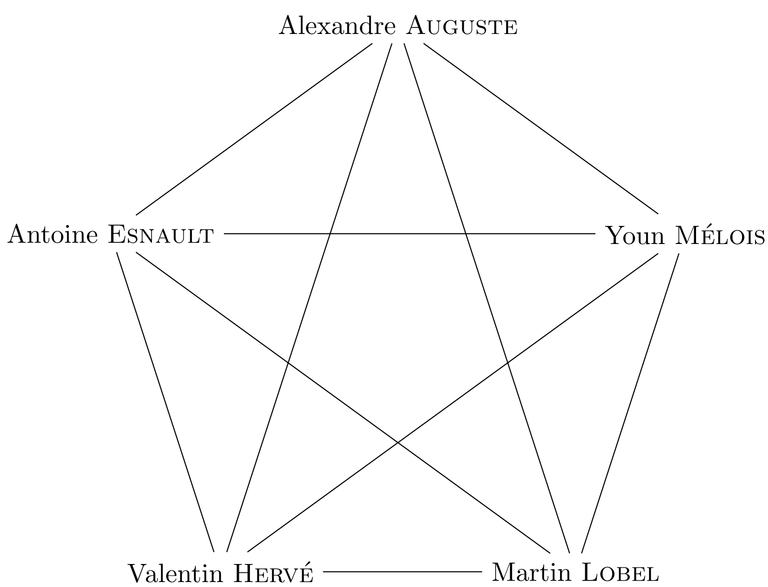
Final Project
The Maximum Edge Weight Clique Problem



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 14, 2023

Contents

1 Introduction **3**
1.1 Presentation . . . 3
1.2 Configuration . . . 4
1.3 Example of real-life situations . . . 4

2 Exact Algorithm **6**
2.1 Presentation . . . 6
2.2 How it works . . . 6
2.3 Pseudocode . . . 11
2.4 Complexity . . . 11
2.5 Bad Instance . . . 12
2.6 Experiments . . . 13
2.7 Analysis . . . 13

3 Constructive Algorithm **15**
3.1 Presentation . . . 15
3.2 How it works . . . 17
3.3 Pseudo code . . . 19
3.4 Complexity . . . 20
3.5 Instance . . . 21
3.6 Experiments . . . 21
3.7 Analysis . . . 21

4 Local Search Algorithm **22**
4.1 Presentation . . . 22
4.2 How it works . . . 22
4.3 Pseudo code . . . 22
4.4 Complexity . . . 22
4.5 Instance . . . 22
4.6 Experiments . . . 22
4.7 Analysis . . . 22

5 Grasp Algorithm **23**
5.1 Presentation . . . 23
5.2 How it works . . . 23
5.3 Pseudo code . . . 23
5.4 Complexity . . . 23
5.5 Instance . . . 23
5.6 Experiments . . . 23
5.7 Analysis . . . 23

6 Conclusion **24**

References **25**

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.

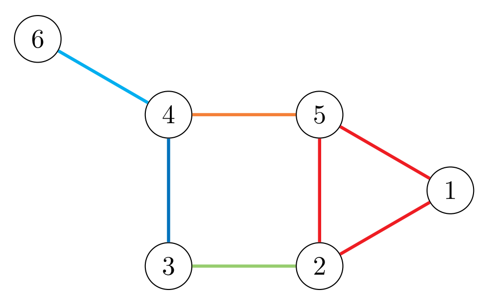


Figure 1: Basic graph example

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

- Ajouter les configurations que l'on a utilise pour les tests du style la config

For this project we used **C++** to develop and implement our algorithms. It is a general-purpose programming language that has been used for a long time and is still widely used today. It is a compiled language, which means that it is translated into machine code before being executed. This allows it to be very fast and efficient. It is also a statically typed language, which means that the type of each variable must be declared before it is used. This allows the compiler to check the type of each variable and to detect errors at compile time.

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**.

