Assignment 1 - Maximum Weighted Matching

108287 - Miguel Belchior Figueiredo

Abstract – **This report presents a comparative analysis of two algorithms for solving the maximum weighted matching problem: an exhaustive search algorithm and a greedy heuristic approach. Both algorithms are examined through design, formal analysis, and experimental evaluation.**

I. Introduction

The maximum weighted matching problem is an optimization problem in the field of graph theory, with a broad range of practical applications in various fields, such as biotechnology, social analysis, etc. Formally, the problem tries to find for a given undirected graph $G(V, E)$, with n vertices and m edges the maximum weighted matching. A matching in G is a set of pairwise non-adjacent edges, i.e., no two edges share a common vertex. A maximum weighted matching is a matching for which the sum of the weights of its edges is as large as possible.
This is a problem that can be solved in polynomial time $O(V^2E)$ by using the Edmonds' algorithm. However, this report focuses on the design, implementation and analysis of both an exhaustive algorithm and a greedy heuristic approach. It is important to note the distinction between the following greedy strategies as it is a topic that should be reminded throughout the report:

- **Algorithms** which always provide the optimal solution.
- **Heuristics** which in general provide an approximation and can sometimes provide the optimal solution. The final solution will be composed of a sequence of choices of the best looking alternative according to a predefined heuristic. As this choice is locally optimal and irrevocable (there is no backtracking) it is not guaranteed to find the optimal solution. The greedy approach described and analyzed in this report falls under this category.

II. Exhaustive Search Algorithm

A. Design

As the name indicates, an Exhaustive Search Algorithm systematically checks all possible candidates in order to determine whether or not they meet the problem's outlined conditions and find the best solution.

In the maximum weighted matching problem, all possible subsets of edges are evaluated to determine if they form a valid matching. For each valid matching, the corresponding total weight is calculated and the maximum weight matching is updated when it exceeds the previous maximum. As it can be deducted from the problem's description there may be multiple solutions with the same weight for the same graph. Therefore, this algorithm identifies one possible matching with maximum weight, but other algorithms may provide different matchings as long as the respective weight is the same - the algorithm could also be easily changed to track all the possible solutions in a list for instance, as all possible candidates are already evaluated. The algorithm's behavior can be expressed through the following pseudocode:

```
Func ExhaustiveMaxWeightMatching(
G(n,m))
  max_weight_matching ← empty set
  max_weight ← 0

  For each subset size r from 1 to m
    For each subset of edges of size r
      If is_matching()
        Calculate matching weight
        If new weight is a new max
          Set new max_weight_matching
          Set new max_weight
  return max_weight_matching,
max_weight
```

One vital point of the algorithm is the `is_matching` function which checks if a set of edges is a matching by keeping track of all vertices on the subset and checking if the next processed edge has a common vertex. This function's behaviour is defined in the following pseudocode:

```
Func is_matching(matching)
  nodes ← empty set

  For each edge in matching
    u, v, weight ← edge.unpack()
    If u in nodes or v in nodes
      return False
    Update nodes with edges u and v
  return True
```

B. Formal Analysis

As previously mentioned, the exhaustive algorithm is an iterative algorithm that for all possible sizes r, traverses through every subset of edges of size r in G (with minimum and maximum sizes of the subsets being one and m, respectively). Thus, the total number of subsets traversed in this solution is given by the following formula:

$$\sum_{r=1}^{m} \binom{m}{r} = 2^m - 1 \tag{1}$$

Hence, the time complexity of the subset generation and related evaluation is proportional to 2^m, as every subset needs to be verified. For each of these edge subsets, the algorithm checks if corresponds to a valid matching - the `is_matching()` function determines if a subset of edges forms a valid matching by parsing all of its edges vertices until a repeated vertex is found or there are no edges left. In the case that it does, the weight of the matching is calculated in order to verify if there is a new max weight.

Therefore, the running time of the `is_matching()` function significantly impacts the overall complexity of the algorithm and it depends on the specific configuration of the edge subset and whether or not a conflict is detected early of after processing multiple edges:

- **Best Case** - The function returns early as a node is found to be repeated, invalidating the matching. In the best case, that would happen when checking the second edge, as an edge set with just one edge always constitutes a valid matching because there are no other edges to conflict with. Consequently, the complexity for this case is constant: $O(1)$.
- **Worst Case** - The subset of edges is a matching and therefore every edge in the matching will be checked, performing set lookups and updates for each edge. Therefore, the time complexity of the worst case is $O(m)$, where m is the number of edges in the matching.
- **Average Case** - The average case depends on how likely it is for the edges in the possible matching to have repeated nodes.

