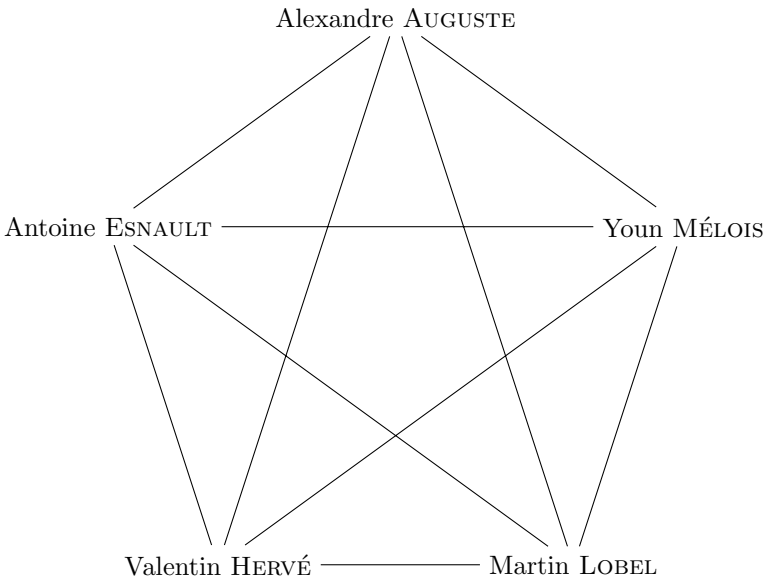
ISEN ALL IS DIGITAL! NANTES | yncréa

Final Project
The Maximum Edge Weight Clique Problem



Abstract
The Maximum Edge Weight Clique (MEWC) problem is an optimization problem in graph theory that asks for the clique (a subset of vertices, all adjacent to one another) with the maximum total weight in an edge-weighted undirected graph. In the MEWC problem, each edge has a weight, and the weight of a clique is the sum of the weights of its edges. The goal is to find a clique with the maximum possible weight.

Class group:
CIR3 - Team 1

Teacher:
Leandro Montero

January 23, 2023

Contents

1 Introduction

1.1 Presentation

Graph theory is a branch of mathematics that deals with the study of graphs, which are mathematical structures used to model pairwise relationships between objects. Graphs consist of vertices (also called nodes) that are connected by edges. The edges can be either directed (one-way) or undirected (two-way) and can also have a weight.

The Maximum Edge Weight Clique (MEWC) problem is an optimization problem in graph theory that asks for the clique (a subset of vertices, all adjacent to one another) with the maximum total weight in an edge-weighted undirected graph. In the MEWC problem, each edge has a weight, and the weight of a clique is the sum of the weights of its edges. The goal is to find a clique with the maximum possible weight.

Now, the MEWC problem is **NP-hard**, which means that it is not possible to find an efficient algorithm to solve it in polynomial time or that this problem is at least as hard as the hardest problems in NP. It is also the generalization of the Maximum Clique Problem (MCP), which is the special case where all edges have the same weight.

For example, the following graph $G = (V, E)$ has for its set of vertices $V = \{1, 2, 3, 4, 5, 6\}$ and for its set of edges $E = \{(1, 2), (1, 5), (2, 3), (2, 5), (3, 4), (4, 5), (4, 6)\}$. As we can see, the red edges $\{(1, 2), (1, 5), (2, 5)\}$ form a clique of size 3 and the other colored edges are each clique of size 1. We can also easily deduct that the maximum clique of G is the red clique of size 3.

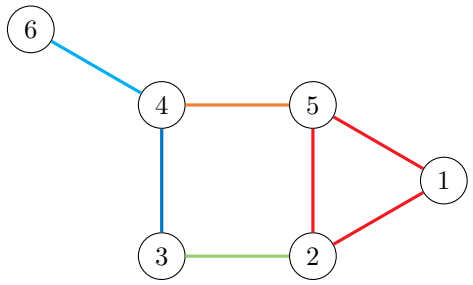


Figure 1: Basic graph example

The MEWC problem can be used to model various types of real-world situations where the goal is to find a subset of objects with the maximum total weight, and the objects are connected by weighted edges. Here are a few examples of such situations:

- **Network design**: In a communication network, the MEWC problem can be used to find the optimal subset of devices (vertices) to include in the network, such that the total cost of communication between the devices (edges) is maximum.
- **Protein interaction**: In biology, the MEWC problem can be used to find the optimal subset of proteins (vertices) in a protein-protein interaction network, such that the total interaction strength (edges) between the proteins is maximum.
- **Social network analysis**: In a social network, the MEWC problem can be used to find the optimal subset of individuals (vertices) with the maximum total relationship strength (edges) between them.

1.2 Configuration

For this project we used **C++** to develop and implement our algorithms. It's a programming language that has been around for a while and is still popular today. It's a compiled language, which means that it gets translated into machine code before it runs, making it very fast and efficient. Additionally, C++ is a statically typed language, requiring you to specify the data type of a variable before you use it. This allows the compiler to check the type of each variable and to detect errors during the compilation.

Though C++ is a very powerful language, it is also very complex and has a lot of features. This can make it difficult to use. That's why we had a debate about which language to use for this project. Our choices were C++ and Python. We eventually chose C++ because it is a language that we are all familiar with and that we have all used before. We also chose it because it is a very powerful language that allows us to implement very efficient algorithms.

The **Standard Template Library** (STL) is a library of C++ that contains a lot of useful classes and functions. It is a very powerful tool that allows programmers to work with dynamic data structures without having to implement them themselves, and having to deal with memory management.

To have the best possible performance, we chose to use `unordered_set` and `unordered_map` to store the vertices and edges of the graph. These data structures are implemented using hash tables, which allows them to have constant time complexity for insertion, deletion and search.

To be able to work on the project efficiently and to be able to share the code between us, we used **GitHub**[1]. It is a web-based platform for version control, collaboration, and sharing of code, as well as a community of developers who contribute to open source projects and share their knowledge. It was a tool that was difficult for some to get used to quickly, especially on the configuration of the project at home, but which brought us a significant gain in efficiency once we had understood how to use it. And to share information and communicate between us, we used **Discord**.

To run our algorithms and to test them, we used a virtual server with **Debian** as an operating system. The CPU is an **AMD EPYC 7282** at 2.8 GHz and with 8 GB of RAM. Although not as powerful as some personal computers, it is still powerful enough to run our algorithms in a reasonable amount of time and to perform the tests we needed homogeneously in the background.

1.3 Example of real-life situations

As we said before, the MEWC has many real-world applications in various fields such as social networks, chemistry, bioinformatics. Now, we will give a concrete example of a real-life situation that can be modelled as a MEWC problem.

The team formation process for a project, or for the search for a particular social group, is a situation that can be viewed as a MEWC problem. Let's imagine that the Student Office

[1] `https://github.com/sehnryr/Final-Graph-Project-ISEN-CIR3`

of ISEN Nantes is looking to reinforce the video games club of its school. Indeed, the latter has no succession for the following year and is thus led to die if no member presents himself. The future members of the office will have to be in contact with each other during a whole year, and it is thus important to find people with common interests so that no tension is formed during their studies. The office has access to the Steam profiles of the students within ISEN (Steam is a video game digital distribution service that gives information about the games played by each one) as well as a record made by the gaming club of the different games played by their members. The fact of playing games in common could bring some people closer, and this makes it a good criterion to create a group that could take over the club because it would share common interests.

To model this problem, we can represent each student as a vertex and each the games played in common as an edge. The weight of each edge will be the number of games played in common. The goal of the office is to find a group of students that will take over the club. To do this, we will use the MEWC algorithm to find a group of students that has the highest affinity.

In the following example, we will restrict ourselves to 9 students of the CIR3 class of ISEN Nantes since it would be too complicated to represent every student of the school. We will also assume that the students have played the game listed in the table below.

Students	**Game played**
Youn	Minecraft, Civilization, Lost ARK, Among US
Martin	Minecraft
Valentin	Genshin, Minecraft, Civilization
Bastien	Genshin, Minecraft, Lost ARK, Among US
Guillaume	CSGO, Genshin, Overwatch, Stardew Valley
Dorian	CSGO, Paladins, Overwatch
Antoine	League of Legends, Stardew Valley
Thomas	League of Legends, The Last of US
Alexandre	League of Legends, Dofus

Which would give us this graph :

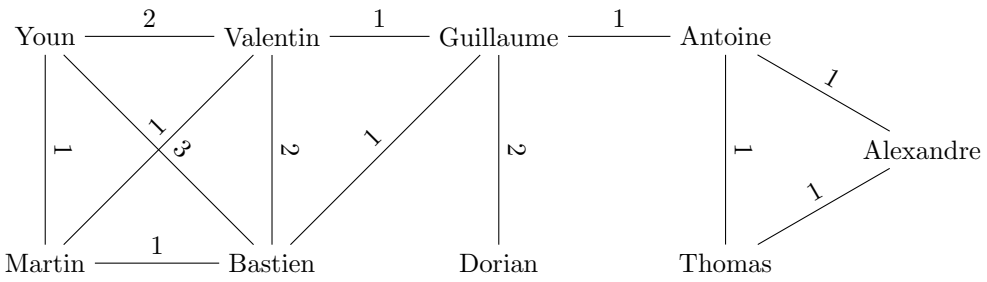


Figure 2: Graph representing the games played between the students

The goal of the Maximum Edge Weight Clique problem in this context would be to find a complete subset of individuals such that the sum of the weights of the edges between the individuals is maximized. In this example, the maximum weight clique would be the clique

consisting of nodes Youn, Valentin, Martin, Bastien with a total weight of $2+1+1+1+3+2 = 10$.

2 Exact Algorithm

2.1 Presentation

A naive approach to the exact algorithm for solving the MEWC problem works by exploring all potential cliques in the graph and selecting the clique with the maximum weight. To do this, the algorithm uses a recursive function to explore all possible subsets of V. For each subset, the algorithm computes the weight of the clique formed by the vertices in the subset. If the weight is greater than the current maximum weight, the clique is selected as the current maximum. This process is repeated until all possible cliques have been explored, at which point the algorithm returns the clique the maximum weight.

The time complexity of this approach is $\mathcal{O}(n^2 \times 2^n)$, where n is the number of vertices in the graph. This is due to the fact that there are 2^n possible subsets of V, and the algorithm must compute the weight of the clique, which by itself has a complexity of $\mathcal{O}(n^2)$ because there are at most $\frac{n(n-1)}{2}$ edges in a complete graph, and compare it to the current maximum weight. Therefore, the algorithm is only feasible for small-scale graphs.

However, we could only look at the maximal cliques of the graph, which would reduce the number of subsets to explore. A clique is maximal if it is not a subgraph of any other clique. To find all maximal cliques in a graph, we can use the **Bron-Kerbosch** algorithm[1]. This algorithm has a time complexity of $\mathcal{O}(\sqrt[3]{3}^n)$, where n is the number of vertices in the graph. This is optimal, as it has been proven by J. W. Moon & L. Moser in 1965[2] that there are at most $\sqrt[3]{3}^n$ maximal cliques in any n-vertex graph and $\sqrt[3]{3} \approx 1.44 < 2$.

We can use the Bron-Kerbosch algorithm to find all maximal cliques in the graph, and then compute the weight of each clique. This approach has a time complexity of $\mathcal{O}(n^2\sqrt[3]{3}^n)$, which is much better than the previous approach. However, this algorithm is still not feasible for large-scale graphs.

2.2 How it works

As said before, our algorithm first uses the Bron-Kerbosch algorithm to obtain all the maximal cliques of the input graph. Then, it iterates through those maximal cliques to looks for which cliques have the highest weight.

The Bron-Kerbosch pivot algorithm that we use is a more efficient variant of the initial algorithm. The basic form of the algorithm is inefficient in the case of graphs with many non-maximal cliques as it makes a recursive call for every clique, maximal or not. To save time and allow the algorithm to backtrack more quickly in branches of the search that contain no maximal cliques, Bron and Kerbosch introduced a variant of the algorithm involving a "pivot vertex".

At each step, the algorithm keeps track of three groups of vertices: R which is a partially constructed (non-maximal) clique, P which is the candidates vertices that could be included in the clique and X which is the excluded vertices that already have been searched (because doing so would lead to a clique that has already been found). The algorithm tries adding the candidate vertices one by one to the partial clique, making a recursive call for each one.

After trying each of these vertices, it moves it to the set of vertices that should not be added again. At each recursion, P and X are restricted to the neighbors of current vertex being added to R and when P and X are both empty there are no further elements that can be added to R, R is a maximal clique and the algorithm reports R.

The recursion is initiated by setting R and X to be the empty set and P to be the vertex set of the graph. Within each recursive call, the algorithm considers the vertices P in turn; if there are no such vertices, it either reports R as a maximal clique if X is also empty, or continue. Then, a pivot vertex u is chosen from $P \cup X$ since any maximal clique must include either u or one of its non-neighbors, for otherwise the clique could be augmented by adding u to it. Only u and its non-neighbors needs to be tested as the choices for the vertex v that is added to R in each recursive call to the algorithm. For each vertex v chosen from $P \backslash N(u)$, with $N(u)$ being the neighbor set of u, it makes a recursive call in which v is added to R and in which P and X are restricted to $N(v)$, which finds and reports all cliques extensions of R that contains v. Then, it moves v from P to X to exclude it from consideration in future cliques and continues with the next vertex in $P \backslash N(u)$.

To illustrate the Bron-Kerbosch algorithm, let's use the example in Figure 1 on page 3:

Step 0:

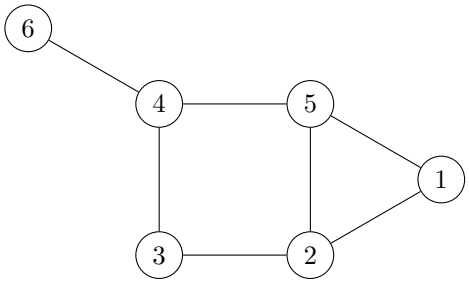


Figure 3: Graph illustration for the exact algorithm at step 0

At the initial step, as said before, we will initialize R and X to be the empty set and P to be the set of vertices of the graph.