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 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.

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

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 :

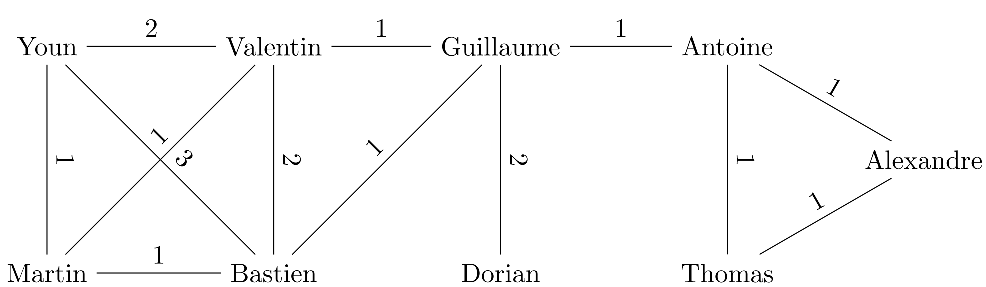


Figure 2: Graph representing the games played between the students

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:

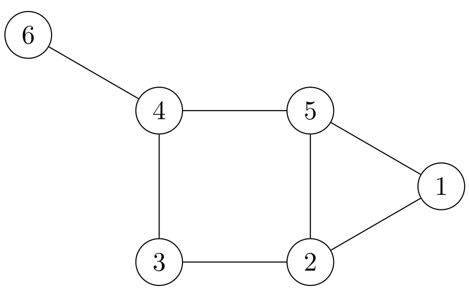


Figure 3: Graph illustration for the exact algorithm at step 0

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.

$$\boxed{R = \{\} \quad P = \{1,2,3,4,5,6\} \quad X = \{\}}$$

Step 1:

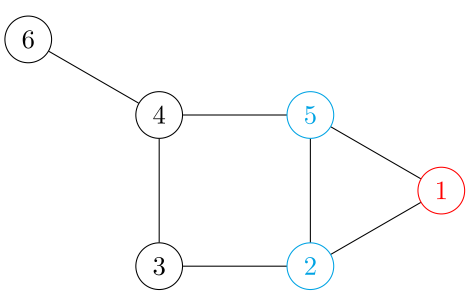


Figure 4: Graph illustration for the exact algorithm at step 1

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:

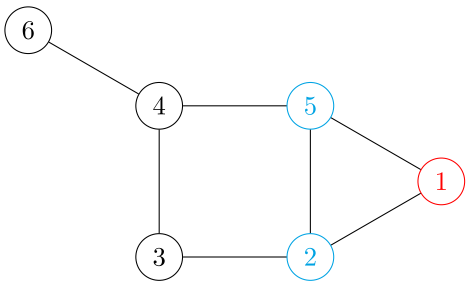


Figure 5: Graph illustration for the exact algorithm at step 2

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:

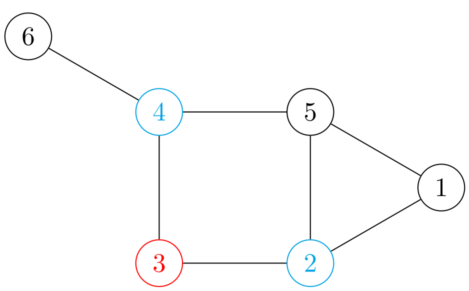


Figure 6: Graph illustration for the exact algorithm at step 3

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:

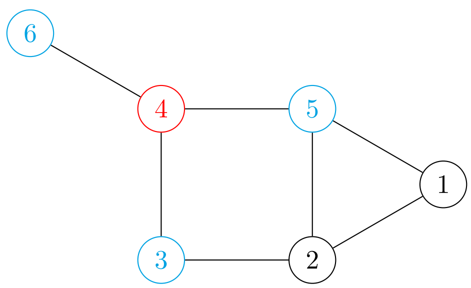


Figure 7: Graph illustration for the exact algorithm at step 4

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:

