ISEN
ALL IS DIGITAL!

yncréa

Final project- graph

The Minimum Bisection Problem

Dorian RENA – Florent BUTERY – Esteban FANJUL – Quentin FRANQUET – Cassie PERIDY
CIR3
Team 5

Number of words :

☑ By sending this document to the teacher, we certify that this work is our and that we have read the rules relating to referencing and plagiarism.

Table of Contents

Introduction 4
Applications in real life 5
Destabilization in times of war: 5
Football team: 5
Construction company team: 6
Exact Algorithm 6
Improvements : 7
Pseudo-Code : 8
Complexity : 10
Analyze of the performances of the algorithm : 11
Constructive Heuristic Algorithm 12
Why Constructive Heuristic : 12
What criteria : 12
Why those criteria : 12
Pseudo-code : 14
Complexity : 15
Analyze of performance of the algorithm : 15
Local Search Heuristic Algorithm 16
Pseudo code : 20
Explanation: 20
Complexity: 21
Analyze of performance of the algorithm : 21
Meta-Heuristic Algorithm 22
Operating of the algorithm: 22
Pseudo-code : 24
Complexity: 26
Analyse of performance of the algorithm : 28
Choice of parameters & use case of the algorithm: 29
Comparison of the algorithms: 30
Executions of the algorithms: 33
Test 1 : 33

Test 2 : 34
Test 3 : 34
Conclusion 35

Introduction

The objective of this project is to study the Minimum Bisection Problem (MBP). Concretely, starting from a given graph, the MBP allows to separate its vertices into two groups of equal sizes while trying to minimize the number of edges linking the two groups. We start from a simple non-oriented graph, $G = (V, E)$ with an even number of vertices.

Since the Minimum Bisection Problem is a N-P hard problem where it has been proven that it is difficult to find an optimal solution, the following procedure was followed:

First, we used the exact algorithm to solve this problem. Since the exact algorithm takes a long time to complete, we will use the heuristic method to try to get as close as possible to an optimal solution but in a shorter time.

We therefore used an algorithm called the constructive heuristic which allows us to build a solution based on "smarts" criteria.

We then moved to the local search heuristic, which, starting from a given solution, generally given by the constructive heuristic, makes it possible to look for a better solution by visiting the neighbors of the summits in question. It therefore only finds a local optimized solution like its name says.

Finally, we finished with the meta-heuristic algorithm, more precisely the tabu-search, that calls the previous algorithm with randomization bounded by different criteria.

Then, we will present in more detail the possible applications of this problem in real life. Then, each algorithm whose pseudo code is given will be explained in more detail and compared.

To carry out this project, we had to use different tools.

First of all, our project was realized in C++ language using the Clion development environment (IDE) by JetBrains.

In addition, to be able to realize this project, since we were 5 working on it, we need a way to work on it on our side and be able to share the code. We therefore decided to use GitHub to realize this project.

Without forgetting, LaTeX which allowed us to realize the proper formatting of our various pseudo-code present throughout this report, and the graph online site that allowed us to draw the different graphs.

Furthermore, to communicate between each of us, we use the Discord group which allows an easy way to communicate with both computer and smartphone.

We also used to display the data of the different algorithms in graphics GNU plot 5.4 and therefore our data is stored in .csv files.

Finally, we decided to run the different algorithms on the same computer that had nothing else open without interacting with it during executions. The computer has an Intel core i7 processor with 16GB of ram and was plugged into the wall.

It should be noted that all the algos were run with the same input graphs so that there is no error where one graph has a weaker solution than another. We made for each execution an average of the time taken by the algos and an average of the solutions obtained, for the 10 same graphs (same number of vertices and probability).

Applications in real life

The MBP can be used in several concrete cases, as we shall see in a few examples.