In conclusion, the total complexity for the exhaustive search algorithm is $O(m \cdot 2^m)$. This is an exponential complexity, dominated by 2^m, which arises from the exhaustive generation of every possible subset of edges. It is also possible to conclude from the complexity, that the running time of the algorithm heavily depends on the number of edges rather than the number of nodes.

C. Experimental Analysis

To validate the results of the formal analysis, the algorithm was applied to randomly generated graphs containing varying numbers of vertices for four different edge densities. The graphs included in the experiments had the following properties:

- successively larger random graphs with $n \in [4, 9]$, n being the number of vertices in the graph. The upper limit for the experiments was defined as it was the higher number of vertices for which the problem could be computationally resolved for the max edge density in a reasonable amount of time (using the exhaustive approach);
- the number of edges of the generated graph corresponds to $\lfloor n \cdot \mathit{edge\_density} \rfloor$, with $\mathit{edge\_density} \in \{0.125, 0.25, 0.5, 0.75\}$ and n being the number of vertices. Only generated graphs with a positive non null number of edges were considered;
- graph vertices are 2D points on the XOY plane, with randomly generated integer valued coordinates between 1 and 1000;
- the number of edges sharing a vertex is randomly determined. However, that is only done after ensuring the connectivity of every single node, i.e., all nodes should have at least one edge, which is also attributed randomly;
- A fixed randomization seed was used to ensure reproducibility of the graph generation process - 108287.

The following metrics were registered for analysis:

- **Number of solutions/configurations tested**, which being an exhaustive algorithm corresponds to all possible combinations of the graph's edges;
- **Number of basic operations**, where the basic operation corresponds to verifying if each node of every edge of a possible matching has already been traversed/processed in the `is_matching` function;
- **Execution Time**;
- **isExpectedResult** - metric that verifies the correctness of the result based on networkX's built in function `max_weight_matching`[1].

An overview of the experimental results is present on Table II but visit the author's github page for the full result[2].

[1]https://networkx.org/documentation/stable/reference/-algorithms/generated/networkx.algorithms.matching.max-_weight_matching.html

[2]https://github.com/Migas77/AA-108287/blob/main/Assignment_1/charts/greedy/results.txt

C.1 Number of Configurations Testes

The number of the subsets tested over the number of edges is plotted in Figure 1. Before starting to analyze the chart result it is important to note:

- The y axis is on a logarithmic scale;
- As previously mentioned, the experiment was conducted on graphs with $n \in [4, 9]$ for each value of $edge\_density \in \{0.125, 0.25, 0.5, 0.75\}$. The x-axis represents the number of edges, calculated as the product of the number of nodes and edge density by the formula $\lfloor n \cdot edge\_density \rfloor$, while the color of the of the points indicates the number of nodes.

Given the behavior of the plotted line begin linear and the scale of the y axis being logarithmic, it is reasonable to infer that the number of configurations tested grows exponentially with the number of edges. It is also possible to conclude that the number of nodes is only relevant in terms of the maximum number of edges it permits, with the number of subsets depending solely on the number of edges of the graph. This conclusion is also supported by the results presented in Table I, where the number of subsets tested for a graph with 5 nodes and 0.75 edge density, as well as for a graph with 6 nodes and 0.5 edge density, is the same, 127, due to their number of edges also being the same, 7. This results in graphs with certain configurations of nodes and edge densities not being represented in the Figure 1, as they end up having the same number of edges, the same number of subsets and consequently, the same axis coordinates - this is precisely what happens to every entry of each graph with 5 nodes as their results are overridden by subsequent experiments.

These results also support the previously done experimental analysis as the collected data points in the experiment perfectly follows, under the logarithmic scale, the function corresponding to the closed formula $2^m - 1$ where m is the number of edges of the graph. That can also be verified by the results presented in Table I.

C.2 Number Of Basic Operations

The number of basic operations by the number of edges is plotted in Figure 2. The graph follows the same conditions as the graph in Figure 1, also having a logarithmic scale on the y axis and the same configuration of nodes and edge densities.