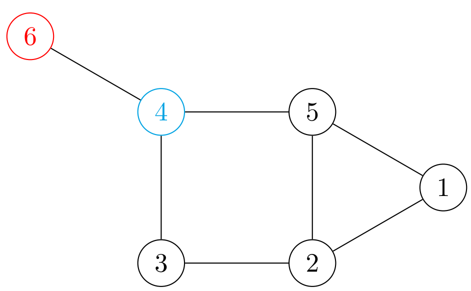


Figure 8: Graph illustration for the exact algorithm at step 5

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($\emptyset$, $\{1,2,3,4,5,6\}$, $\emptyset$)
BronKerbosch(1, $\{2,5\}$, $\emptyset$)
BronKerbosch($1,2$, $\{5\}$, $\emptyset$)
BronKerbosch($1,2,5$, $\emptyset$, $\emptyset$) : report $\{1,2,5\}$
BronKerbosch(3, $\{2,4\}$, $\emptyset$)
BronKerbosch($2,3$, $\emptyset$, $\emptyset$) : report $\{2,3\}$
BronKerbosch($2,4$, $\emptyset$, $\emptyset$) : report $\{2,4\}$
BronKerbosch(4, $\{5,6\}$, 3)
BronKerbosch($4,5$, $\emptyset$, $\emptyset$) : report $\{4,5\}$
BronKerbosch($4,6$, $\emptyset$, $\emptyset$) : report $\{4,6\}$
BronKerbosch(6, $\emptyset$, 4)