R = {}	P = {1,2,3,4,5,6}	X = {}

Step 1:

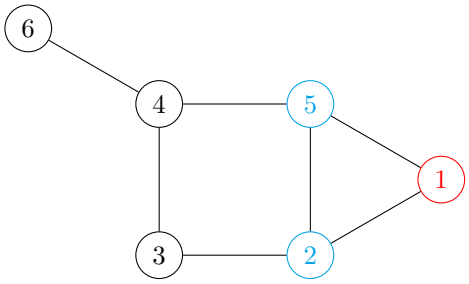


Figure 4: Graph illustration for the exact algorithm at step 1

In the first call of the function, since P is not empty, we chose a pivot vertex u in $P \cup X$. As it would be more time-consuming to search for an appropriate pivot vertex, we simply chose the first one available, which will be represented in red. In this example we will always take the first element of P. The neighboring vertices of u are represented in blue.

After that, we will iterate over the vertices of $P \backslash N(u)$, or in this case, $\{1, 3, 4, 6\}$, and for each vertex v we will make a recursive call to the function with v added to R and P and X restricted to $N(v)$, which will find and report all cliques extensions of R that contains v.

Step 2:

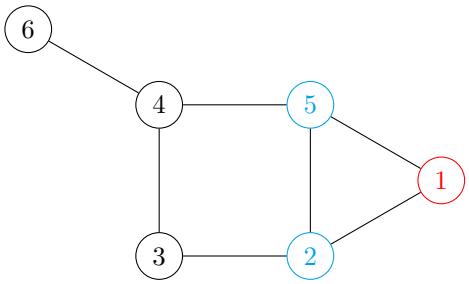


Figure 5: Graph illustration for the exact algorithm at step 2

At this step, since P is not empty, we will choose a pivot vertex u in $P \cup X$. Here $u = 2$ and the algorithm will iterate over the vertices of $P \backslash N(u)$, which is $\{2\}$.

The next recursive call will be made with $P = \{5\}$, the pivot vertex will be $u = 5$ and then the next recursive call will report the clique $\{1, 2, 5\}$, which will be added to the list of cliques.

After that, the algorithm will backtrack to $R = \{\}$ since there will be no more iteration to do, and will add 1 to X as it was visited in its entirety.

The recursive call process looks like this:

R = {1}	P = {2,5}	X = {}
R = {1,2}	P = {5}	X = {}
R = {1,2,5}	P = {}	X = {}

Step 3:

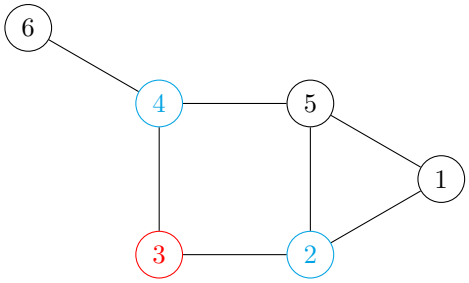


Figure 6: Graph illustration for the exact algorithm at step 3

At this step, since P is not empty, we will choose a pivot vertex u in $P \cup X$. Here $u = 2$ and the algorithm will iterate over the vertices of $P \backslash N(u)$, which is $\{2, 4\}$.

The next recursive calls will be made with $P = \{\}$ and $X = \{\}$, so they will report the cliques $\{2, 3\}$ and $\{3, 4\}$, which will be added to the list of cliques.

After that, the algorithm will backtrack to $R = \{\}$ since there will be no more iteration to do, and will add 3 to X as it was visited in its entirety.

The recursive call process looks like this:

R = {3}	P = {2,4}	X = {}
R = {2,3}	P = {}	X = {}
R = {3,4}	P = {}	X = {}

Step 4:

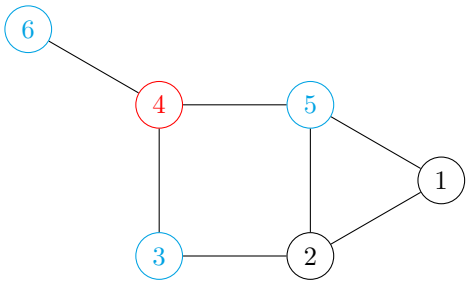


Figure 7: Graph illustration for the exact algorithm at step 4

At this step, since P is not empty, we will choose a pivot vertex u in $P \cup X$. Here $u = 3$ and the algorithm will iterate over the vertices of $P \backslash N(u)$, which is $\{5, 6\}$.

The next recursive calls will be made with $P = \{\}$ and $X = \{\}$, so they will report the cliques $\{4, 5\}$ and $\{4, 6\}$, which will be added to the list of cliques.

After that, the algorithm will backtrack to $R = \{\}$ since there will be no more iteration to do, and will add 4 to X as it was visited in its entirety.

The recursive call process looks like this:

R = {4}	P = {5,6}	X = {3}
R = {4,5}	P = {}	X = {}
R = {4,6}	P = {}	X = {}

Step 5:

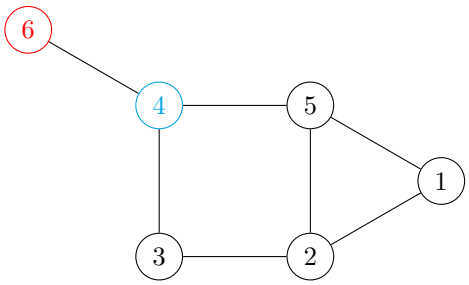


Figure 8: Graph illustration for the exact algorithm at step 5

At this step, since P is empty and X is not, the algorithm will choose a pivot vertex u in X. Here $u = 4$ but the algorithm will not be able to iterate further as $P \backslash N(4) = \{\}$.

The recursive call process looks like this:

R = {6}	P = {}	X = {4}

The Bron-Kerbosch algorithm is now finished, and we have obtained the following maximal cliques:

$$\{(1, 2, 5), (2, 3), (3, 4), (4, 5), (4, 6)\}$$

Now that we obtained the maximal cliques of the graph, we can compare them to find the maximal edge-weighted clique.

We will simply iterate over the maximal cliques and sum the weights of the edges that are in the clique. The clique with the highest weight will be the maximal edge-weighted clique.

If we want to explain everything in a more concise way, we can use the call process and tree of the algorithm.

```
BronKerbosch(∅, {1, 2, 3, 4, 5, 6}, ∅)
    BronKerbosch(1, {2, 5}, ∅)
        BronKerbosch(1, 2, {5}, ∅)
            BronKerbosch(1, 2, 5, ∅, ∅) : report {1, 2, 5}
    BronKerbosch(3, {2, 4}, ∅)
        BronKerbosch(2, 3, ∅, ∅) : report {2, 3}
        BronKerbosch(2, 4, ∅, ∅) : report {2, 4}
    BronKerbosch(4, {5, 6}, 3)
        BronKerbosch(4, 5, ∅, ∅) : report {4, 5}
        BronKerbosch(4, 6, ∅, ∅) : report {4, 6}
    BronKerbosch(6, ∅, 4)
```

Figure 9: Call process of the Bron-Kerbosch algorithm

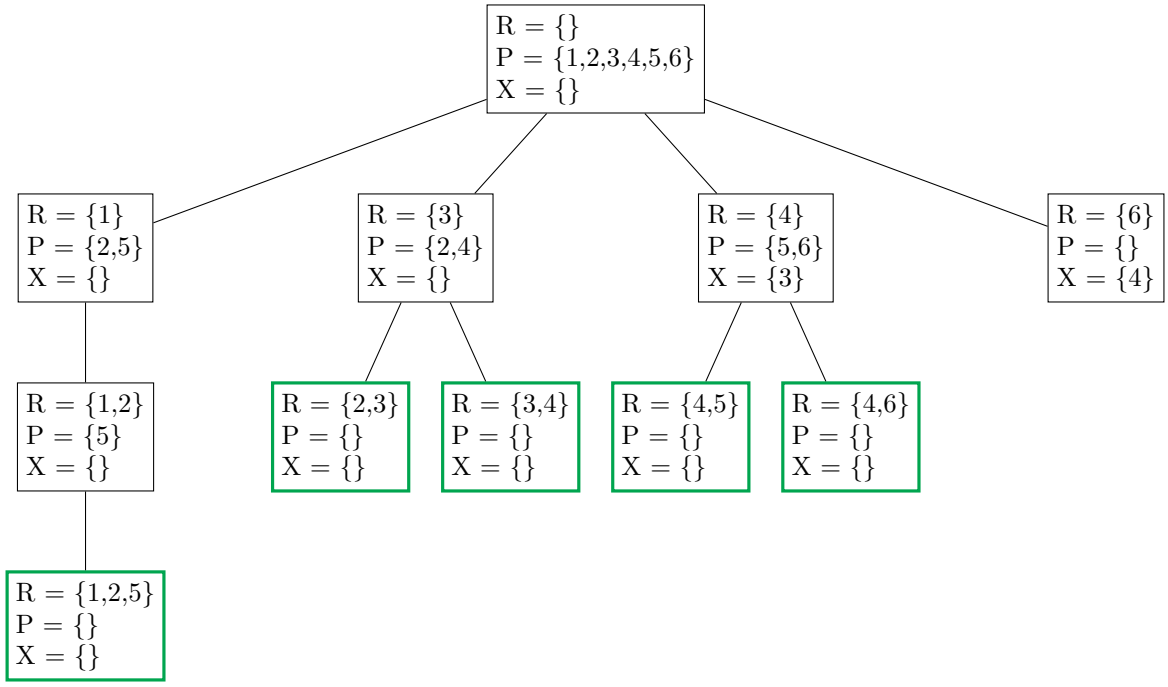


Figure 10: Call tree of the Bron-Kerbosch algorithm

2.3 Pseudocode

Algorithm 1 Exact MEWC algorithm

```
1: procedure ExactMEWC(G = (V, E))
2:     maxClique ← ∅                          ▷ Variable to store the maximum weight clique
3:     maxCliqueWeight ← 0
4:     R ← ∅                                  ▷ Set that will contain the built clique in the recursive calls
5:     P ← V                                  ▷ Candidate vertices
6:     X ← ∅                                  ▷ Excluded vertices
7:     maximalCliques ← ∅                     ▷ The set that will contain the maximal cliques of the graph
8:     BronKerbosch(R,P,X,maximalCliques) ▷ Call the BronKerbosch algorithm to get the maximal
       cliques of the graph
9:     for all clique ∈ maximalCliques do
10:        weight ← getCliqueWeight(clique)   ▷ Get the weight of the current clique
11:        if weight > maxCliqueWeight then
12:            maxClique ← clique
13:            maxCliqueWeight ← maxClique
14:    return maxClique
```

Algorithm 2 Bron-Kerbosch algorithm

```
1: procedure BronKerbosch(R,P,X,maximalCliques)
2:     if P = ∅ & X = ∅ then
3:         maximalCliques ← maximalCliques ∪ {R} ▷ If P and X are both empty sets, R is a maximal
           clique of the graph
4:     u ∈ P ∪ X                              ▷ Chose a pivot vertex from P ∪ X
5:     for all v ∈ P\N(u) do
6:         BronKerbosch(N ∪ {v},P ∩ N(v),X ∩ N(v))  ▷ Make recursive call with reduced candidate set
7:         P ← P\{v}
8:         X ← X ∪ {v}
```

Algorithm 3 Clique weight function

```
1: procedure getCliqueWeight(C)               ▷ We assume that the clique is complete
2:     weight ← 0
3:     n ← |C|
4:     for all i ∈ {1, 2, ..., n} do          ▷ For each vertex in the clique
5:         for all j ∈ {i + 1, i + 2, ..., n} do   ▷ For each other vertex in the clique
6:             weight ← weight + w_ij         ▷ Add the weight of the edge between the two vertices
7:     return weight
```

2.4 Complexity

Let be a graph $G = (V, E)$ such that $n = |V|$, we can now calculate the complexity of our algorithm. The cost of attributing a value to a variable should always be $\mathcal{O}(1)$.

The worst case complexity of the Bron-Kerbosch algorithm is $\mathcal{O}(\sqrt[3]{3}^n)$, we will not prove it here, but it is a well-known fact proven by Moon & Moser [2] that there are at most $\sqrt[3]{3}^n$ maximal cliques in any n-vertex graph.

The complexity of the weight calculation is $\mathcal{O}(n^2)$, because we need to iterate two times over the vertices of the clique to get every edge and by extension their cumulative weight.

The worst case complexity of the algorithm is therefore $\mathcal{O}(n^2\sqrt[3]{3}^n)$, as we need to iterate through the computed maximal cliques by the Bron-Kerbosch algorithm and calculate their weight.

Now, the worst case complexity is not the average complexity of the algorithm. Realistically, the complexity will vary depending on the connectivity and degeneracy of the graph. Thought calculating its average complexity is not trivial, we can still give a lower bound of the complexity.

Seeing how the algorithm works by excluding the neighboring vertices of the pivot vertex from the iteration, the lower bound of the complexity will be the special case where the graph is empty with no edges connecting the vertices. We can calculate that complexity by using the following formula:

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ T(n-1) + n & \text{if } n > 0 \end{cases} \tag{1}$$

Base case: $T(0) = 1$
Inductive step: $T(k) = \frac{1}{2}(k^2 + k + 2)$

$$\begin{aligned} T(k+1) &= \frac{1}{2}((k+1)^2 + (k+1) + 2) && (2)\\ &= \frac{1}{2}(k^2 + 2k + 1 + k + 3) && (3)\\ &= \frac{1}{2}(k^2 + k + 2) + \frac{1}{2}(2k + 2) && (4)\\ &= T(k) + (k+1) && (5) \end{aligned}$$

Since we proved that $T(n) = \frac{1}{2}(n^2 + n + 2)$, we can conclude that the complexity of the lower bound of the algorithm is $\mathcal{O}(n^2)$.

Also, to match this theoretical complexity, we need to use efficient data structures with constant insertion, deletion and lookup time.

2.5 Bad Instance

The exact algorithm cannot have a bad instance, because it will always find the optimal solution as it checks every possible maximal clique. However, it is important to note that the algorithm is not efficient for large graphs, as it will require super polynomial time to run.