As already presented in Figure 1, in Figure 2 the plotted line also follows a linear pattern with a logarithmic y axis scale. Thus, it is reasonable to infer that the number of basic operations grows exponentially with the number of edges. It is also possible to verify that graphs with the same number of edges possess close values for the operations counter. For instance, the previously mentioned example of the graphs with 5 nodes and 0.75 edge density and the one with 6 nodes and 0.5 edge density carried out 443 and 451 basic operations, respectively - results in Table I. That is not a standalone case has it can be verified through further inspections of both Figure 2 and Table I.

These results also support the identified algorithmic complexity of the exhaustive search algorithm $O(m \cdot 2^m)$, because $bo(m) \leq c \cdot m \cdot 2^m$, for all $m \geq 1$, with a positive constant c.

C.3 Execution Time

The execution time results over the number of edges are plotted in Figure 3. It also possesses a logarithmic scale on the y axis and the same configuration of nodes and edge densities. Apart from the first initial computed values, the graph displays a linear relationship between the execution time and the number of edges. Therefore, it is reasonable to infer that the execution time scales exponentially with the number of edges. As in both previous figures, Figure 1 and Figure 2, it can also be concluded that the execution time depends heavily on the number of edges rather than on the number of nodes.

It is now clear that the algorithm cannot solve this problem for large problem instances within a reasonable amount of time. The experimental computer was unable to solve the maximum weighted matching problem using the exhaustive approach for 10 nodes with an edge density of 0.75, without taking too much time - under 30 minutes. Therefore, it is only possible to make an estimation for the execution time of the algorithm for larger problem instances based on the previously executed problems. Based on the last three entries which have 9 nodes on Table I it is possible to verify that the gap between execution times `t(m)` doubles each time the number of edges is incremented by 9 - which actually is consistent with the previously observed algorithmic complexity:

- $\frac{t(18)}{t(9)} = \frac{9.74e{-2}}{3.82e{-4}} \approx 254.98$
- $\frac{t(27)}{t(18)} = \frac{49.67}{9.74e{-2}} \approx 509.96$

Following this reasoning, the same exhaustive algorithm would compute the result for the max weighted matching problem for graphs with 36, 45 and 54 edges in approximately 50663.4, 103353336 and $4.2e11$ seconds, respectively, which is equivalent to approximately 0.59, 11.96 and 4880574.2 days, respectively - this last execution time being equivalent to approximately 13371.44 years.

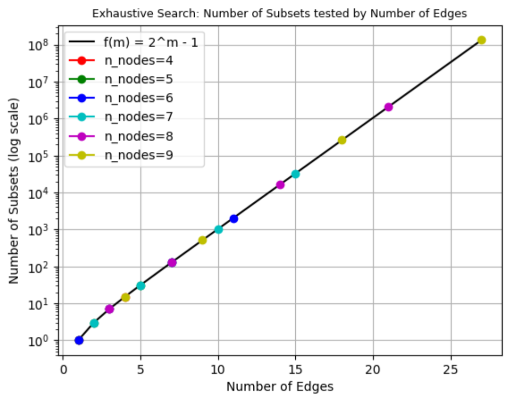

Fig. 1 - Number of Subsets over the Number of Edges - Log. scale on the y axis

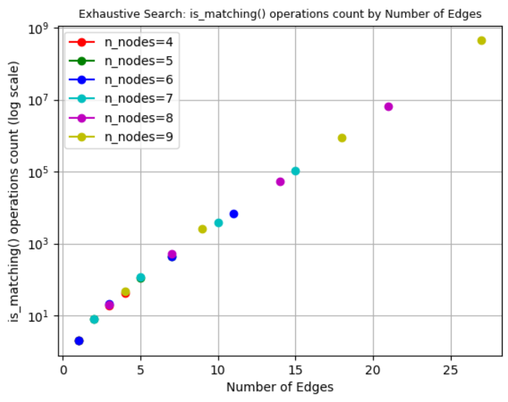

Fig. 2 - Number of Basic Operations over the Number of Edges - Log. scale on the y axis

