figure 4 consists of an undirected graph G (V, E) with four vertices and three weighted edges.

According to the greedy heuristic, the search will start by sorting the edges by their weight in a descending order which will lead to the edge E ($v1$, $v3$, *weight* = 4) being searched first. However, upon selecting that edge for the matching, both the remaining edge will be excluded to be part of the matching, since they share $v1$ and $v3$ with the selected edge. Even though the two edges could possibly be part of a matching, that matching won't be searching in this algorithm.

The maximum weighted matching returned by the greedy search will be {E ($v1$, $v3$, *weight* = 4)} having a maximum weight of four, but if the matching {E ($v1$, $v2$, *weight* = 3); E ($v3$, $v4$, *weight* = 3)}} was searched, the final maximum weight would be six.

C. Experimental analysis.

The testing of this algorithm was similar to the one done for the exhaustive search, but in this case a larger set of input graphs was considered. Data regarding graphs with the number of vertices $v \in [4, 264]$ was obtained as results of the algorithm iterating each of those graphs. The edge density and the range of edge weights was the same as used before.

The same metrics were also maintained (except the number of solutions checked), so it would be possible to examine the results produced by the two approaches easier.

The Figure 5 represents the number of operations over the number of vertices. The curve suggests the exponential growth of the number of operations regarding larger input graphs.

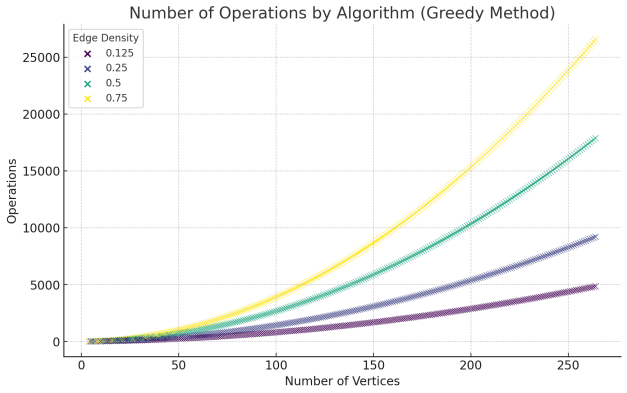

Fig. 5 - Number of operations over the number of vertices in the heuristic approach.

The Figure 6 represents the execution time (in seconds) over the number of vertices. The behaviour is identical to the described before.

Some data regarding time values that didn't follow the growth path were registered. However, considering that these values are so lower (log scale 10^0), these values were attributed to possible memory spikes during the runs.

Examining data, it's possible to see that graph with a greater number of vertices could be easily tested, however, the growth slope of the execution time won't change regardless of the input size.

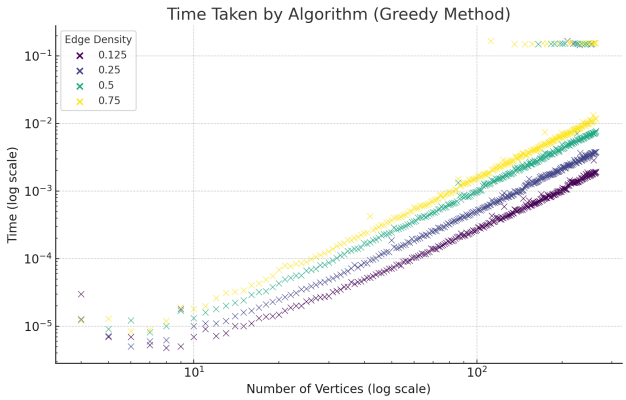

Fig. 6 – Execution time over the number of vertices in the heuristic approach.

Since the object in study is the greedy heuristic approach, it's not possible to check more than one solution. Once the algorithm finishes iterating the edges' weights, it automatically declares the produced matching as the one to be returned.

V. Comparing Both Algorithms

Merging the results from both studies, it's possible to produce new visualizations that can show the differences between the two algorithms. The number of operations, time and time taken can be compared in the Figure 7 and Figure 8.

The key point to have take when viewing this figures is the difference between the magnitudes of the values registered. Since both figure are presented with the axis on a logarithmic scale, it's also possible see the contrast between the two computation complexities.

The last comparation in this study is represented in the Figure 9, which shows the maximum weight returned over the number of verticies for both algorithms, with the *x* and *y* axis on the algorithm scale. In this figure we can take conclusions about the previously mentioned topic regarding the correctness of the the greedy algorithm.

Since the exhaustive search always produces the optimal maximum weighted matching, the values in the visualization related to that algorithm can be assumed to be right.

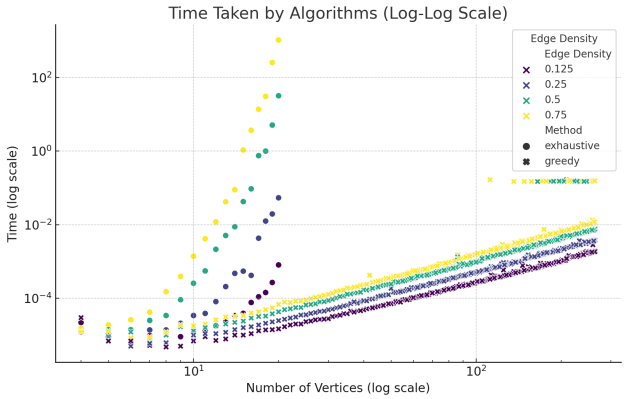

Fig. 7 – Execution time over the number of vertices in both approaches (logarithm scale).