2.6 Experiments

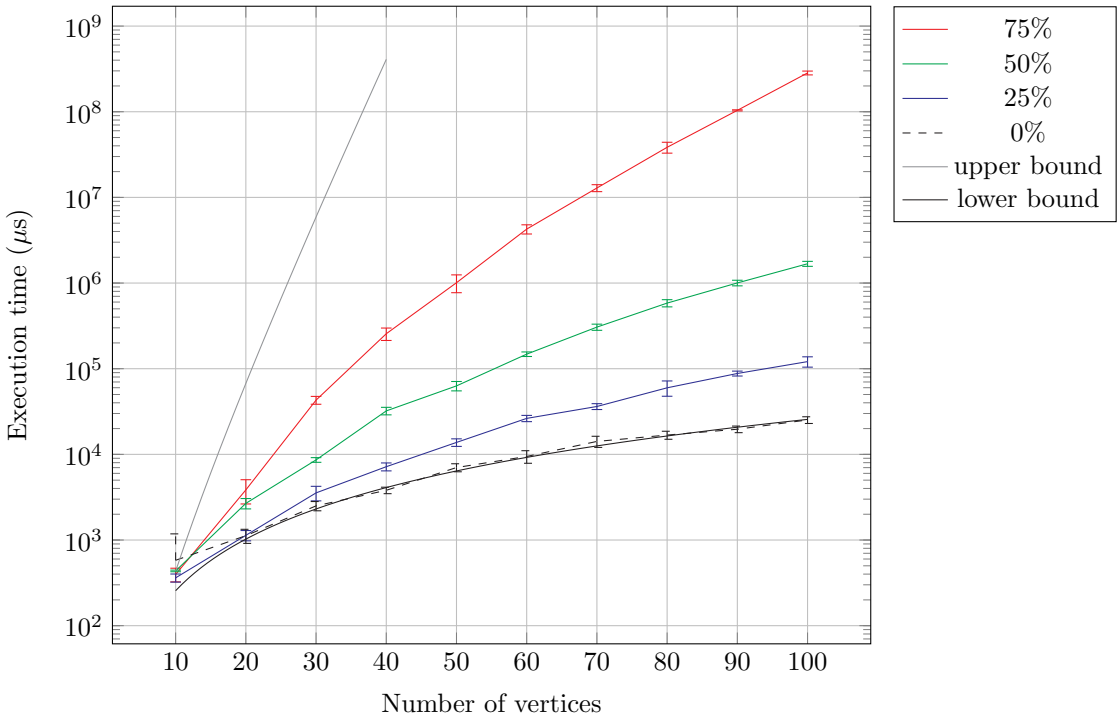


Figure 11: Execution time of the exact algorithm for different percentages of connectivity.

For this experiment, we generated five random graphs with 10 to 100 vertices to find an average execution time for each percentage of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 11. At each number of vertices, an average execution time is calculated, and the standard deviation is represented by the error bars. The number of graphs generated for each percentage of connectivity and number of vertices is limited to 5 to reduce the execution time of the experiment which already takes more than 4 minutes to compute only one result for the 100 vertices case for the 75% connectivity.

2.7 Analysis

In figure 11, we can observe the exponential increase in the execution time of the exact algorithm with the number of vertices for different percentages of connectivity. Note that the upper bound appears as a straight line because of the logarithmic scale but is equal to $n^2\sqrt[3]{3}^n$. The lower bound is matching the 0% connectivity with a constant factor of $2.9 \pm 0.09\%$.

Increasing the connectivity of the graph has a chance to increase the complexity of the structure of that graph, which in turn increases the time complexity of the algorithm. The upper bound is the worst case scenario, where the graph maximizes the number of maximal cliques. The lower bound is the best case scenario, where the graph is empty.

So, the exact algorithm is good if we want to find the exact solution for a few vertices. However, if we were to solve this problem for, let's say, 500 vertices, the exact algorithm

would take 8 months to finish and find the solution. And that is only for 75% connectivity. If we were to account for the worst case scenario, the algorithm would take two hundred duovigintillion years ($2 \cdot 10^{71}$) to finish. We can easily conclude that the exact algorithm is not a viable option for scalability.

3 Constructive Algorithm

3.1 Presentation

A **heuristic algorithm** is a type of algorithm that uses criteria to try to find an approximate solution to a problem, rather than an exact one. Heuristic algorithms are often used when it is not possible to find an ideal solution to a problem, or when a perfect solution would be too time-consuming to compute. They are also used when an approximate solution is good enough. Heuristic algorithms are commonly used in artificial intelligence, computer science, and other fields. They are often used to solve optimization problems, search problems, and other types of problems where an exact solution is not necessary or practical.

A **constructive heuristic** algorithm is a type of heuristic algorithm that is used to find a solution to a problem by building it incrementally. Constructive heuristics start with a partial solution and gradually improve it until a complete solution is found.

To solve the MEWC with a constructive algorithm, we will need some criteria :

1. Add a first Vertex to make a partial solution.
2. Seek for one of his neighbors. And add it to solution if it's a neighbor of all vertices in solution(to be sure to hold a clique in the final result).
3. Repeat the step 2 until no vertices is available to add
4. Calculate the weight of the clique found

We will then have for solution a maximum clique with his weight.

It is important to define the different criteria in a rigorous way.

The first one is the choice of the first Vertex. Indeed, it is this one which will define the quality of our solution. Taking it randomly would be useless and counterproductive for the sake of solving MEWC or for any other problem. There is a lot of implementation which depends on its use. In our project, as it was important, we thought about how to choose it and 2 answers appeared to us, and we have a debate on the subject because we could not reach a consensus on it. We hesitated between these two solutions:

- The first idea was to take the highest degree vertex of the graph given as input.
  - One reason to choose the highest degree vertex as the first vertex in the solution is that it may be more likely to be part of a maximum edge weight clique(because it's the case where it's the most likely to be member of the biggest clique who could be the MEWC). This is because it will allow more edges to be added to the solution, which can increase the overall sum of edge weights in the clique.
  - Another reason is that it may be more likely to be connected to other high degree vertices. This means that by adding the highest degree vertex to the clique first, we may be able to include other high degree vertices in the clique as well, which can further increase the overall sum of edge weights.
- The second one was to take the vertex with the highest sum of weights of the edges.

– One reason to choose the vertex with the highest edge weight as the first vertex in the clique is that it may be more likely to be part of a maximum edge weight clique(because it's the case where it's the most likely to be member of the most weighted clique who could be the MEWC). This is because adding a vertex with a high edge weight to the clique will contribute more to the overall sum of edge weights in the clique, which is what we are trying to maximize.
– Moreover, taking a vertex with higher degree can find adjacent vertices that have lower weight edges, which decreases the probability of finding a maximum weight clique. This does not happen with this choice.

We will illustrate these explanations with figure 12 and 13 which shows the advantages of each over the other.

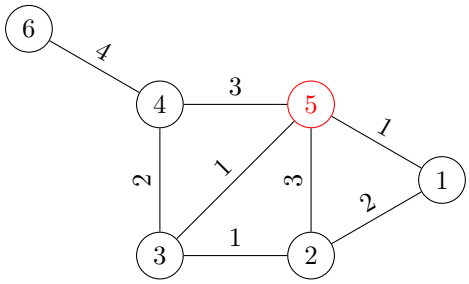


Figure 12: Graph illustration for the highest degree vertex

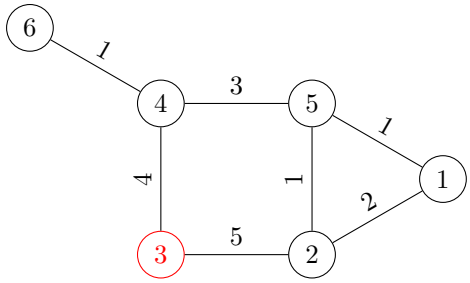


Figure 13: Graph illustration for the vertex with the highest sum of weights of the edges

To know which one we were going to use, we implemented both ideas and performed tests on a number of random graphs to see which was the most consistent. For that, we created 10 instances for different number of vertices (from 100 to 1000) and for different connectivity (25 50 75). We applied each of the two algorithms on each instance and averaged the 10 tests to get a usable result. We took the same criterion 2 to homogenize the results (take the vertex which has an adjacent edge with the highest weight compared to the last vertex added in the solution). The following results were obtained. We will name the best sum of adjacent weight *sum*, and the highest degree vertex *degree*.

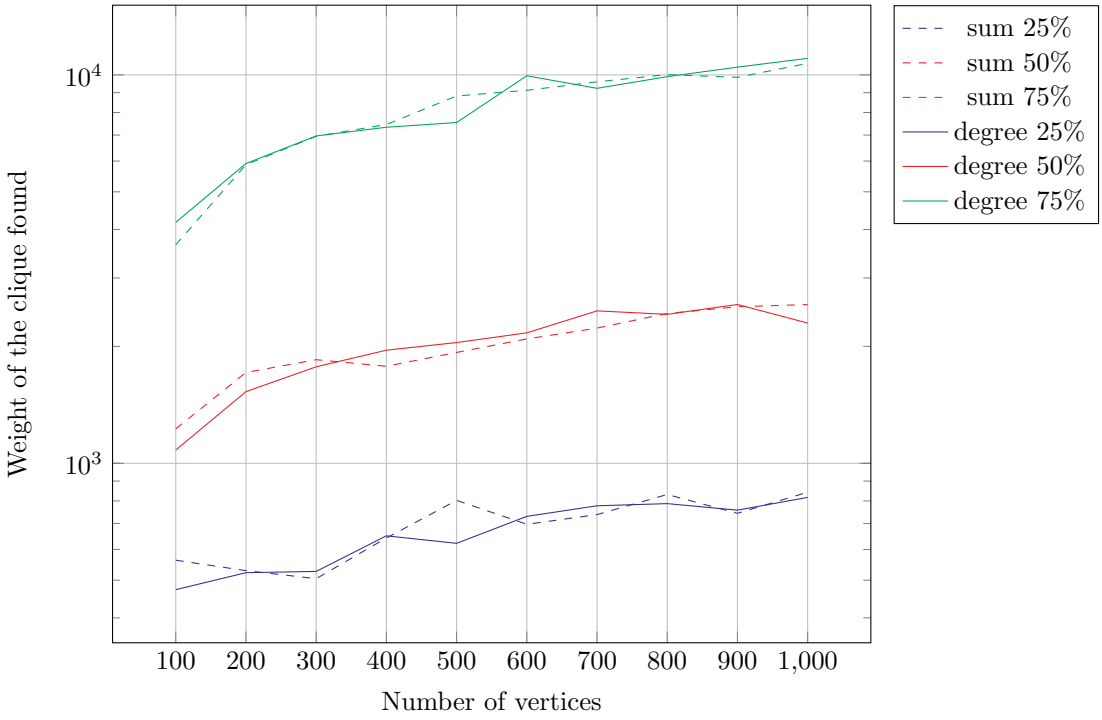


Figure 14: Weight of the clique found for different percentages of connectivity by two criteria algorithm(based on the highest degree/based on the highest sum of the weigh of its adjacent edge)

We can observe in figure 14 that the two results obtained are identical. This can be explained by the fact that on an average of several graphs, the values become more and more regular and homogeneous which does not favor either of the two criteria (while each can be better than the other on some cases). We therefore decided to choose the highest degree vertex because it have less algorithm complexity than its opponent (although both are of the same absolute complexity $\mathcal{O}(n^2)$).

To choose criterion 2, we also considered two implementations. Seek each time for the last vertex added to S its neighbors whose edge is the highest. Or search as done in criterion 1, the highest degree neighbors of the last vertex added to S.

To know which one we were going to use, we implemented both ideas and performed the same process as for choosing criterion 1. The following results were obtained. We will name the neighbors whose edge is the highest $weightN$, and the highest degree neighbors $degreeN$

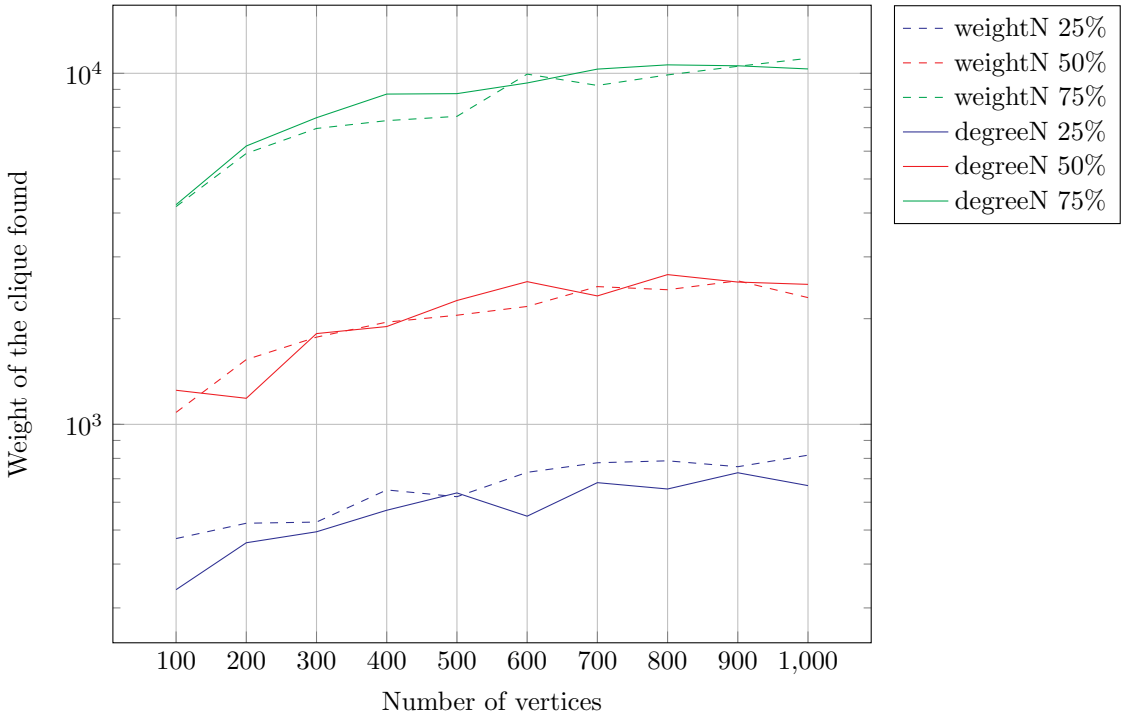


Figure 15: Weight of the clique found for different percentages of connectivity by two criteria of the constructive algorithm (neighbors that share the highest weighted edge/neighbors of the highest degree)

We can observe in figure 15 that the two results obtained are also identical or present a difference that is not important enough to be noticeable. This can be explained with the same explanation as for the choice of criterion 1. We then chose to take the highest degree neighbors as criterion 2 because its implementation just requires us to make a sorted list of the different vertices of the graph. This makes us gain in constant complexity compared to its opponent.