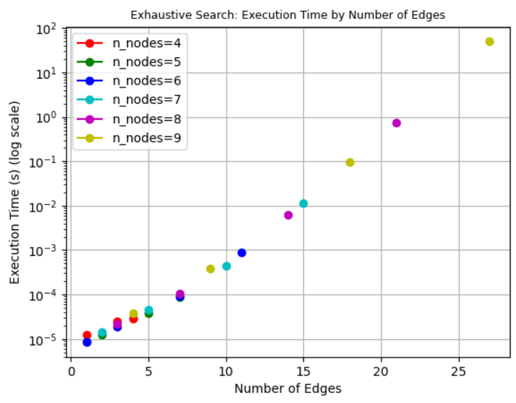

Fig. 3 - Execution Time over the Number of edges - Log. scale on the y axis

TABLE I
Performance Metrics by Nodes and Edge Density of Exhaustive Algorithm

# Nodes	Edge Density	# Edges	#Subsets Tested	#is_matching Operations	Exec. Time (s)	isExp. Result
4	0.25	1	1	2	1.25e-5	True
4	0.5	3	7	19	2.45e-5	True
4	0.75	4	15	43	2.83e-5	True
5	0.125	1	1	2	8.50e-6	True
5	0.25	2	3	8	1.26e-5	True
5	0.5	5	31	109	3.88e-5	True
5	0.75	7	127	443	8.86e-5	True
6	0.125	1	1	2	8.44e-6	True
6	0.25	3	7	21	1.89e-5	True
6	0.5	7	127	451	9.34e-5	True
6	0.75	11	2047	7032	8.75e-4	True
7	0.125	2	3	8	1.46e-5	True
7	0.25	5	31	116	4.64e-5	True
7	0.5	10	1023	3884	4.46e-4	True
7	0.75	15	32767	108011	1.14e-2	True
8	0.125	3	7	20	2.22e-5	True
8	0.25	7	127	512	1.07e-4	True
8	0.5	14	16383	54168	6.23e-3	True
8	0.75	21	2097151	6.65e6	0.742	True
9	0.125	4	15	48	3.78e-5	True
9	0.25	9	511	2607	3.82e-4	True
9	0.5	18	262143	9.07e5	9.74e-2	True
9	0.75	27	1.34e8	4.38e8	49.67	True

III. Greedy Heuristic

A. Design

Following the observed results of both the formal and experimental analysis of the exhaustive search algorithm it is clear that, while it is indeed a straightforward approach to solving the problem and get an optimal solution, its efficiency is low only allowing to solve small-sized instances in a reasonable amount of time. Therefore, searching the optimal solution exhaustively when the space is exponentially proportional to the input size is not feasible for larger problem instances.

That being the case, in this section, a greedy heuristic will be used to construct the final solution. This approach uses a predefined heuristic to make at each step of computing the locally optimal choice that satisfies the problem's constraints. This choice, apart from being locally optimal, it is irrevocable, and consequently, it is not guaranteed to find an optimal solution. On that account, the primary advantage of this algorithm is that it produces approximated results of the globally optimal solution - in some cases it will even reach it - while requiring significantly less time compared to the exhaustive search algorithm.

Three different heuristics were tested throughout this section. However, the algorithm remains the same with its only variation lying in the parameter by which the list of edges is sorted initially. The behavior of the algorithm can be described using the following pseudocode:

```
Func GreedyMaxWeightMatching(
G(n, m),h)
    max_weight_matching ← empty list
    max_weight ← 0
    matched_vertices ← empty set
    sorted_edges ← sorted(G.edges, crt=h)
```

```
For each edge in sorted_edges
    u, v, weight ← edge.unpack()
    If (u not in matched_vertices and
        v not in matched_vertices)
        Add u to matched_vertices set
        Add v to matched_vertices set
        Append edge to max_weight list
        max_weight += weight

return max_weight_matching,
max_weight
```

Firstly, the algorithm initializes all structures and variables, with special emphasis to the *sorted_edges* variable where the graph's edges are sorted according to the predefined heuristic (*crt* stands for criteria). Then, the algorithm iterates through the sorted edges, adding each edge to the matching if neither of its vertices are present in the *matched_vertices* set, i.e., if an edge with common vertices has not yet been added to the matching. That being the case all the variables and in-memory data structures are updated at this point too. In the end, the *max_weight_matching* and *max_weight* are returned.