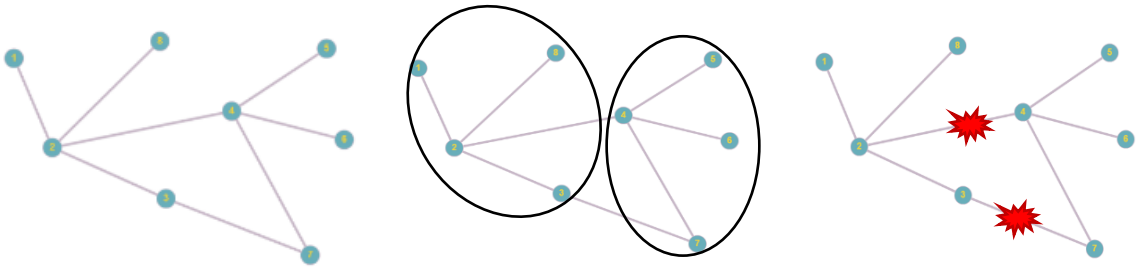
Destabilization in times of war:

In a war context, country A can model country B with its major cities and the connections between them, apply the MBP to see the most strategic links to destroy to cause trouble in the target country and divide the country in 2. This can be used to block food or military supply transactions.

Vertex : Cities

Edges : Roads, train tracks

Direction : No (because the edges go in both directions)



Schema destabilization in times of war

Football team:

Let's imagine that, in the context of football, a coach wants to divide his squad into two distinct teams while minimizing the interactions between these two teams. This can be modelled as an MBP. This can be useful if the 2 teams are not playing on the same evening, and you don't want team 1 or 2 to disturb the players of the other team who are mentally preparing for the day of the match.

Vertex : Each player in the football squad.

Edges : Interactions, communication, or collaboration between players.

Direction : No (because the communication is bidirectional).

Construction company team:

Imagine that a company in the field of construction seeks to organize its team to be on 2 remotes sites at the same time. The goal is to divide the staff into two teams to minimize the number of interactions required between these two teams to solve the problems of a worksite. A carpenter can help a plumber solve a problem and vice versa while a plumber cannot help another plumber on a carpentry problem.

Vertex : Every person is represented.

Edges : Interactions or collaborations between technicians, representing the possibility of sharing knowledge or working together on problems, are represented by edges.

Direction : No (because collaboration is often reciprocal)

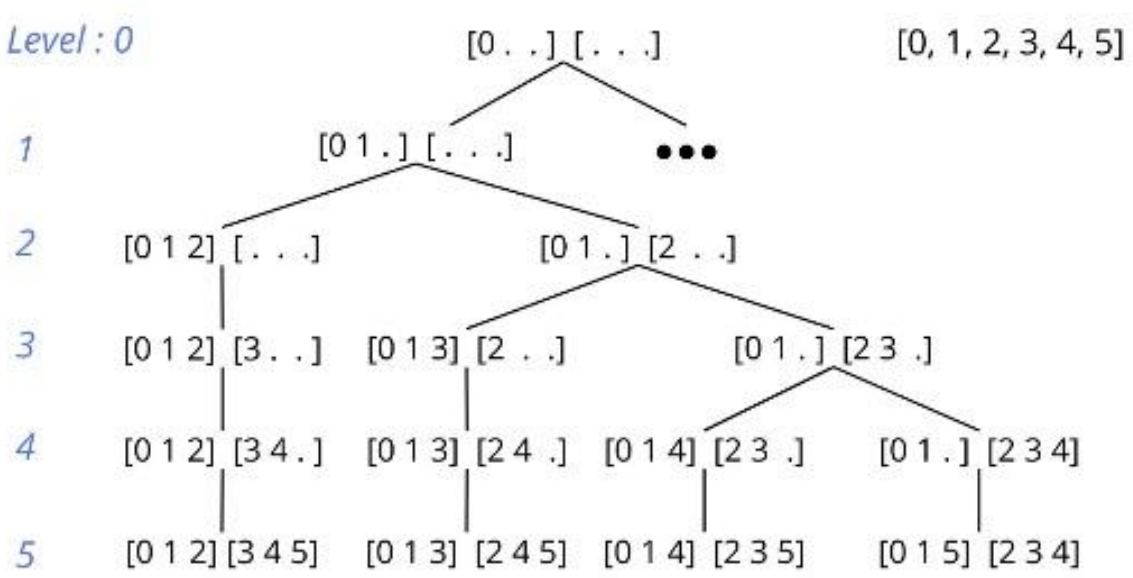
The solution to this problem could be the creation of two separate maintenance teams, where communication and internal collaboration within each team is encouraged, while interactions between the two teams are minimized.