Figure 9: Call process of the Bron-Kerbosch algorithm

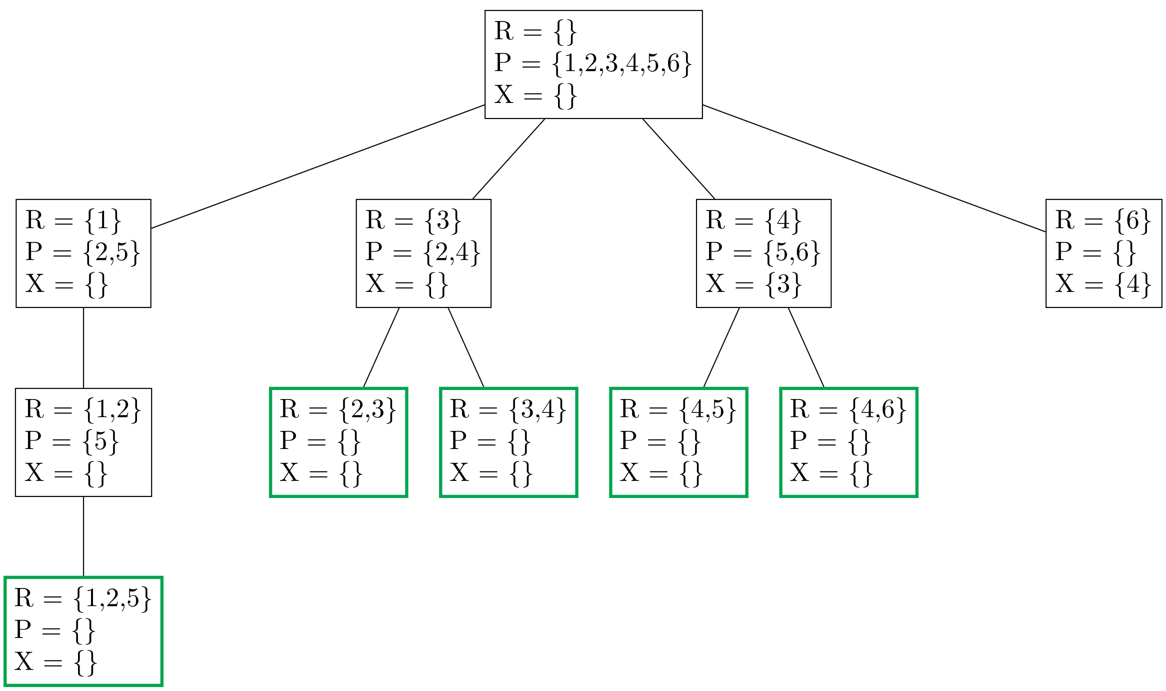


Figure 10: Call tree of the Bron-Kerbosch algorithm

2.3 Pseudocode

Algorithm 1 Exact MEWC algorithm

```
procedure ExactMEWC(G = (V, E))
    maxClique ← ∅                          ▷ Variable to store the maximum weight clique
    maxCliqueWeight ← 0
    R ← ∅                                  ▷ Set that will contain the built clique in the recursive calls
    P ← V                                  ▷ Candidate vertices
    X ← ∅                                  ▷ Excluded vertices
    maximalCliques ← ∅                     ▷ The set that will contain the maximal cliques of the graph
    BronKerbosch(R,P,X,maximalCliques)     ▷ Call the BronKerbosch algorithm to get the maximal
    cliques of the graph
    for all clique ∈ maximalCliques do
        weight ← getCliqueWeight(clique)   ▷ Get the weight of the current clique
        if weight > maxCliqueWeight then
            maxClique ← clique
            maxCliqueWeight ← maxClique
    return maxClique
```

Algorithm 2 Bron-Kerbosch algorithm

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

Algorithm 3 Clique weight function

```
procedure getCliqueWeight(C)                    ▷ We assume that the clique is complete
    return Σ_{i,j∈C, i<j} w(i, j)
```

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

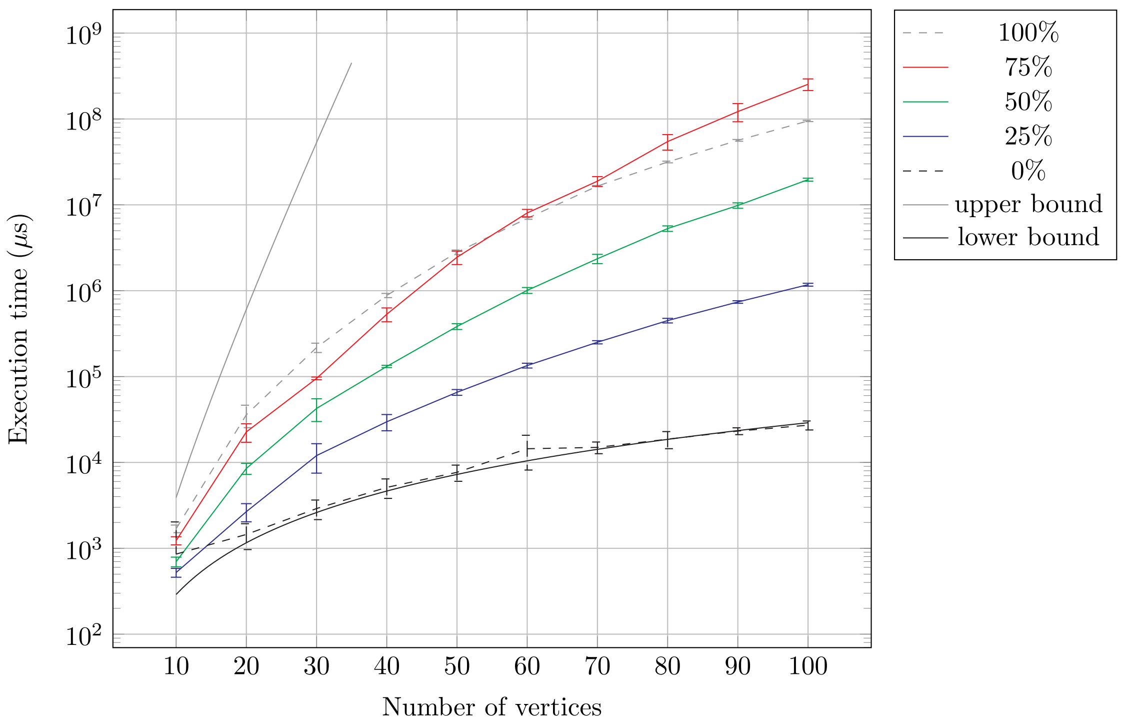


Figure 11: Execution time of the exact algorithm for different percentages of connectivity.