Now that the algorithm's flow is outlined, it's important to emphasize that the precision of the approximated solutions is largely determined by the chosen heuristic to sort the edges of the graph, which consequently impacts which edges and by which order they are added to the matching. The following three heuristics were tested:

- $\mathbf{G.degree(u) + G.degree(v)}$ - Edges are sorted in **ascending order** of the sum of their vertices degrees. This heuristic prioritizes edges with vertices with lower degrees, effectively favoring edges located in sparser regions of the graph.
- **edge_weight** - Edges are sorted in **descending order** of their weights. This heuristic prioritizes edges with higher weights.
- $\dfrac{\mathbf{edge\_weight}}{\mathbf{G.degree(u) + G.degree(v) + 1}}$ - Edges are sorted in **descending order** according to the predefined formula. This heuristic pretends to find a balance between the two previous heuristics and prioritizes edges that maximize the score, i.e., prioritizes edges with higher weights and with vertices with lower degrees - edges with higher weights located in sparser regions of the graph. This was the heuristic with the best precision value and therefore it's the one that will be used throughout the experimental analysis.

B. Formal Analysis

To conduct a formal analysis, the algorithm can be divided in two sequential phases:

1. The algorithm first sorts the edges of the graph according to the predefined heuristic;
2. It then traverses all sorted edges to construct a matching that maximizes its total weight.

The first phase, is delegated on python's *sorted* function that uses a version of the merge sort algorithm called TimSort, which has a worst and average-case time complexity of $O(m \cdot \log m)$, with m being the number of edges of the graph. The second phase, traverses all sorted edges, checking membership in *matched_vertices* and updating all the necessary variables and structures. As checking membership in a set has an average time complexity of $O(1)$, the entire second phase runs in $O(m)$ time, with m being the number of edges of the graph - there is no variation between best, average or worst cases in this phase of the algorithm because all edges are traversed through, independently of how early the matching is constructed. Therefore, the overall time complexity of the algorithm is $O(m \cdot \log m)$ as for large values of $O(m \cdot \log m)$ dominates $O(m)$. Thus, the time complexity of the algorithm is primarily dictated by the sorting step. This way, the Quasilinear time complexity of the greedy heuristic approach is significantly better than that of the exhaustive search and it will allow to compute results for larger problem instances in a reasonable amount of time.

C. Experimental Analysis

The experimental analysis of the greedy approach follows a similar framework to that of the exhaustive algorithm with the following differences:

- successively larger random graphs with $n \in [4, 1000]$ with a step of 25 and n being the number of vertices in the graph. Unlike the exhaustive algorithm, which had a low upper bound of 9 due to computational constraints, the greedy approach allows for the evaluation of much larger problem instances.

The following metrics were registered for analysis:

- **Number of solutions/configurations tested** - number of edges traversed to obtain the max weighted matching - tracking the number of subsets didn't make sense as there is only one set to track which is the constructed solution;
- **Number of basic operations**, where the basic operation corresponds to verifying whether or not each node of every edge does not belong to the `matched_vertices` set, i.e.,

whether or not each node of every edge is common to other edges in the matching;
- **Execution Time**;
- **Precision** in comparison to the returned weight of the networkX built-in function `max_weight_matching`.

An overview of the experimental results is present on Table II but visit the author's github page for the full result[3].

C.1 Number of Configurations Tested

The number of the subsets tested over the number of edges is plotted in Figure 4. From the graph it is clear that the number of configurations depends linearly on the number of edges. It is also possible to conclude that the number of nodes is only relevant in terms of the maximum number of edges it permits, with the number of configurations depending solely on the number of edges of the graph. This conclusion is also supported by the results presented in Table II, where the number of configurations is always the same as the number of edges of the graph, independently of its number of nodes. These results, as well as those on Table II, also support the identified algorithmic complexity of the second phase of the greedy approach $O(m)$ as $conf(m) \leq c \cdot m$, for all $m \geq 1$, with a positive constant c.

C.2 Number of Basic Operations

The Figure 5 represents a linear relationship between the number of basic operations over the number of edges. The results in Figure 5, as well as those on Table II, are within what's expected as the number of membership checks performed on the `matched_vertices` set is always less than or equal to twice the number of edges (worst case scenario).

C.3 Execution Time

Two graphs were plotted for analyzing the execution time. The Figure 6 plots the execution time over the number of edges. In this figure, it is possible to note an "approximately" linear relationship (Quasilinear) between the execution time and the number of edges. By comparison with the linear relationship on the previous Figure 5 it is possible to denote a differentiation at the beginning of the graph correspondent to the logarithmic component of the algorithms complexity. As the number of edges m grows the execution time of the algorithm is dominated by its linear component. Therefore, the experimental results support the previously identified Quasilinear time complexity $O(m \cdot \log m)$. However, it is possible to identify some outliers that deviate from this pattern. Such deviations are not unusual in greedy algorithms, where the performance can sometimes be affected by specific input configurations. Another thing that is important to note is the difference of the execution times comparatively to the ones of the exhaustive algorithm in Table I, while solving problems with a way higher number of edges. This way, the greedy approach affirms itself as a good alternative to compute solutions for larger problem instances.