Exact Algorithm

The objective of the exact algorithm is to find a solution to the problem by trying all possibilities. For an array containing all vertices, the algorithm tests all possible combinations that split into two arrays of fair sizes. The number of edges linking the two groups based on adjacency lists is then calculated for each combination.

To reduce the execution time, we decided to improve our algorithm by stopping the execution when we cannot find a better solution in the following combinations.

To better understand this algorithm, we can represent all possible combinations by a binary tree where each node corresponds to the 2 sub-arrays that we will fill in as we go. The root is the first element of the array. At each level, if we go through the right side, we add the following element in the right sub-group and the same for the left side, we fill the left sub-group. When a sub-group is filled, then there is only the possibility to add in the other sub-group, so we remove a side. The following tree is then obtained:



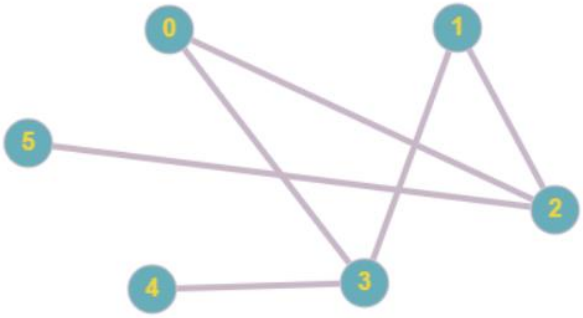
Representation of combinations with a binary tree

Improvements :

To have all the possibilities, it is possible to both start by putting the 0 (first element) in the group on the left or in the group on the right and to take into account the place of the vertices in the table. For example, [0, 1, 2] [3, 4, 5] is not equal to [3, 4, 5] [0, 1, 2] which is not equal to [4, 3, 5] [0, 2, 1]. But in the case of our problem, the place of the root in the groups does not matter as well as the order of the vertices in each group so we can already reduce the number of possible combinations. Therefore, we decided to put the 0 in the left group.

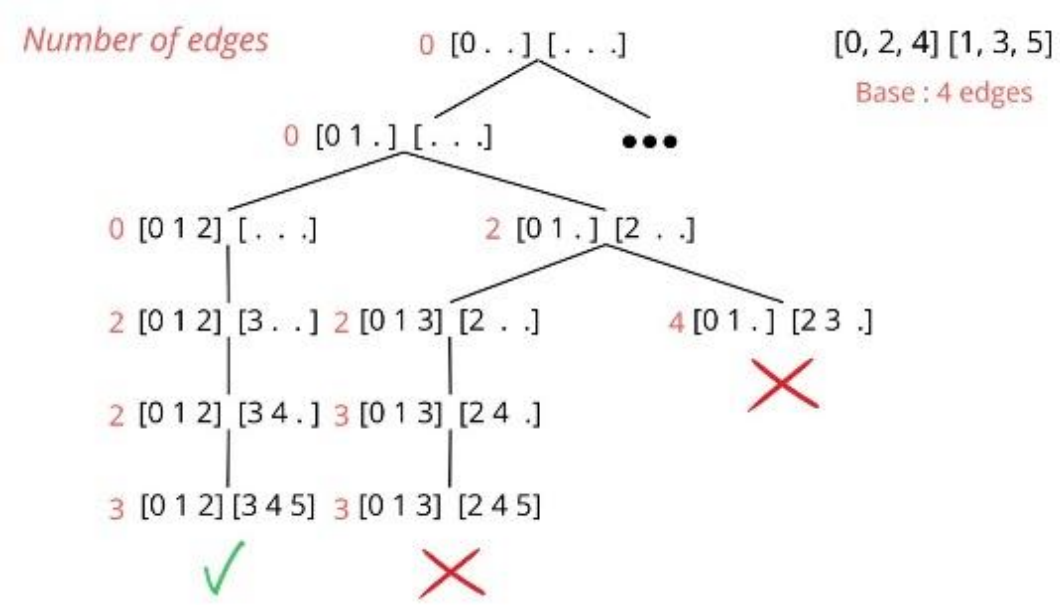
To further improve our algorithm, we decided to start from a base case by separating our array of vertices according to the even or odd indices and then calculating the number of edges linking the two groups. We then relied on this case to eliminate possibilities and thus avoid the need to browse the entire tree.