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 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 rules of thumb 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 exact solution to a problem, or when an exact solution would be too time-consuming to compute. They are also used when an approximate solution is good enough, or when finding an exact solution is not the primary goal. 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 add to it until a complete solution is found. They are commonly used to solve optimization problems, where the goal is to find the optimal solution, or the solution that is the best among all possible solutions.

As we said, the use of the constructive heuristic in solving the MEWC start with a partial solution and build it until a complete solution is found. To do this, the latter will be subject to a number of criteria.

1. Add The First Vertex to solution.
2. Seek for his most weighted 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). Store the weight of the edge between every members of the solution and the new vertex.
3. Repeat the step 2 until no vertices is available to add

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

In these choices of criteria, we can notice one that is quite important. The one to choose the First Vertex. Indeed, it is this one which will define the quality of our solution. Taking it randomly would be useless and counter productive for the sake of solving MEWC. 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 have 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. 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.
  - Another reason is that we can potentially reduce the search space for the rest of the algorithm. This is because we know that any clique we find must include this vertex, which means that we can eliminate any vertices that are not connected to it as potential members of the clique. This can help to make the algorithm more efficient by reducing the number of vertices that we need to consider.

We will illustrate these explanations with figure 11 and 12 which shows the advantages of each over the other.

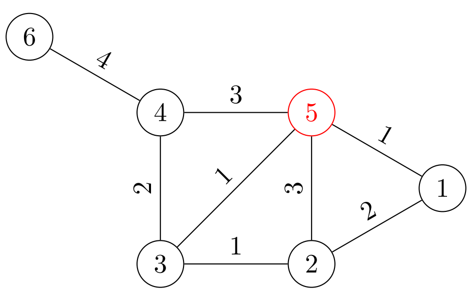


Figure 12: Graph illustration for the highest degree vertex

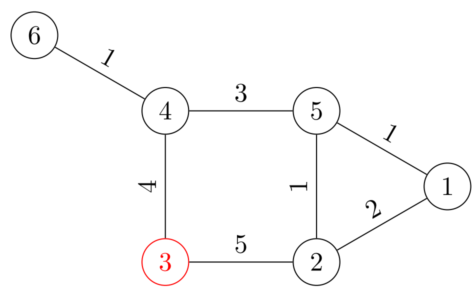


Figure 13: Graph illustration for the vertex with the highest sum of weights of the edges

We implemented both ideas and performed tests on a number of graph to see which was the most consistent and we concluded that it was the vertex with the highest sum of weights of the edges. Moreover, in most graphs (at least the randomly generated ones), we had, in general, a rather homogeneous weights of edges which worked better with this one.

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 keeps track of two groups of vertices : S which is a partially constructed (non-maximal) clique and the partial solution that we will gradually implement. Moreover, 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. Furthermore, we got W which is the total weight of S that we will implement

The algorithm begins by identifying the vertex with the highest sum of weights of its edges. If there are multiple vertices with the same maximum sum, the vertex with the highest degree is selected. The selected vertex is then added to S and its neighbors are added to P. Then the algorithm selects a new vertex from the candidates set P and adds it to S. We add to W the weight of the edge between the new vertex selected and every vertices in S. This selected vertex is the one with the most weighted edge among all the vertices in P. After this step, P is updated by considering only the neighbors of the vertices that are already part of S, and by excluding the vertices that are already part of S from P. This process is then repeated 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:

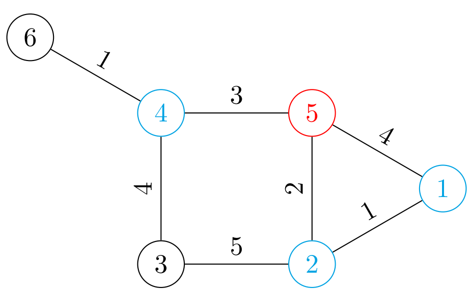


Figure 14: Graph illustration for the constructive algorithm at step 0

At the initial step, as said before, we will initialize S, P and W by searching for the vertex with highest sum of weights of its edges v by iterating every vertex on the graph. In this example, we will eventually find 3 and 5 which have a maximum sum of 9. The algorithm will then take the vertex of highest degree between them and will add it to S, represented in red. The neighboring vertex of v will be added to P, represented in blue.

$\boxed{S = \{5\} \quad P = \{1,2,4\} \quad W = 0}$

Step 1:

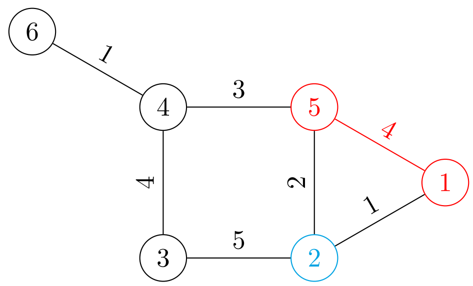


Figure 15: Graph illustration for the constructive algorithm at step 1

Then the algorithm will iterate all the neighbors of v, and look for the one who shares the edge with the highest weight, which replace v (here 1). If there are several edges with the same weight, the algorithm will take the one with the highest degree. After having found it, we add it to S as well as the weight of the edges of all the vertices in S and v to W (here 4), we update P by keeping only the common neighbors of the members of S (here only 2).

$$\boxed{\text{S} = \{5{,}1\} \quad \text{P} = \{2\} \quad \text{W} = 4}$$

Step 2:

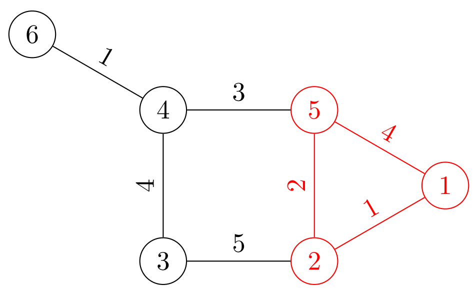


Figure 16: Graph illustration for the constructive algorithm at step 2

Then, the algorithm will repeat the process of step 1 until P is empty. It will look for a new v (here 2). After finding it, it will add it to S as well as the weight of the edges of all the vertices in S and it to W (here 2 + 1). It will then update P which here will be empty, the algorithm stops and we get the maximum clique in red and his weight (here (5,1,2) of weight 7).

$$\boxed{\text{S} = \{5{,}1{,}2\} \quad \text{P} = \{\} \quad \text{W} = 7}$$

The constructive algorithm is now finished and we have obtained the following maximum clique of weight 7 :

$$(1, 2, 5)$$

3.3 Pseudo code

Algorithm 4 Constructive algorithm

```
procedure Constructive(G = (V, E))
    S ← ∅                                        ▷ Variable to store the clique that represents the Solution
    Sweight ← 0                                  ▷ Variable to store the total weight of S
    P ← ∅                                        ▷ Lists of vertex to store the vertex that could be added to S
    FirstVertex ← getFirstVertex(G)              ▷ Call a function to get the vertex of highest degree
    addVertex(S, FirstVertex)                    ▷ add the FirstVertex to S
    lastVertex ← FirstVertex                     ▷ Variable to store the last element added to S
    getPotentielCandidate(G, FirstVertex, P)     ▷ add to P all the neighbors of FirstVertex
    while P.size() ≠ 0 do
        updateClique(G, lastVertex, P, R)        ▷ Call the function that update S
        updatePotentielCandidate(G, lastVertex, P)   ▷ Call the function that update P
```