The Figure 7 illustrates the execution time over the number of vertices. The y-axis is logarithmic, and each line in the graph represents a fixed edge density for varying number of vertices - note that the legend on the top left corner isn't ordered. As expected, graphs with higher edge densities exhibit longer execution times for the same number of vertices - which is not surprising because graphs with the same number of nodes but with larger edge densities have more edges which is what actually influences the algorithm runtime.

C.4 Precision

As previously mentioned, it was not expected for this algorithm to consistently produce the optimal solution. However, since maximum weighted matching is an optimization problem, the algorithm's effectiveness is measured by the precision of its solutions while minimizing computation time and resource usage. In the last three columns of Table II, are presented the expected max weight (using networkX built-in function), the actual max weight computed with the greedy strategy and the computed precision which corresponds to:

$$P = \frac{actual\_max\_weight}{expected\_max\_weight} \qquad (2)$$

By analyzing the results, it is possible to conclude that although the algorithm fails to achieve the optimal solutions - which only happens for small problem instances - it achieves a very high average precision of 0.975. Moreover, precision values rarely drop below 0.9 and show no tendency to decline as the number of edges increases.

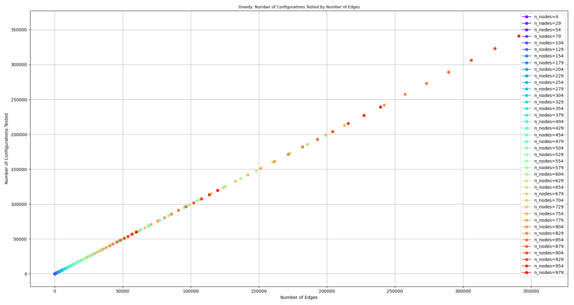

Fig. 4 - Number of Configurations Tested over the Number of Edges

[3]https://github.com/Migas77/AA-108287/blob/main/Assignment_1/charts/greedy/results.txt

TABLE II
Performance Metrics by Nodes and Edge Density

#Nodes	Edge Density	#Edges	#Configs Tested	#Basic Ops.	Exec. Time (s)	Precision
4	0.25	1	1	2	1.1422e-05	1
4	0.5	3	3	6	1.3941e-05	1
4	0.75	4	4	6	1.3592e-05	1
29	0.125	50	50	77	5.9877e-05	0.986486
29	0.25	101	101	135	0.000104842	0.918594
29	0.5	203	203	228	0.000195823	0.976397
29	0.75	304	304	345	0.000289416	0.958841
54	0.125	178	178	223	0.000185933	0.969928
54	0.25	357	357	427	0.000347828	0.97236
54	0.5	715	715	793	0.000679853	0.96623
54	0.75	1073	1073	1188	0.00106037	0.959246
79	0.125	385	385	488	0.00039669	0.913252
79	0.25	770	770	896	0.000858791	0.93071
79	0.5	1540	1540	1690	0.00155598	0.968808
79	0.75	2310	2310	2573	0.00265455	0.96043
104	0.125	669	669	802	0.000682804	0.930855
104	0.25	1339	1339	1511	0.00124989	0.937448
104	0.5	2678	2678	2873	0.002667	0.964968
104	0.75	4017	4017	4257	0.00386266	0.975323
129	0.125	1032	1032	1211	0.00110937	0.944852
129	0.25	2064	2064	2326	0.0020688	0.956827
129	0.5	4128	4128	4497	0.00462546	0.958814
129	0.75	6192	6192	6572	0.00630652	0.977455
154	0.125	1472	1472	1677	0.00143054	0.942136
154	0.25	2945	2945	3260	0.00287614	0.958143
154	0.5	5890	5890	6338	0.00600718	0.967137
154	0.75	8835	8835	9376	0.00903764	0.973522
179	0.125	1991	1991	2352	0.0020097	0.932388
179	0.25	3982	3982	4351	0.00417005	0.972442
179	0.5	7965	7965	8396	0.00833346	0.977568
179	0.75	11948	11948	12504	0.0126647	0.981345
204	0.125	2588	2588	2953	0.00269694	0.935954
204	0.25	5176	5176	5616	0.00561512	0.96782
204	0.5	10353	10353	10882	0.0108495	0.977813
204	0.75	15529	15529	16017	0.016366	0.980288
229	0.125	3263	3263	3691	0.00354944	0.946972
229	0.25	6526	6526	6988	0.00676702	0.965935
229	0.5	13053	13053	13818	0.0139115	0.973217
229	0.75	19579	19579	20397	0.0210049	0.983545
254	0.125	4016	4016	4518	0.00407519	0.951789
254	0.25	8032	8032	8661	0.00850696	0.959383
254	0.5	16065	16065	16835	0.0170738	0.979894
254	0.75	24098	24098	24906	0.0257753	0.979829