Constructive heuristic can be contrasted with other types of heuristics, such as local search heuristics, which try to find a solution by making small changes to an existing solution, or random heuristics, which generate solutions randomly and then choose the best one. Constructive heuristics are often used when it is important to find a solution that is complete and comprehensive, rather than just a local improvement.

Examples of some famous problems that are solved using constructive heuristics are the flow shop scheduling, the vehicle routing problem and the open shop problem.

3.2 How it works

To explain how our algorithm works, we will keep track of two groups of vertices : S which is a partially constructed clique and the partial solution that we will gradually implement. Moreover, we also have Z which is the list of all the vertices sorted according to a criterion (the best degree). Furthermore, we got P which is the candidates vertices that could be included in the clique, and which represents the union of all vertex neighbors of the vertices in S.

The algorithm begins by forming a list of all the vertices sorted according to a criterion (the best degree). This is to facilitate the identification of the next vertices that we will add to our solution S. This makes it easier to identify the next vertices to be added to our solution S, it also saves complexity because we are not bound to check each criterion at each iteration. Then the algorithm will initialize P, and insert in it all the vertices of the graph.

It will then retrieve the first element of the list Z that also belongs to P, and add it to the solution S. The selected element will then be removed from Z. After this step, P is updated by considering only the neighbors of the vertices that are already part of S. This process is then repeated recursively until no more vertices are left in P, at which point the algorithm has obtained its maximum clique S.

To illustrate the Constructive algorithm, let's use the example in Figure 1 on page 3 while adding some weight to its edges:

Step 0:

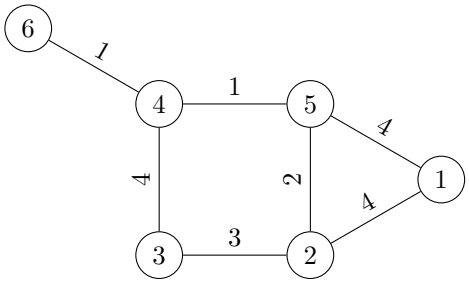


Figure 16: Graph illustration for the constructive algorithm at step 0

At the initial step, as said before, we will initialize S, P and Z by sorting the vertices based on a criterion. Here we chose the vertex of the highest degree.

S = {∅}	P = {1,2,3,4,5,6}	Z = {2,4,5,1,3,6}

Step 1:

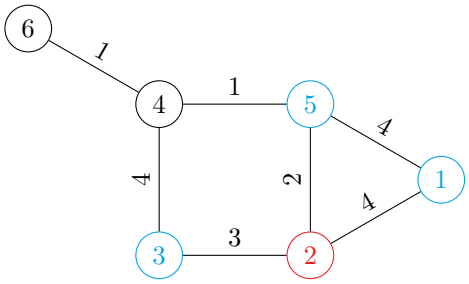


Figure 17: Graph illustration for the constructive algorithm at step 1

In step 1, the algorithm will take the first vertex of Z if it is common to P (here 2). In case of a tie, the algorithm will take the first checked vertex. He will then add it to S, represented in red. P, represented in blue, will be updated by taking only the vertices that are neighbors to all the members of S (here, S is composed only of vertex 2, so we take only the neighbors of 2).

S = {2}	P = {3,5,1}	Z = {4,5,1,3,6}

Step 2:

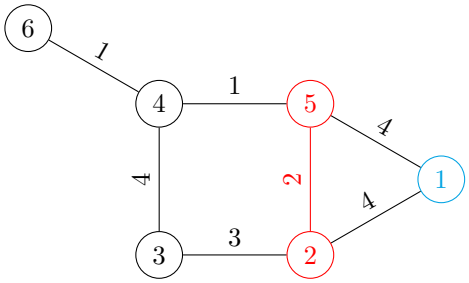


Figure 18: Graph illustration for the constructive algorithm at step 2

In step 2, the algorithm will repeat the process of step 1 by making a recursive call until P is empty. We can note in the process that our algorithm iterates again on some elements of Z, deleting them and refactoring the list does not improve the complexity, but implementing it as we did improve it in general(by not checking every neighbors each times).

S = {2,5}	P = {1}	Z = {4,1,3,6}

Step 3:

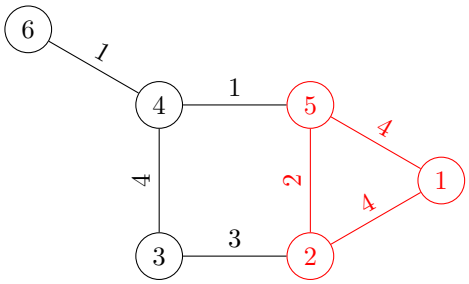


Figure 19: Graph illustration for the constructive algorithm at step 3

In step 3, the algorithm repeat the step 1 by making a recursive call. It will finally find 1 which is the last vertex of P. The algorithm stop, and we get the maximum clique in red $(1, 2, 5)$.

S = {2,5,1}	P = {}	Z = {4,3,6}

Now we calculate the weight of this clique. The constructive algorithm is now finished, and we have obtained the following maximum clique of weight 10 :

$$(1, 2, 5)$$

3.3 Pseudocode

Algorithm 4 constructiveMEWC function

```
1: procedure CONSTRUCTIVEMEWC(G)
2:     S ← ∅                                  ▷ Variable to store the clique that represents the Solution
3:     P ← G.vertices()                       ▷ Lists of vertex to store the vertex that could be added to S
4:     Z ← SORTVERTICESDEGREE(G, P)           ▷ variable to store the vertices sorted by degree
5:     CONSTRUCTIVERECURSIVEMEWC(G, S, P, Z)
6:     GETCLIQUEWEIGHT(S)
7:     return S
```

Algorithm 5 constructiveMEWCRecursive function

```
1: procedure CONSTRUCTIVEMEWCRECURSIVE(G, S, P, Z)
2:     if P = ∅ then
3:         return
4:     v ← GETBESTVERTEX(P, Z)                ▷ Get the vertex with the highest degree
5:     S ← S ∪ {v}
6:     CONSTRUCTIVERECURSIVEMEWC(G, S, P ∩ (N(v) ∪ v), Z)
```

Algorithm 6 getBestVertex function

```
1: procedure GETBESTVERTEX(P, Z)
2:     bestVertex ← ∅
3:     for v ∈ Z do
4:         if ∃v ∈ P then
5:             bestVertex ← v
6:             Z ← Z − {v}
7:             break
8:     return bestVertex
```

Algorithm 7 sortVerticesDegree function

```
1: procedure SORTVERTICESDEGREE(G, P)
2:     degrees ← (∅, ∅)                       ▷ Make a pair (vertex, degree)
3:     Z ← ∅
4:     adjMatrix ← G.getAdjacencyMatrix()
5:     for all vertex ∈ P do
6:         neighbors ← adjMatrix[vertex.id()]
7:         degrees.push_back(vertex, |N|)
8:     SORT(degrees)                          ▷ Sort the vertices by degree
9:     for all [vertex, degree] ∈ degrees do
10:        Z.push_back(vertex)
11:    return Z
```

Algorithm 8 Clique weight function

```
1: procedure getCliqueWeight(S)                ▷ We assume that the clique is complete
2:     weight ← 0
3:     n ← |S|
4:     for all i ∈ {1, 2, ..., n} do           ▷ For each vertex in the clique
5:         for all j ∈ {i + 1, i + 2, ..., n} do   ▷ For each other vertex in the clique
6:             weight ← weight + w_ij          ▷ Add the weight of the edge between the two vertices
7:     return weight
```

3.4 Complexity

Let be a graph $G = (V, E)$, such that $n = |V|$ and $m = |E|$, we can now calculate the complexity of our algorithm. The cost of attributing a value to a variable should always be $\mathcal{O}(1)$. The cost of getting an adjacency matrix of the graph should always be $\mathcal{O}(1)$, this is due to the efficient implementation of our classes to get it.

The worst complexity of our algorithm is when we study a complete graph. This means that all vertices have the maximum number of neighbors possible.

First, we will call sortVerticesDegree to sort all the vertices in ascending order. To do this, we will initialize a pair vector (vertex, degree) that we will fill with its adjacent matrix. This operation takes $\mathcal{O}(n)$ times. We will then use the sort function to sort the vertices according to their degree, which takes $\mathcal{O}(nlogn)$ times[2]. Then we will fill Z with the sorted vertices of the pair vector. The operation takes $\mathcal{O}(n)$ times.

The complexity of sortVerticesDegree will be :

$$T(n) = n + nlogn + n \in \mathcal{O}(nlogn) \tag{6}$$

In the function constructiveMEWCRecursive, we call the getBestVertex function which will iterate on all the elements of Z ($\mathcal{O}(n)$ because the graph is complete). To check if the elements of Z is member of P, we use the find() function which is constant[3]. The function is $\mathcal{O}(1)$ if there are no hash collisions, can be $\mathcal{O}(n)$ if there are hash collisions or hash is the same for any key. Something that does not happen in our case, it will then always be $\mathcal{O}(1)$. We will then refactor the elements of Z at the last iteration on the elements Z and break. The operation take $\mathcal{O}(n)$ times but 1 times.

The complexity of getBestVertex will be :

$$T(n) = (n + n) \in \mathcal{O}(n) \tag{7}$$

Moreover, in the function constructiveMEWCRecursive, we will iterate on all the elements of the adjacency matrix of the vertex we're looking to check its neighbors in order to make the intersection of P and the neighbors. To do this, we use the count() function which is constant[4]. The function is $\mathcal{O}(1)$ in the same logic of the find() function.

[2] https://en.cppreference.com/w/cpp/algorithm/sort
[3] https://en.cppreference.com/w/cpp/container/unordered_set/find
[4] https://en.cppreference.com/w/cpp/container/unordered_set/count

We then can calculate the complexity of the function constructiveMEWCRecursive.

$$T(n) = \begin{cases} 3 & \text{if } n = 0 \\ T(n-1) + 3n & \text{if } n > 0 \end{cases} \tag{8}$$

By subtitution, we have :

$$\begin{aligned} T(n) &= T(n-1) + 3n && (9)\\ &= [T(n-2) + 3n] + 3n && (10)\\ &= ... && (11)\\ &= T(n-i) + i * (3n) && (12)\\ &= [T(n-i+1) + 3n] + i * (3n) && (13)\\ &= ... && (14)\\ &= T(n-n+1) + (3n) * (n-1) && (15)\\ &= 3 + 3n^2 - 3n \in \mathcal{O}(n^2) && (16) \end{aligned}$$

The complexity of the weight calculation is $\mathcal{O}(n^2)$, because we need to iterate two times over the vertices of the clique to get every edge and by extension their cumulative weight.

So the total complexity of constructiveMEWC is :

$$T(n) = nlogn + n^2 + n^2 \in \mathcal{O}(n^2)$$

3.5 Bad Instance

As we have seen when choosing our criteria, although our algorithm works very well in general, some instances may be more advantageous on other criteria. So there are instances where our algorithm will not be the best.

Our algorithm takes the highest degree vertex, so a bad instance would be when the highest degree vertex is not part of the MEWC.

Also, since we generated the graphs randomly, we usually get a consistent result. But if we are in a case with very heterogeneous values it can happen to have results very far from the MEWC. Here in figure 20, the MEWC includes a vertex that is not reachable with this choice.

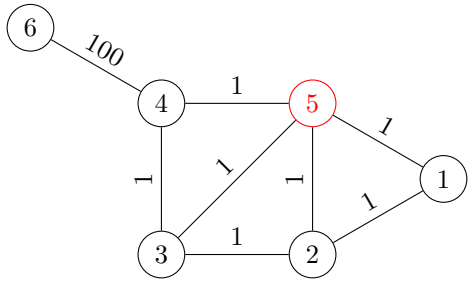


Figure 20: Graph illustration for bad instance of our algorithm when we choose the highest degree vertex as criteria 1

Moreover, our algorithm takes the highest degree vertex to add a new vertex to the solution S. This raises the same problems as those presented by the criterion. The algorithm can form a clique with a vertex that does not belong to the MEWC or is worse while the initial clique could be part of the MEWC or a better solution. Here in figure 21, the algorithm will take $5 \to 2 \to 3$ when every other choice could lead to a better clique.