Algorithm 5 First Vertex function

```
procedure getFirstVertex(G)
    bestVertex ← ∅                               ▷ Variable to store the bestVertex found
    MaxNumNeighbors ← 0                          ▷ Variable to store the max number of neighbors found
    vertices ← getVertices(G)                    ▷ get the lists of every vertex of G
    for vertex ∈ vertices do                     ▷ Iterate over all the vertices in G
        if ||N(vertex)|| > MaxNumNeighbors then
            MaxNumNeighbors ← ||N(vertex)||      ▷ replace the maximum size
            bestVertex ← vertex                  ▷ replace bestVertex with the vertex
    return bestVertex
```

Algorithm 6 getPotentielCandidate function

```
procedure getPotentielCandidate(G,lastVertex,P)
    for neighbor ∈ N(lastVertex) do
        P.insert(neighbor)                       ▷ Add the vertex to P
```

Algorithm 7 updateClique function

```
procedure UPDATECLIQUE(G,lastVertex,P, S)
    weight ← 0                    ▷ Variable to store temporary the weight of each edge of lastVertex
    bestWeight ← 0                ▷ Variable to store the max weight found
    CliqueWeight ← 0              ▷ Variable to store the weight that we will add to the clique
    bestVertex ← ∅                ▷ Variable to store the bestVertex found
    for vertex ∈ P do
        if vertex = lastVertex then
            continue
        else
            edge ← G.GETEDGE(lastVertex, vertex)     ▷ get the edge between lastVertex and vertex
            weight ← edge.GETWEIGHT()    ▷ get the weight of the edge between lastVertex and vertex
            if weight > bestWeight then ▷ if the weight of vertex is better than the best weight we found
                bestWeight ← weight                   ▷ replace the bestWeight found
                bestVertex ← vertex                   ▷ replace the bestVertex found
                weight ← 0                            ▷ reset weight to 0
            else
                weight ← 0                            ▷ reset weight to 0
    vertices ← S.getVertices()        ▷ Variable to store a list of the vertices of S
    for v ∈ vertices do   ▷ For each v in vertices, add the weight of it and bestVertex in cliqueWeight
        edge ← GRAPH.GETWEIGHT(v, bestVertex)
        cliqueWeight ← cliqueWeight + edge.GETWEIGHT()
    ADDVERTEX(S, bestVertex)                          ▷ add bestVertex in S
    lastVertex ← bestVertex           ▷ replace the lastVertex of S by bestVertex
    ADDWEIGHT(S, cliqueWeight)        ▷ add the weight that bring the new vertex in S
```

Algorithm 8 updatePotentielCandidate function

```
procedure UPDATEPOTENTIELCANDIDATE(G,lastVertex,P)
    neighbors ← ∅                     ▷ Variable to store the neighbors of lastVertex
    for all vertex ∈ N(lastVertex) do     ▷ add every neighbors of lastVertex in neighbors
        neighbors.insert(N(vertex))
    for v ∈ P do
        if neighbors.count(i) = 0 then              ▷ if v ∉ neighbors, remove v in P
            P.erase(v)
        else if lastVertex = v then                 ▷ if lastVertex ∈ P, remove it
            P.erase(v)
```

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 generating a matrice adjacent 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.

3.5 Instance

3.6 Experiments

3.7 Analysis

4 Local Search Algorithm

4.1 Presentation

4.2 How it works

4.3 Pseudo code

4.4 Complexity

4.5 Instance

4.6 Experiments

4.7 Analysis

5 Grasp Algorithm

5.1 Presentation

5.2 How it works

5.3 Pseudo code

5.4 Complexity

5.5 Instance

5.6 Experiments

5.7 Analysis

6 Conclusion

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.