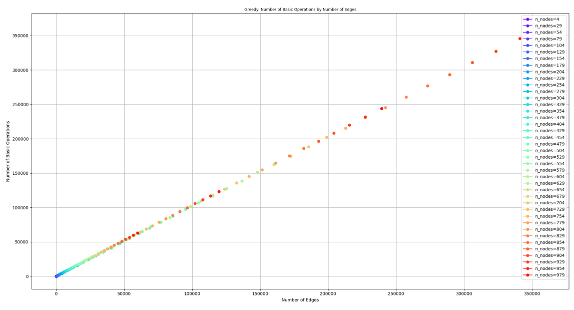

Fig. 5 - Number of Basic Operations over the Number of Edges

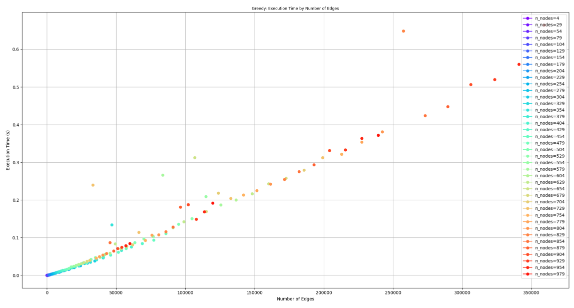

Fig. 6 - Execution Time over the Number of Edges

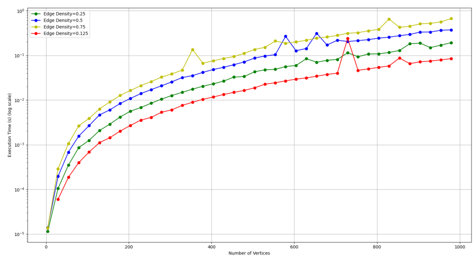

Fig. 7 - Execution Time over the Number of Vertices - Log. scale on the y axis

IV. Conclusion

This report, presented a comparative analysis of two algorithms designed to solve the maximum weighted matching problem: the exhaustive search algorithm and a greedy heuristic approach. Both approaches were reviewed in terms of their time complexity, with both design, formal and experimental analysis being performed.

The exhaustive search algorithm, while guaranteeing an optimal solution it demonstrated to be significantly inefficient due to its exponential time complexity: $O(m \cdot 2^m)$. It's performance was shown to be heavily constrained by the number of edges of the graph (and not by its number of edges), making it impractical for larger problem instances. The followed experimental analysis validated the previous theoretical assumptions, being highlighted its exponential growth in execution time, number of subsets tested, and basic operations performed.

The greedy heuristic approach, while not guaranteeing an optional solution, proved to be a more scalable alternative for larger problem instances with its quasilinear time complexity: $O(m \cdot \log m)$, derived from the TimSort algorithm used for sorting the graph's edges according to the predefined heuristic. Three different heuristics were tested, being the heuristic that balanced edge weight and vertex degree the one that computed the results with the best precision. The experimental results also showed the greedy approach's linear growth in configurations tested and operations performed, and that the algorithm consistently delivered high precision solutions in a fraction of the time required by the exhaustive method. Therefore, it provides a good balance between solution quality and computational efficiency making it viable for large problem instances.

References

[1] Wikipedia contributors, "Maximum weight matching — Wikipedia, the free encyclopedia", `https://en.wikipedia.org/wiki/Maximum_weight_matching`, 2023, [Online; accessed 17-November-2024].

[2] GeeksforGeeks contributors, "Timsort", `https://www.geeksforgeeks.org/timsort/`, 2024, [Online; accessed 17-November-2024].

[3] NetworkX contributors, "Networkx documentation", `https://networkx.org/documentation/stable/reference/index.html`, 2024, [Online; accessed 17-November-2024].