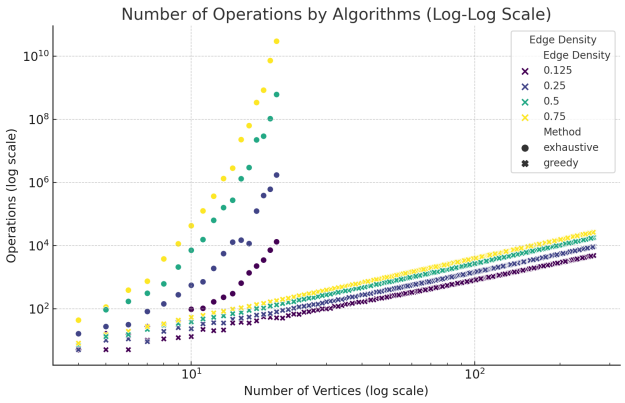

Fig. 8 – Number of operations over the number of vertices in both approaches (logarithm scale).

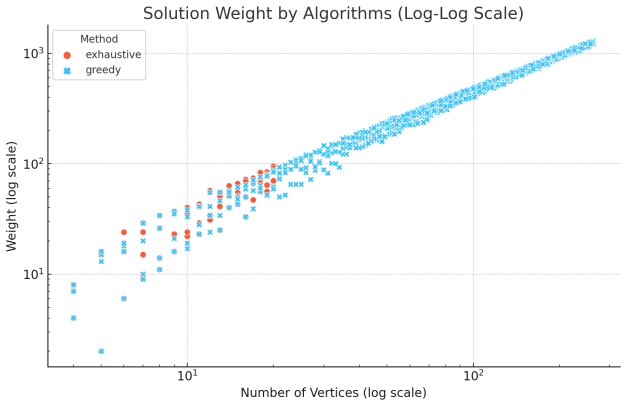

Fig. 9 – Weight over the number of vertices in both approaches (logarithm scale).

Upon closer look it's possble to see in the Figure 9 that the greedy heuristic algorithm didn't produce results that varied by a lot from the expected. This is the most important factor when considering the algortihm as reliable.

Sometimes the decision of implementing greedy heuristics comes to the necessity of obtain correct values, but in other ocasions a decente range of posssible values is enough to understand the expected solution to a input graph.

As the Figure 9 suggests that the weigth increseses with the graph's own complexity, i.e. number of vertices and edge density.

VI. Final Considerations

In this report, some factors regarding the conditions of the experimental analysis weren't completely explored. One of those unspoken topics is the range of weight that can be applied when re-preforming this experiment.

Theorically speaking, both algorithm are prepared to input graphs containing weights of integer values, but they can be easily updated to use double values or other *Python* data structures as weight. However, the resulting matchings and weights should not behave in the same matter as the ones explored in this study.

Besides that, there're cases where both algorithm wouldn't be able to completely return accurate values. These cases are the following:

- The edges' weight isn't quantifiable.
- The sum of all edges' weight is equal to zero.
- The graph doesn't contain weighted edges.

In the first case, the algorithms would simply broke, because most of their condicional behaviour is based on comparisons using *greater than* and *lesser than* verifications. In the second case, no maximum weighted matching would be produced by the exhaustive search, since the condition for updating that value would never be fulfilled, but the heuristic approach would return a matching based on the order that the edge we added to the graph. In the third case, both algorithm would also broke, because both of them require the graph to have weighted edges.

Besides that, the code to represent the data in the visualizations was produced using "*Matplotlbib*", "*Seaborn*" and "*Pandas*" python's packages and the code iterated thought the results produced by each algorithm according to each input graph.

VII. Conclusions

This report describes two algorithms, an exhaustive search and a greedy heuristic, that were developed in order to solve the maximum weighted problem in graph theory. The problem comes down to finding the matching, i.e., a set of pairwise non-adjacent edges, for which the sum of the weights of its edges is as large as possible in the graph.

It also details the conducted experiments, which resulted in the formal and experimental analysis of both approaches. For the exhaustive approach, it was determined a computational complexity of $O(2^n)$, an expected complexity considering the selected approach. For the greedy heuristic, the analysis resulted in a complexity of $O(n \log n)$, where sorting the graph's edges is the predominant factor. The effectiveness of the greedy heuristic is also explained, regarding the objective behind its usage.

The experimental analysis of the algorithms and the comparison between them presents the trade-off between accuracy and execution time.

References

[1] Wikipedia Contributors, "Maximum weight matching", https://en.wikipedia.org/wiki/Maximum_weight_matching, August 2023.

[2] Huang, Chien-Chung & Kavitha, Telikepalli. (2016). New Algorithms for Maximum Weight Matching and a Decomposition Theorem. Mathematics of Operations Research. 42. 10.1287/moor.2016.0806

[3] Guillaume Ducoffe, Alexandru Popa, The use of a pruned modular decomposition for Maximum Matching algorithms on some graph classes, HAL open science, https://normandie-univ.hal.science/hal-01773568, April 2018

[4] Duan, R., & Pettie, S., Linear-time approximation for maximum weight matching. Journal of the ACM, 61(1), https://web.eecs.umich.edu/~pettie/papers/ApproxMWM-JACM, January 2014

[5] Robert Preis, Linear-time 1/2 Approximation Algorithm for Maximum Weighted Matching in General Graphs, Lecture Notes in Computer Science book series (LNCS, volume 1563), https://link.springer.com/chapter/10.1007/3-540-49116-3_24, January 2002