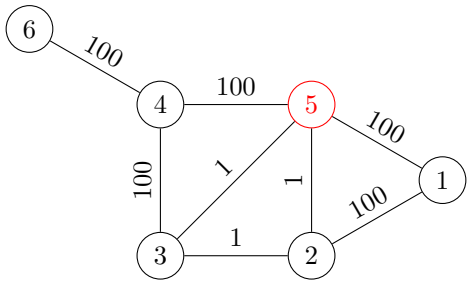


Figure 21: Graph illustration for bad instance of our algorithm when we choose the highest degree vertex as criteria 2

3.6 Experiments

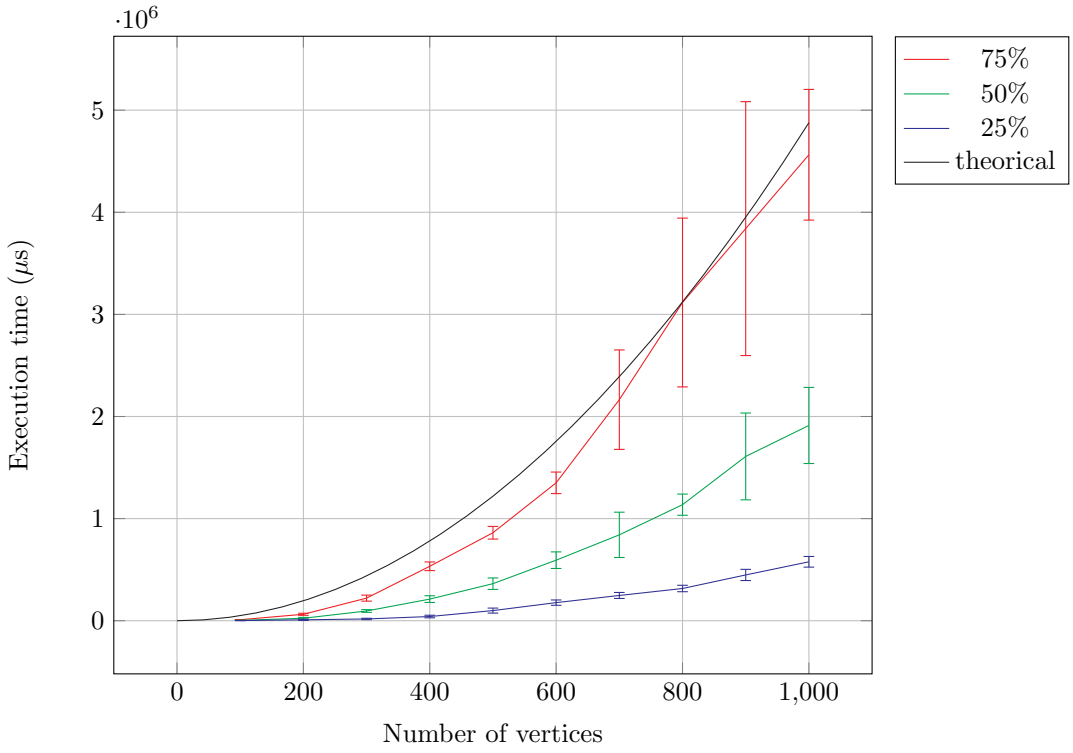


Figure 22: Execution time of the constructive algorithm for different percentages of connectivity.

For this experiment, we generated twenty random graphs with 100 to 1000 vertices to find an average execution time for each percentage of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 40. At each number of vertices, an average execution time is calculated, and the standard deviation is represented by the error bars. The number of graphs generated for each percentage of connectivity and number of vertices is limited to 20 because it is a sufficient number of tests to get what we want to analyze. Moreover, we use a linear scale that allows us to highlight the results we want to analyze.

3.7 Analysis

In figure 40, we can observe the polynomial increase in the execution time of the constructive agorithm with the number of vertices for different percentages of connectivity. Note that the theorical is matching the 75% connectivity with a constant factor of 4.9.

As we know, the worst case of the algorithm is when the graph is complete. That is, connectivity is at its maximum (100%). As we can see, increasing the connectivity and the number of vertices of the graph has a chance to increase the complexity of the structure of that graph, which in turn increases the time complexity of the algorithm.

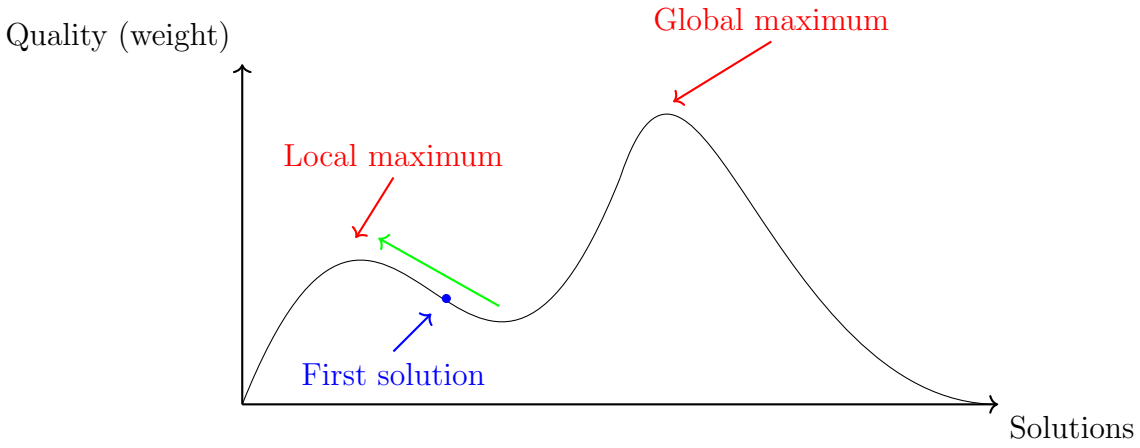
We can notice some factors appearing according to the different number of connectivity. The algorithm takes in average 9 times more time with 75% than with 25% and in average 3 times more time with 50% than with 25%.

So, the constructive algorithm is good if we want to find an approximative solution for a large number of vertices because on them, we obtain a rather short execution time. Howewer, we will see that it has some boundaries in terms of results obtained. We will detail these points further in the conclusion.

4 Local Search Algorithm

4.1 Presentation

The local search algorithm is also a heuristic algorithm, like the constructive one. So, like the constructive algorithm, the local search will find a solution that may or may not be the best one, and we will have no idea how far we are from the real solution. The difference between constructive and local search is that constructive is based on criteria to find a solution whereas local search will find a first solution and visit the "neighboring" solutions to find an optimal solution. This "neighborhood" criterion is the most important one for this algorithm, it is the one that will choose to enter one way or another, and therefore it is the one that will give the final solution and define its quality. The neighbor of a solution is found by making small changes to the solution found to see if a better solution can be obtained. If this is the case, then we start again by looking at the neighbors of this new solution up to a certain stopping condition. As can be seen from this definition, the local search algorithm will only take the neighbors that have a better solution, so if to reach the optimal solution we have to go through a decrease in the quality of the solution, then we will never reach this optimal solution. Finally, if we consider a graph with the neighboring solutions on the x-axis and the quality of these solutions on the y-axis (in this case the weight of the clique found), we will observe that the local search algorithm will only give the local maximum, and not the global maximum.



The first thing to do was to find an initial solution that is most likely to be close to the best solution. To do this, the algorithm relies on an arbitrary criterion that will work in most cases: take the highest degree vertex in the graph, its highest degree neighbor and try to form a maximal clique with these two vertices. This criterion will allow in most cases to form a rather large clique because if we take the highest degree vertex, we have more chance to form a larger clique, and thus we have more chance to have a clique of great weight.

The second and harder thing to do was to find the neighborhood criterion: what is the neighbor of a maximal clique? After some research, the solution we found was to take our

current maximum clique, remove a vertex from it (the one that adds the least weight to the clique) and see if we can find a new, better maximum clique. The problem with this solution for finding neighbors was that we only took into account very few cases: we did not take into account the case where we had to delete two or more vertices in our clique to find a better one. The solution found to solve this problem was to try to remove all sets of k vertices and see if we could get a better clique. The new problem with this way of doing things was the complexity: to find all sets (or part of sets) of k vertices among n (the total number of vertices in the clique), the complexity became $\mathcal{O}(n!)$. Now, the principle of a heuristic algorithm like local search is to find an approximate solution but with polynomial complexity and $n!$ is anything but polynomial, and it would take like forever to run this algorithm for a large graph with many of vertices and edges.

After several trials running both the factorial complexity algorithm that takes sets of vertices to find a neighboring solution and the one that removes only one vertex at a time, we found that both found an approximately similar solution where the one that removed only one vertex ran almost instantaneously as opposed to the other that took several minutes. As the goal of the local search algorithm is to find an approximate solution in a very short time, we decided to keep the first neighborhood criterion found, that is to locate a vertex to try to improve the clique found.

4.2 How it works

As said before, the local search algorithm takes an initial solution, and as long as a certain condition is not met, it looks at the neighbors of this initial solution and keeps the neighboring solution if it is better. Here is a rough overview of how all local search algorithms work:

1. Finding an initial solution
2. Look at a neighbor solution of the current one
    - If the new solution is better than the old one, keep it as a solution
    - Else, keep the old one as solution
3. As long as a certain stop condition has not been reached, repeat step 2

First, in our case, finding the initial solution is done by taking the highest degree vertex, then its highest degree neighbor. These two vertices form a clique which will be improved to form a maximal clique as shown below:

Step 0:

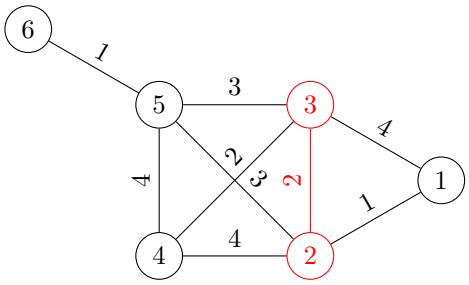


Figure 23: Graph illustration for the local search algorithm at step 0

The first vertex to be taken will be 2, of degree 4. Among its neighbors $\{1, 3, 4, 5\}$, the highest degree vertex is 4 ($d(3) = 4$) so we form a clique with the vertices $\{2, 3\}$.

Step 1:

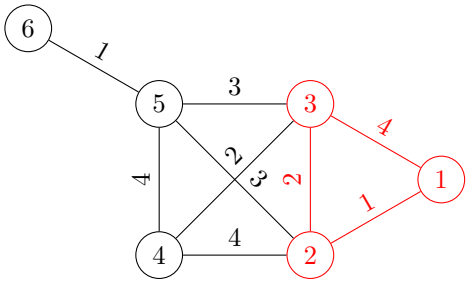


Figure 24: Graph illustration for the local search algorithm at step 1

We can assume that the first vertex looked at to find a maximal clique will be 1, and as it is the neighbor of both 2 and 3 (all the vertices of the current clique), we add it to the clique. We now look for common neighbors of the three vertices of the clique: we find none, so the clique $\{1, 2, 3\}$ forms a maximal clique: this is our initial solution and its weight is $w(C_{init}) = 1 + 2 + 4 = 7$.

We now have an initial solution. The objective will be to look at its neighbors to find a better solution (that is a maximal clique with a higher weight). This is done by deleting a vertex in our current solution. With one less vertex, the clique may not be maximal anymore, so we will try to make it maximal. If the weight of the new clique is greater than the old one, then we keep this new clique as the current solution. Otherwise, we keep the old one and try to improve it by removing another vertex. To choose which vertex to remove, we choose the one that adds the least weight to the clique. This will increase the chances of finding a better solution.

Step 2:

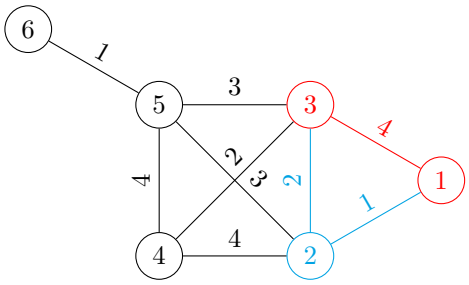


Figure 25: Graph illustration for the local search algorithm at step 2

Among the three vertices of the current solution, the 2 is the one which adds the least weight (if we remove it, we reduce the weight of the clique by $2 + 1 = 3$). We obtain then a clique made up of the vertices $\{1, 3\}$. Now, these two vertices have no neighbor in common, so this clique is maximal and of weight $w(C_1) = 4$. As $w(C_1) = 4 < w(C_{init}) = 7$, we do not keep this solution.

Step 3:

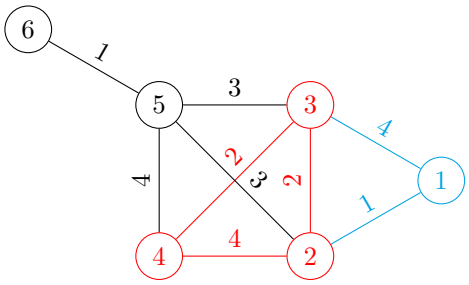


Figure 26: Graph illustration for the local search algorithm at step 3

The second vertex of minimum weight is 1. If we remove it, we observe that the vertex 4 is adjacent to the two remaining vertices in the clique ($\{2, 3\}$). So we add it to our solution, and we obtain the clique $C_2 = \{2, 3, 4\}$ with $w(C_2) = 9$. Let's see if this clique is maximal or if we can still improve it.

Step 4:

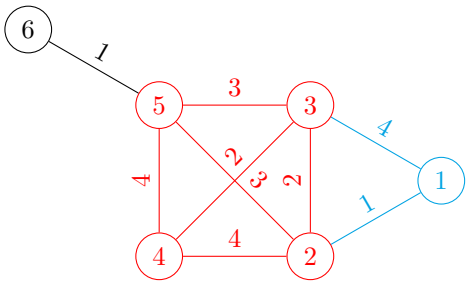


Figure 27: Graph illustration for the local search algorithm at step 4

We can see that vertex 5 is adjacent to all the vertices of C_2. We can therefore add it to the clique: $C_2 = \{2, 3, 4, 5\}$ and $w(C_2) = 18$. We notice that the vertices of the clique have no common neighbor, so we keep C_2 as is. Furthermore, we then have $w(C_2) = 18$ greater than $w(C_{init}) = 7$. We therefore take C_2 as the current solution.

Now that we know how the solution is improved, all that remains is to see when the algorithm ends: what is the stop condition? A simple way to stop the algorithm is to define the stop condition as the moment when we have no more vertices to test, that is when we have removed all vertices one by one from the current solution without finding a better solution. Thus, we

will have tested all the possibilities, and we will obtain a maximal clique which cannot be improved with this local search algorithm.

4.3 Pseudocode

Algorithm 9 Local search MEWC algorithm

```
1: procedure LocalSearchMEWC(G = (V, E))
2:     solution ← findInitialSolution(G)                ▷ Variable to store the actual solution
3:     solutionWeight ← getCliqueWeight(solution) ▷ Variable to store the weight of the actual solution
4:     testedVertices ← ∅                               ▷ Variable to store the vertices that have been tested
5:     while |testedVertices| is not |solution| do
6:         neighborSolution ← findNeighbor(G, solution, testedVertices) ▷ Get one of the neighbors of
           the actual solution (fill testedVertices with the vertex removed in this loop)
7:         neighborWeight ← getCliqueWeight(neighborSolution) ▷ Get the weight of the neighbor so-
           lution
8:         if neighborWeight > solutionWeight then
9:             solution ← neighborSolution
10:            solutionWeight ← neighborWeight
11:            testedVertices ← ∅
12:    return solution
```

Algorithm 10 Find initial solution algorithm

```
1: procedure findInitialSolution(G = (V, E))
2:     maxDegreeVertex ← 0                              ▷ Variable to store the vertex of max degree
3:     for all vertex ∈ V do
4:         if d(vertex) > d(maxDegreeVertex) then
5:             maxDegreeVertex ← vertex
6:     maxDegreeVertex2 ← 0 ▷ Variable to store the neighbor vertex of maxDegreeVertex of max degree
7:     for all vertex ∈ N(maxDegreeVertex) do
8:         if d(vertex) > d(maxDegreeVertex2) then
9:             maxDegreeVertex2 ← vertex
10:    initialSolution ← {maxDegreeVertex, maxDegreeVertex2} ▷ The clique formed by the two vertices
       of max degree
11:    improveClique(G, initialSolution)                ▷ Improve initialSolution to have a maximal clique
12:    return initialSolution
```

Algorithm 11 Find neighbor algorithm

```
1: procedure FINDNEIGHBOR(G = (V, E),C_i,testedVertices)
2:     minWeightVertex ← 0        ▷ Variable to store the vertex that adds a minimum weight in the clique
3:     w_min ← ∞                  ▷ Variable to store the min weight added by a vertex
4:     for all v_1 ∈ V & v_1 ∉ testedVertices do
5:         weight ← 0             ▷ Variable to store the weight added by v_1
6:         for all v_2 ∈ N(maxDegreeVertex) & v_2 ≠ v_1 do
7:             weight ← weight + w(v_1, v_2)
8:         if weight < w_min then
9:             minWeightVertex ← v_1
10:            w_min ← weight
11:    newClique ← C_i − {minWeightVertex}   ▷ Copy the initial clique but remove the tested vertex
12:    X ← {minWeightVertex}      ▷ Set of vertices that we don't want to be in the new solution
13:    improvement ← IMPROVECLIQUE(G, newClique, X, w_min) ▷ improvement will be greater than 0 if
       improveClique() find a clique that adds a weight greater than w_min
14:    if improvement <= w_min then
15:        testedVertices ← testedVertices + {minWeightVertex}
16:        return C_i             ▷ If no better solution have been found, return the initial clique
17:    return newClique
```

Algorithm 12 Improve clique algorithm

```
1: procedure IMPROVECLIQUE(G = (V, E),C,X,w_min,w_actual improv)
2:     C_c ← C                    ▷ Variable to store a copy of clique
3:     w_improv ← 0               ▷ Variable to store the weight improvement done by adding a vertex in this iteration
4:     total ← 0                  ▷ Variable to store the weight improvement made by adding vertices from this iteration
5:     for all v ∈ V with v ∉ C_c & v ∉ X do
6:         commonNeighbor ← True  ▷ Boolean to know if v is the common neighbor of all vertices of C
7:         for all v_c ∈ C_c do
8:             if (v, v_c) ∉ E then
9:                 commonNeighbor ← False
10:                break
11:        if commonNeighbor then
12:            for all v_c ∈ C_c do
13:                w_improv ← w_improv + w(v, v_c)
14:            C_c ← C_c ∪ {v}
15:            total ← w_improv + IMPROVECLIQUE(G, C_c, X, w_min, w_actual improv + w_improv) ▷ Try improv-
               ing the clique by adding another vertex
16:            if w_actual improv + w_improv ≤ w_min then
17:                return 0       ▷ That means that we didn't find a better solution than the old one
18:            else
19:                C ← C_c        ▷ Copy the improved clique int the original one
20:                return total   ▷ Return the total improvement
21:        else
22:            X ← X ∪ {v}
23:    return 0                   ▷ Return 0 if we have no more vertices that are common neighbors of C
```

4.4 Complexity

As for the other algorithms, let be a graph $G = (V, E)$ such that $n = |V|$. The cost of attributing a value to a variable should always be $\mathcal{O}(1)$.

The complexity of the clique improvement algorithm is $\mathcal{O}(n^2)$ each time it is called. This

algorithm is recursive only if it finds an improvement in the clique size. However, each time it is called, IMPROVECLIQUE() can only find at most n improvements, so it recursively calls itself at most n times. So the complexity of IMPROVECLIQUE() is $\mathcal{O}(n^3)$ when called by a function other than itself.

The complexity of the best neighbor search algorithm is $\mathcal{O}(n^3)$. It goes through $n*n$ vertices when computing the vertex that adds a minimum weight, and as shown below the complexity of IMPROVECLIQUE() is $\mathcal{O}(n^3)$. The complexity of FINDNEIGHBOR() is then $\mathcal{T}(n^2+n^3) = \mathcal{O}(n^3)$.

The complexity of the initial solution search algorithm is $\mathcal{O}(n^3)$. As for the neighbor search algorithm, it is IMPROVECLIQUE() that will define the complexity of this algorithm because it goes through all the n vertices 2 times (that is $\mathcal{O}(n)$ because get the degree of a vertex is $\mathcal{O}(1)$ as the class is made), and then calls the function IMPROVECLIQUE() of complexity $\mathcal{O}(n^3)$ to find a maximal clique. The complexity of FINDINITIALSOLUTION() is thus $\mathcal{T}(2n+n^3) = \mathcal{O}(n^3)$.

Finally, the local search MEWC algorithm starts by finding an initial solution, which is $\mathcal{O}(n^3)$. It gets the weight of this solution ($\mathcal{O}(n)$) and enters a *while* loop and does not exit until the size of $testedVertices$ is equal to the number of vertices of the current solution (that is until we have tested to remove each vertex one by one without finding a better solution). At each iteration in this *while* loop, the algorithm calls the function FINDNEIGHBOR() of complexity $\mathcal{O}(n^3)$ and GETCLIQUEWEIGHT() of complexity $\mathcal{O}(n)$. It only remains to know how many iterations are done in the *while* loop. In the worst case, the algorithm tries to remove all the vertices each time before finding one that improves the solution, and in all the algorithm can only find n better solutions (because each time we find a solution, this one has necessarily a size greater than or equal to the previous one, and the more we increase the size of the clique, the more the number of possible cliques of this size in the graph decreases with a number of n in the case where we have only cliques of 1 vertex). This gives $n*n$ iterations in the *while* loop. So we have for the local search MEWC algorithm a complexity of $\mathcal{T}(n^3+n+n^2*(n^3+n)) = \mathcal{O}(n^5)$.

In reality, the actual complexity of this algorithm is only a tiny fraction of $\mathcal{O}(n^5)$. Most of the time when calculating the complexity in our case, the n represents the size of the actual solution clique instead of the total number of vertices in the graph. Similarly, the number of iterations in the *while* loop is much less than n^2 and would be closer to n in most cases, except in very special cases.

4.5 Instance

The local search algorithm is a heuristic algorithm. As a heuristic algorithm, there will be a lot of bad instances for the local search algorithm because the objective is to be much faster than the exact algorithm and not to have the exact solution. The algorithm has to make choices at several points that will define the quality of the result obtained. The two main moments where it can make the wrong choice are: when choosing the initial solution and when choosing which best neighbor to go to.

For the first point, if the local search algorithm starts towards a solution too far from the optimal one, it will never be able to approach it. One can imagine in an extreme case a graph

divided into two parts connected by only two vertices, with the first part like a star and the second a complete subgraph as shown in the example below:

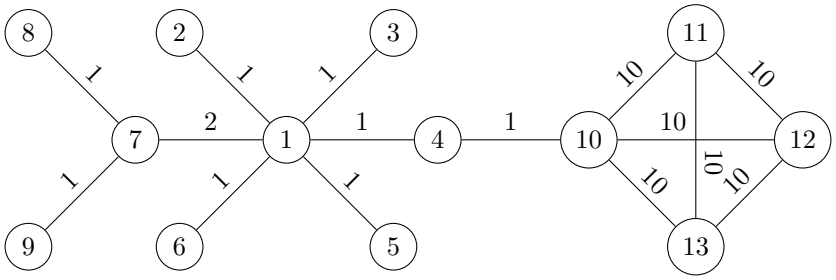


Figure 28: Graph illustration for a case of local search algorithm bad instance

Here, the first vertex that will be taken to form the initial solution will be 1 because $d(1) = 6$ is the maximum degree and the second vertex will be 7 because it is the highest degree neighbor of 1. This initial solution already forms a maximum clique of weight 2. When we look at the neighbors of this solution by removing either 1 or 2, we will never obtain a better solution than the initial solution. However, we observe well on this example that the maximum clique is the clique formed by the vertices $\{10, 11, 12, 13\}$ of weight 60, well higher than 2. The best way to counter this kind of problem would be to use a **meta-heuristic algorithm** like grasp which will allow itself to take solutions of lesser quality to try to reach the best solution.

The second way in which the algorithm can make a bad choice at a given time is by incorrectly choosing which of the best neighbors of the current solution to keep. Let's take an example:

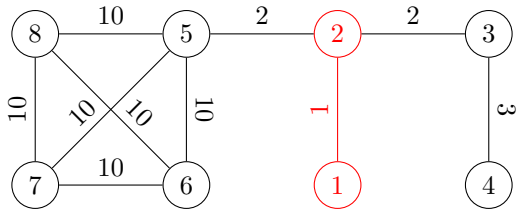


Figure 29: Graph illustration for a second case of local search algorithm bad instance

Here, let us suppose that the current solution is $C = \{1, 2\}$. When the algorithm goes to look for a neighbor of this solution, it will have two possible choices, both offering the same quality of solution: $\{2, 3\}$ and $\{2, 5\}$. The problem arises if he chooses to go towards $\{2, 3\}$. In this case, the best solution that he can find will be $\{3, 4\}$ with a weight of 3 whereas if he had gone towards $\{2, 5\}$, he would have obtained the maximum clique $\{5, 6, 7, 8\}$ with a weight 60. Even if we say in the algorithm that he must go to the solution which has the vertices of higher degree, the problem remains the same if we add a vertex between the 2 and the 5.

But even if there are several possibilities for the algorithm not to act as we would like it to, as we said before the goal of a heuristic algorithm is not to find the best solution, but to find an approximate solution in a very short time compared to the exact algorithm.

4.6 Experiments

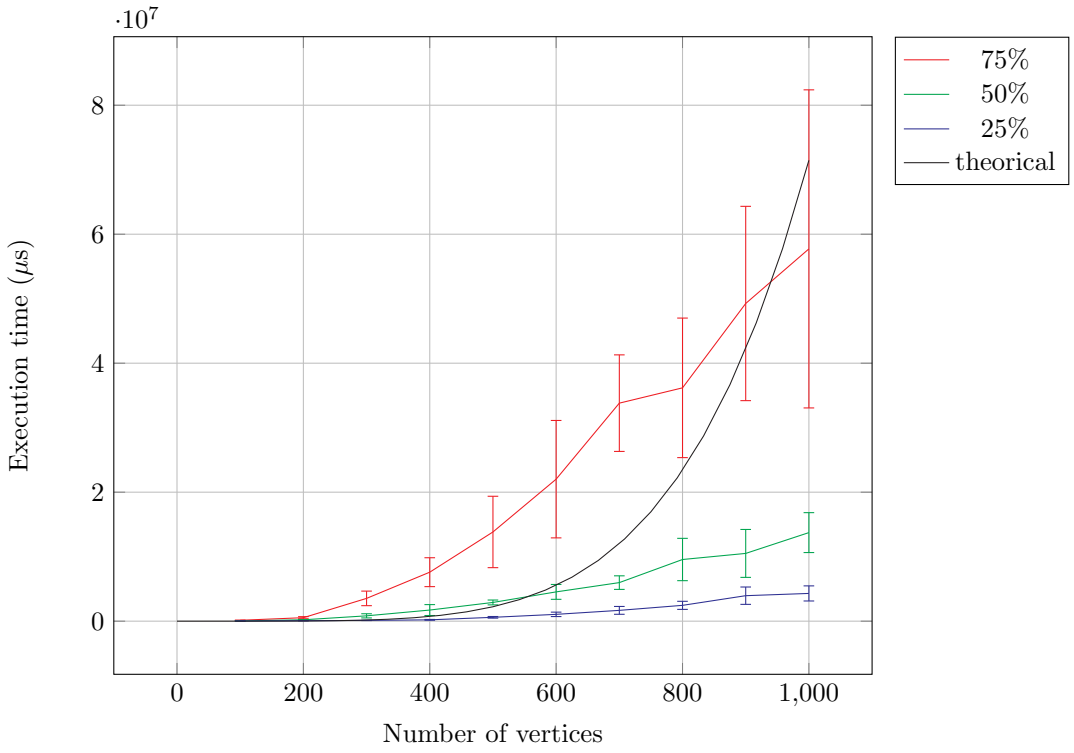


Figure 30: Execution time of the local search algorithm for different percentages of connectivity.

For this experiment, we generated five random graphs with 100 to 1000 vertices to find an average execution time for each percentage of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 30. At each number of vertices, an average execution time is calculated, and the standard deviation is represented by the error bars. The number of graphs generated for each percentage of connectivity and number of vertices is limited to 5 to reduce the execution time of the experiment which already takes more than 8 minutes to compute only one result for 1000 vertices case for the 75% connectivity. Moreover, we use a linear scale that allows us to highlight the results we want to analyze.

4.7 Analysis

In figure 30, we can observe the polynomial increase in the execution time of the constructive algorithm with the number of vertices for different percentages of connectivity. Note that the theoretical is matching the 75% connectivity with a constant factor of 7.146×10^{-8}.

As we know, the worst case of the algorithm is when the graph is complete. That is, connectivity is at its maximum (100%). As we can see, increasing the connectivity and the number of vertices of the graph has a chance to increase the complexity of the structure of that graph, which in turn increases the time complexity of the algorithm.

We can notice some factors appearing according to the different number of connectivity. The algorithm takes in average 18 times more time with 75% than with 25% and in average 4 times more time with 50% than with 25%.

So, the local algorithm is an in-between between the constructive and the exact algorithm in terms of execution time and results obtained. We will detail this points further in the conclusion.

5 Grasp Algorithm

5.1 Presentation

The **Greedy Randomized Adaptive Search Procedures** (GRASP) algorithm is a **meta-heuristic** search technique that has been developed to address difficult combinatorial optimization problems. **GRASP** incorporates restart procedures, controlled randomization, efficient data structures, and preprocessing to obtain good solutions to challenging problems. The algorithm works by incrementally constructing solutions using greedy randomization.

At each step, the algorithm chooses the next vertex of the clique using a greedy randomized heuristic. The heuristic is used to decide which vertex should be added to the current clique in order to maximize the total weight of the clique. The algorithm also incorporates a local search procedure to refine the solutions produced. This local search procedure is used to find cliques with higher weights than those generated by the greedy randomized heuristic. GRASP has been tested on a variety of graph types and has been shown to produce good solutions in a relatively short amount of time. Additionally, the algorithm can be easily modified to work for other graph optimization problems.

This algorithm uses the previously described Local Search algorithm to refine the solutions produced by the greedy randomized heuristic.

5.2 How it works

GRASP mainly uses two parts: the first one is the **greedy randomized heuristic** and the second one is the **local search**. The first part is used to construct the solution and the second one is used to refine it. The algorithm works by incrementally constructing solutions using greedy randomization. At each step, the algorithm chooses the next vertex of the clique using a greedy randomized heuristic. The heuristic is used to decide which vertex should be added to the current clique in order to maximize the total weight of the clique.

Let C denote the clique to be constructed. GRASP begins with an empty clique C. Let $k = 0$, $V_{k=0} = V$ and $E_{k=0} = E$. A plausible greedy choice for the maximum edge weight clique is to select the vertex with the highest sum of weights of its adjacent edges with respect to the graph induced by the yet unselected vertices that are not adjacent to any previously selected vertex. Let $N(v)$ denote the neighboring vertices of v and $w(u, v)$ denote the weight of the edge between vertices u and v. The greedy choice is to select a vertex with the highest sum of weight of its adjacent edges. Instead of selecting the greedy choice, the GRASP construction phase builds a restricted candidate list (RCL) of all vertices having high sum of adjacent edges weight, but not necessarily the highest. Let Γ be the highest sum of adjacent edges weight in V_k, i.e.

$$\Gamma = \max \left\{ \sum_{u \in N(v)} w(u, v) | v \in V_k \right\}$$

And let $\alpha > 0$ be the restricted candidate parameter. The value restricted candidate list is:

$$\text{RLC} = \left\{ v \in V_k | \sum_{u \in N(v)} w_{uv} > \frac{\Gamma}{1+\alpha} \right\}$$

We next describe a k-exchange search procedure in the initial graph G. The idea here is to take as input a clique $C \subseteq V$ of size p and consider all k-tuples of vertices in C, for a given parameter k, $0 < k < p$. For each k-tuple, we consider all possible k-tuples $\{v_{i_1}, ..., v_{i_k}\}$, apply an exhaustive search to find a maximum edge weight clique in the graph induced by the vertices of G such that $V_k = V \backslash \{v_{i_1}, ..., v_{i_k}\}$. If the resulting clique $\mathcal{N}$ is larger that C, the new best solution has to be $\mathcal{N}$.

For practical reasons, we use $k = 1$ in our implementation.

5.3 Pseudocode

Algorithm 13 Grasp algorithm

```
1: procedure GRASP(G = (V, E))
2:     BestSolution ← ∅                ▷ Variable to store the Best clique that represents the Solution
3:     for i = 1, ··· , RETRIES do     ▷ Iterate for RETRIES times
4:         Solution ← CONSTRUCTGREEDYRANDOMIZEDSOLUTION(G)
5:         Solution ← LOCALSEARCHGRASP(G, Solution)   ▷ call a function that search the best clique
6:         UPDATESOLUTION(Solution, BestSolution) ▷ Update the best solution with the solution we found
           if it is better
7:     return BestSol
```

Algorithm 14 ConstructGreedyRandomizedSolution function

```
1: procedure CONSTRUCTGREEDYRANDOMIZEDSOLUTION(G = (V, E))
2:     Solution ← ∅                    ▷ Initialize the solution
3:     P ← V                           ▷ Initialize the set of vertices to be selected
4:     while P ≠ ∅ do
5:         RCL ←MAKERCL(G,P)           ▷ Create the Restricted Candidate List (RCL) of vertices
6:         s ←SELECTELEMENTATRANDOM(RCL)   ▷ Choose randomly a vertex in the RCL
7:         Solution ← Solution ∪ {s}   ▷ Add the vertex chosen randomly to the solution
8:         ADAPTGREEDYFUNCTION(G,s,P)   ▷ Adapt the graph by removing s to P
9:     return Solution
```

Algorithm 15 MakeRCL function

```
1: procedure MAKERCL(G = (V, E),P)
2:     Γ ←GETGAMMA(G,P)                ▷ Get the highest sum weight of the graph
3:     for all v ∈ P do
4:         weight ←GETSUMADJACENTEDGES(G,v)
5:         if weight > Γ/(1+α) then    ▷ Condition for the RLC
6:             RCL ← RCL ∪ {v}         ▷ Add the vertex to the RCL
7:     return RCL
```

Algorithm 16 getGamma function

```
1: procedure getGamma(G = (V, E),P)
2:     Γ ← 0                                                    ▷ Initialize Γ
3:     for all v ∈ P do
4:         weight ←getSumAdjacentEdges(G,v)
5:         if weight > Γ then
6:             Γ ← weight
7:     return Γ
```

Algorithm 17 getSumAdjacentEdges function

```
1: procedure getSumAdjacentEdges(G = (V, E),v)
2:     sum ← 0                                          ▷ Initialize the sum to 0
3:     for all n ∈ N(v) do
4:         sum ← sum + w(v, n)      ▷ Add to the sum the weight of each adjacent edges of the vertex
5:     return sum
```

Algorithm 18 AdaptGreedyFunction function

```
1: procedure AdaptGreedyFunction(G = (V, E),v,P)
2:     P ← P\{v}                    ▷ Remove the vertex from the set of vertices to be selected
3:     for all u ∈ P do
4:         if u ∉ N(v) then                  ▷ If u not adjacent to v then remove it from P
5:             P ← P\{u}
```

Algorithm 19 LocalSearchGrasp function

```
1: procedure LocalSearchGrasp(G = (V, E),S)
2:     w_S ← getCliqueWeight(S)
3:     for all v ∈ S do
4:         G_k ← (V, E)                                  ▷ Create a copy of the graph
5:         G_k ← G_k\{v}                    ▷ Remove the vertex from the copy of the graph
6:         S_k ← LocalSearch(G_k)  ▷ use the local-search algorithm to find a solution within the subgraph
7:         w_{S_k} ← getCliqueWeight(S_k)          ▷ Get the weight of the solution of the subgraph
8:         if w_{S_k} > w_S then
9:             S ← S_k   ▷ If the solution of the subgraph is better than the solution we found then replace it
10:            w_S ← w_{S_k}
11:    return S
```

Algorithm 20 UpdateSolution function

```
1: procedure UpdateSolution(Solution,BestSolution)
2:     if getCliqueWeight(Solution) > getCliqueWeight(BestSolution) then
3:         Solution = BestSolution
```

5.4 Complexity

To calculate the complexity of the algorithm, we will consider the complexity of each part of the algorithm. The first part is the **greedy randomized heuristic** and the second one is the **local search**.

First part: **Greedy randomized heuristic**

getSumAdjacentEdges is a function that returns the sum of the weights of the edges adjacent to a vertex. This function iterates over the edges adjacent to the vertex and sums their weights. The complexity of this function is $\mathcal{O}(m)$.

getGamma is a function that returns the highest sum of adjacent edges weight in V_k. This function iterates over the vertices of P and calls the getSumAdjacentEdges function. The complexity of this function is $\mathcal{O}(n \cdot m)$.

MakeRCL is a function that returns the restricted candidate list. This function calls the getGamma function and iterates over the vertices of P. The complexity of this function is $\mathcal{O}(n \cdot m)$.

AdaptGreedyFunction is a function that adapts the greedy function to the restricted candidate list. This function iterates over the vertices of P and checks if they are not adjacent to a vertex. The complexity of this function is $\mathcal{O}(n)$.

ConstructGreedyRandomizedSolution is a function that constructs a solution using the greedy randomized heuristic. This iterates over the vertices of P and calls the MakeRCL function and the AdaptGreedyFunction function. It also selects a vertex from the RCL and adds it to the clique but as this part is constant in complexity, we can ignore it. The complexity of this function is $\mathcal{O}(n^2 \cdot m)$.

Second part: **Local search**

We know from previously that the complexity of the getCliqueWeight function is $\mathcal{O}(n^2)$ and the complexity of the LocalSearch function is $\mathcal{O}(n^5)$. We can then say that the complexity of the LocalSearchGrasp function is $\mathcal{O}(n^6)$ as it iterates over the vertices of the solution.

Combining the two parts, we can say that the complexity of the Grasp function is $\mathcal{O}(n^6)$.

5.5 Bad Instance

The **Greedy Randomized Adaptive Search Procedures** (GRASP) algorithm is a **meta-heuristic** search technique that has been developed to address difficult combinatorial optimization problems. As a heuristic algorithm, GRASP can have a lot of bad instances where it will not be able to find a good solution. Those instances are the same as the ones for the Local Search algorithm.

One way the algorithm can better handle bad instances is by using its randomized nature. The algorithm can be run multiple times and the best solution found can be returned. This will increase the probability of finding a good solution.

5.6 Experiments

α	iterations	execution time			weight		
		min	avg	max	min	avg	max
0.6	6	0.047	0.053	0.065	568	623.8	693
	7	0.044	0.055	0.081	568	623.8	693
	8	0.045	0.051	0.057	568	623.8	693
0.7	6	0.048	0.057	0.074	568	619.2	693
	7	0.046	0.058	0.077	568	623.8	693
	8	0.048	0.056	0.064	568	623.8	693
0.8	6	0.043	0.055	0.061	568	623.8	693
	7	0.044	0.057	0.080	568	623.8	693
	8	0.047	0.050	0.057	568	623.8	693
0.9	6	0.045	0.054	0.061	568	623.8	693
	7	0.049	0.057	0.067	568	623.8	693
	8	0.049	0.055	0.062	568	619.2	693

Figure 31: Experimental results: GRASP solution statistics (20 vertices, 5 runs)

α	iterations	execution time			weight		
		min	avg	max	min	avg	max
0.6	6	0.299	0.344	0.435	874	1 166.4	1 442
	7	0.283	0.337	0.405	871	1 165.8	1 442
	8	0.246	0.335	0.453	871	1 120.6	1 254
0.7	6	0.304	0.344	0.406	838	1 114	1 254
	7	0.271	0.327	0.404	838	1 114	1 254
	8	0.294	0.350	0.453	871	1 165.8	1 442
0.8	6	0.256	0.327	0.418	838	1 114	1 254
	7	0.286	0.336	0.429	874	1 166.4	1 442
	8	0.259	0.350	0.443	868	1 120	1 254
0.9	6	0.284	0.318	0.358	874	1 089.4	1 254
	7	0.279	0.329	0.421	1 000	1 133.2	1 254
	8	0.269	0.326	0.412	871	1 165.8	1 442

Figure 32: Experimental results: GRASP solution statistics (40 vertices, 5 runs)

α	iterations	execution time			weight		
		min	avg	max	min	avg	max
0.6	6	0.867	1.017	1.111	1 126	1 328	1 693
	7	0.852	1.034	1.153	1 141	1 382.8	1 693
	8	0.837	1.015	1.120	1 126	1 328	1 693
0.7	6	0.752	1.053	1.235	1 126	1 335.2	1 693
	7	0.788	1.021	1.228	1 141	1 359.2	1 693
	8	0.935	1.050	1.171	1 141	1 359.2	1 693
0.8	6	0.805	0.992	1.124	1 141	1 359.2	1 693
	7	0.869	1.093	1.266	1 141	1 340.8	1 693
	8	0.839	1.075	1.336	1 141	1 359.2	1 693
0.9	6	0.720	0.995	1.128	1 126	1 328	1 693
	7	0.803	1.046	1.260	1 141	1 348.2	1 693
	8	0.845	1.059	1.196	1 141	1 379.8	1 693

Figure 33: Experimental results: GRASP solution statistics (60 vertices, 5 runs)

α	iterations	execution time			weight		
		min	avg	max	min	avg	max
0.6	6	1.520	1.852	2.639	1 292	1 539.4	1 766
	7	1.512	1.834	2.699	1 418	1 600.6	1 969
	8	1.550	1.751	2.429	1 292	1 580	1 969
0.7	6	1.564	1.864	2.654	1 292	1 531.6	1 969
	7	1.398	1.772	2.524	1 372	1 593.8	1 766
	8	1.499	1.868	2.575	1 408	1 598.6	1 969
0.8	6	1.358	1.752	2.444	1 426	1 538.4	1 674
	7	1.461	1.932	2.807	1 426	1 577	1 766
	8	1.571	1.865	2.459	1 441	1 589.2	1 674
0.9	6	1.537	1.823	2.472	1 282	1 561	1 969
	7	1.524	1.816	2.433	1 292	1 467.4	1 674
	8	1.512	1.753	2.499	1 282	1 562.6	1 969

Figure 34: Experimental results: GRASP solution statistics (80 vertices, 5 runs)

α	iterations	execution time			weight		
		min	avg	max	min	avg	max
0.6	6	2.384	2.828	3.476	1 460	1 622.8	1 849
	7	2.516	2.864	3.324	1 551	1 644.8	1 849
	8	2.577	2.950	3.327	1 358	1 583.8	1 849
0.7	6	2.497	2.879	3.430	1 452	1 726.8	2 109
	7	2.474	2.735	3.154	1 555	1 708.2	1 849
	8	2.465	2.785	3.085	1 555	1 674.8	1 849
0.8	6	2.362	2.899	3.299	1 358	1 574.8	1 849
	7	2.646	2.855	3.249	1 551	1 675.6	1 849
	8	2.443	2.827	3.287	1 452	1 576.4	1 697
0.9	6	2.355	2.765	3.424	1 522	1 651.2	1 849
	7	2.525	2.998	3.653	1 630	1 714.8	1 849
	8	2.445	2.795	3.353	1 555	1 676	1 849

Figure 35: Experimental results: GRASP solution statistics (100 vertices, 5 runs)

In these tables, we show the test we did for α to determinate which alpha is the better. We move α from 0.1 to 0.9 for 10 to 100 vertices in a graph in which we run each graph 5 times. Like you can see, we decided to not show all the result because it will take so much place. However, the α we show are the best candidate.

So, to determinate which α we will take. For each α we compare them for different iterations going from 5 to 10 iterations with the execution time and weight of the final solution. We took in our comparison the minimum, average and maximum of execution time and weight. For the first table, you can see there is no big difference between the different α choose. If we look closely, we can see the best average of weight is often for 7 iterations. With 7 iterations, we often found the execution times in the middle of the other iterations. After consideration, we decided to choose 7 for Retries.

Now for α, we chose $\alpha = 0.9$ because for each graph with different number of vertices we found this α in the best average of weight and execution time. In addition, it often has the highest minimum weight. In our case, it is the best candidate.

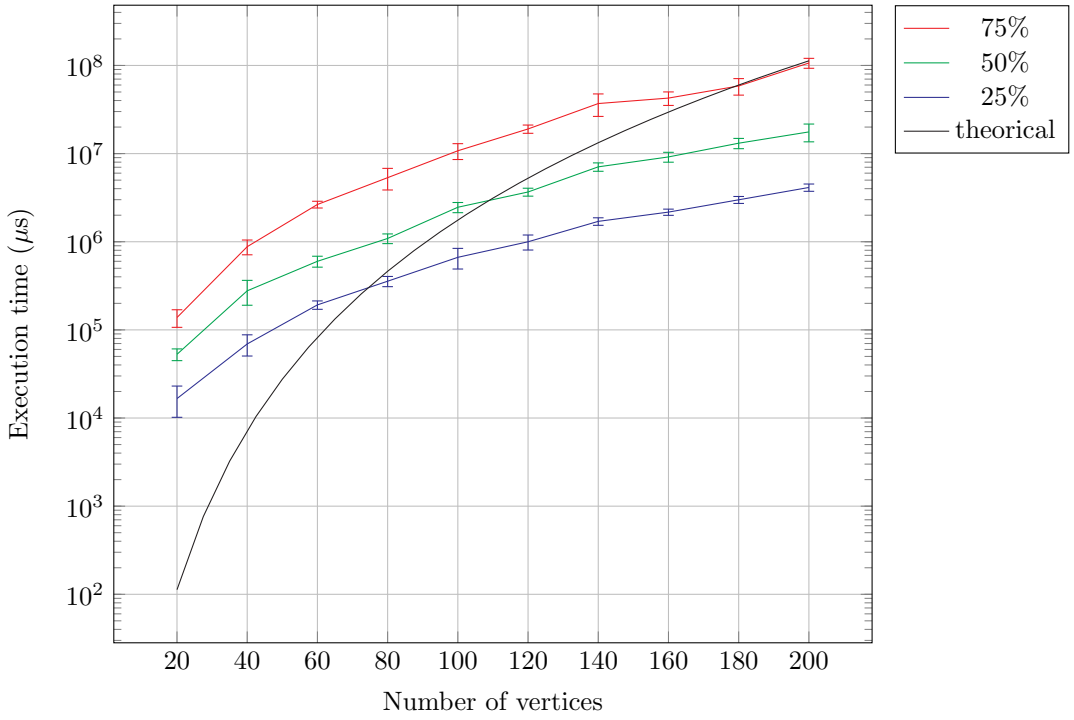


Figure 36: Execution time of the grasp algorithm for different percentages of connectivity.

For this experiment, we generated fifty random graphs with 20 to 200 vertices to find an average execution time for each percentage of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 36. At each number of vertices, an average execution time is calculated, and the standard deviation is represented by the error bars. The number of graphs generated for each percentage of connectivity and number of vertices is limited to 5 because it is a sufficient number of tests to get what we want to analyze. Moreover, we use a semi-log scale in the y-axis that allows us to highlight the results we want to analyze. If we use a linear scale we will not see the 25% of connectivity.

5.7 Analysis

With the figure 36, we can confirm the complexity $\mathcal{O}(n^6)$ we found in the complexity part. Moreover, we can see some similarity with the local search for the execution time. Where they are the same with a constant for the grasp algorithm which is the number of iteration we do to find the best solution.

We can observe in the figure 36, there are a difference between each percentage of connectivity. For the 75% of connectivity, we can observe a factor of 10^3 when we multiply by 10 the vertices and for the 25% of connectivity we have a factor of 10^2.

Instead of the exact, the Grasp algorithm has a polynomial execution time that is better than the exact, and we can use it for graph with a lot of vertices. In addition, he is more efficient than the local-search algorithm by his iteration that can give better solution than the local-search algorithm.

6 Conclusion

As seen in the previous sections, we have been able to implement different algorithm to solve the Maximum Edge Weight Clique problem. We have also been able to compare the different algorithms and to determine which one is the most efficient depending on the number of vertices and the connectivity of the graph.

Since the MEWC problem is NP-hard, it is not possible to find a solution in polynomial time. We have therefore tried to optimize the solution of the problem by trying to find the most efficient algorithm to solve the problem in the best way.

Comparison of algorithms

Speed of results

We are now going to compare the different algorithms from a general point of view, even if it would be necessary to do a study again according to the starting graph we have. Let's look at the complexity and the execution time of these different algorithms.

- **Complexity** : As we compare the complexity of the algorithms we obtain :

  Exact ($O(n^2\sqrt[3]{3}^n)$) - Constructive ($O(n^2)$) - Local Search ($O(n^5)$) - Grasp ($O(n^6)$)

  Since the complexity will vary depending on the connectivity and degeneracy of the graph we will look at the execution time too.

- **Execution time** :

  Constructive (5.10^6) - Local Search (A COMPLETER) - Grasp (10^8) - Exact ($\sim 10^8$)

Accuracy of results

Another very important criterion to consider is the accuracy of the answer obtained.

Accuracy is in this order (from most to least accurate):
Exact - Grasp - Local Search - Constructive

Choice of the algorithm

After this, we decided to conclude according to the number of vertices of our strating graph :

- **10 - 100 Vertices** : We will use the "Exact" algorithm which allows to give a very precise solution but which is slow.
- **100 + Vertices** : In order to don't wait too long for an answer, we plan to use the "Grasp" algorithm and then the "Local Search" which are certainly less precise but which allows to give an answer in less time.

- **5000 + Vertices** : Finally we will use the "Constructive" algorithm which is the least precise but the fastest and allows to answer in an acceptable time

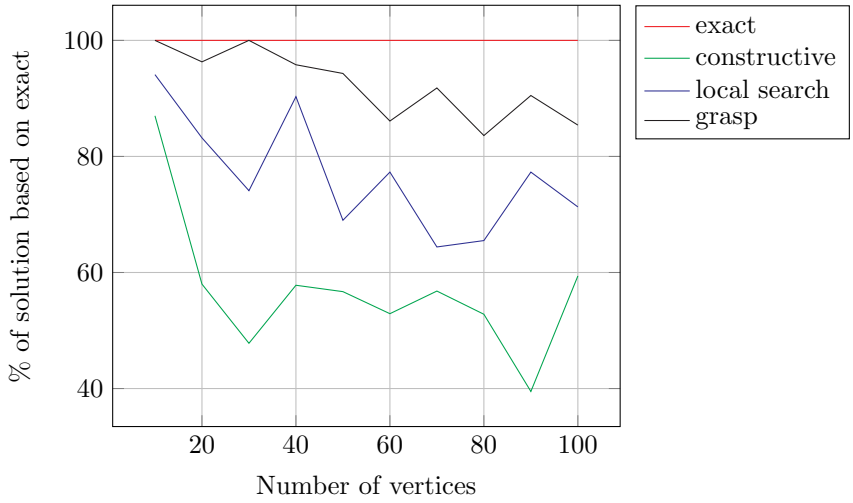


Figure 37: % of solution based on the exact of the different algorithm for 25% of connectivity.

For this experiment, we generated 10 random graphs with 10 to 100 vertices to find an average % for 25% of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 37. At each number of vertices, an average clique weight is calculated, and converted to % based on the result obtained with the exact algorithm. // Ã martin de dÃ©tailler

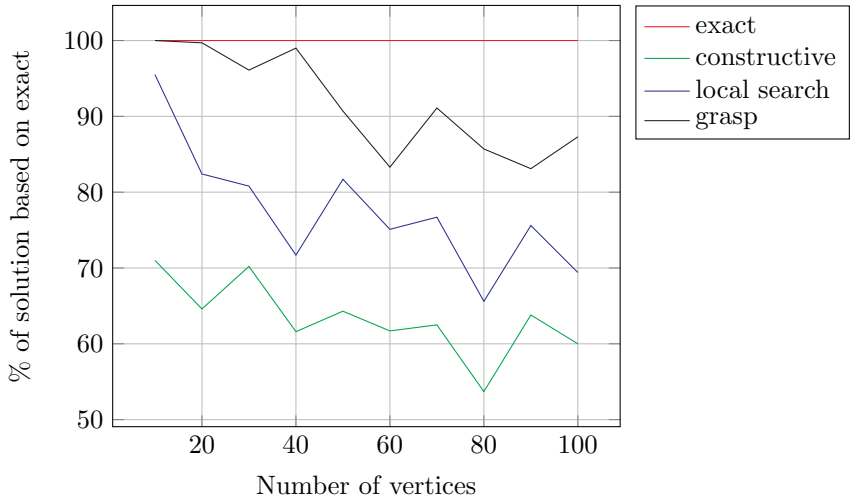


Figure 38: % of solution based on the exact of the different algorithm for 50% of connectivity.

For this experiment, we generated 10 random graphs with 10 to 100 vertices to find an average % for 50% of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 38. At each number of vertices, an average clique weight is calculated, and converted to % based on the result obtained with the exact algorithm. // Ã martin de dÃ©tailler

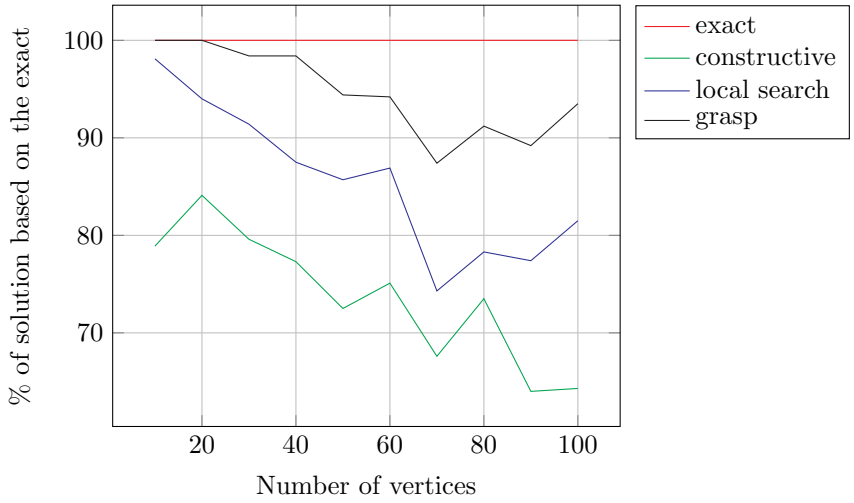


Figure 39: % of solution based on the exact of the different algorithm for 75% of connectivity.

For this experiment, we generated 10 random graphs with 10 to 100 vertices to find an average % of clique weight for 75% of connectivity. The connectivity is the percentage of chance that 2 vertices are linked. The results are shown in figure 39. At each number of vertices, an average clique weight is calculated, and converted to % based on the result obtained with the exact algorithm. // Ã martin de dÃ©tailler

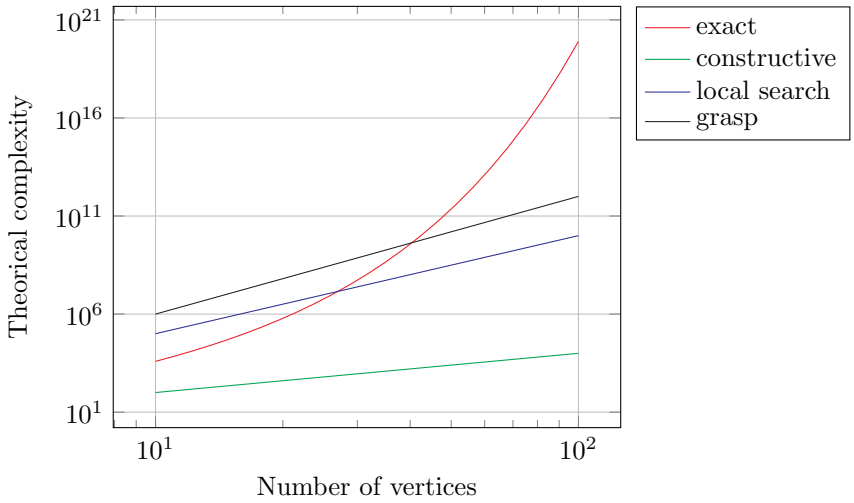


Figure 40: Execution time of the constructive algorithm for different percentages of connectivity.

In the end, we can conclude that each algorithm has its own advantages and disadvantages. The exact algorithm is the most accurate, but it is also the most time-consuming. The constructive algorithm is the fastest, but it is also the least accurate. The local search algorithm is a good compromise between accuracy and time. The grasp algorithm is the most accurate of the heuristic approaches, but it is also the slowest of them.

Generally speaking, the exact algorithm will be employed when the number of vertices is small, and the heuristic algorithms will be used when the number of vertices is large.

References

[1] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, September 1973. https://doi.org/10.1145/362342.362367.

[2] J.W. Moon and L. Moser. On cliques in graphs. *Israel J. Math.*, Match 1965. https://doi.org/10.1007/BF02760024.