Let's take as an example the graph if below :



Exemple graph exact algorithm

We have our basic case: [0, 2, 4] [1, 3, 5] which has 4 edges. If, when going through our tree and before reaching the end of the branch, we realize that the number of edges linking the two groups is greater or equal to the base case then we do not need to continue this branch. If we find a better combination, and both groups are fully populated, then we modify our base case to keep the new best case. In the course of our tree, it would give this :



Improved exact algorithm illustration

First, the left branch is traversed as long as the number of edges is not greater than the base case. Arriving at the leaf, we realize that this combination is better than the first, so we update the base case. We then continue to walk the tree in the same way. When we arrive in the right side of our sub-tree, we realize that with the beginning of combination [0, 1, x] [2, 3, x] we already have a higher number of edges than our base case (4 >= 3) so we do not need to continue on this branch.

Pseudo-Code :

We chose to implement two different functions. First, exactAlgorithm, allows us to create our base case and initialize the tree. The second allows us to scan the tree recursively and find the best combination.

Algorithm 1 exactAlgorithm(Graph G)

```
1: n size of G
2: A, B arrays of n/2 values
3: A_init, B_init arrays of n/2 values
4: result array of 2 arrays
5: if n is odd then
6:     return Error
7: end if
8: for each vertex in G do
9:     if index is even then
10:        A_init.push_back(vertex)
11:    else
12:        B_init.push_back(vertex)
13:    end if
14: end for
15: min ← number of edges linking A_init and B_init
16: A[0] ← 0
17: result[0] ← A_init
18: result[1] ← B_init
19: exactAlgorithmVisit(G, min, A, B, 1, result);
20: return result
```

$O(n^2)$

$O(2^n \times n^2)$

Pseudo-code exactAlgorithm

Algorithm 2 exactAlgorithmVisit(Graph G, min, A, B, i, $result$)

```
1: n size of G
2: A_bis ← A
3: B_bis ← B
4: if i = n then
5:     min ← number of edges linking A and B
6:     result[0] ← A
7:     result[1] ← B
8: else
9:     if A_bis < n/2 then
10:        A_bis.push_back(vertex)
11:        if min > number of edges linking A_bis and B then
12:            exactAlgorithmVisit(G, min, A_bis, B, i + 1, result)
13:        end if
14:    end if
15:    if B_bis < n/2 then
16:        B_bis.push_back(vertex)
17:        if min > number of edges linking A and B_bis then
18:            exactAlgorithmVisit(G, min, A, B_bis, i + 1, result)
19:        end if
20:    end if
21: end if
```

$O(n^2)$

$O(2^n)$

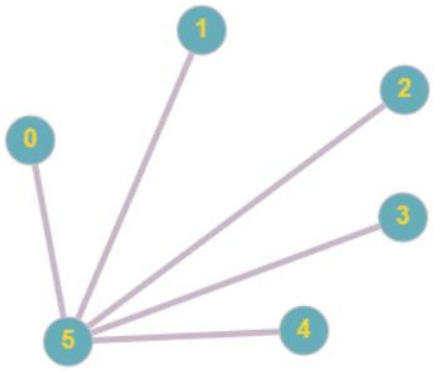
$O(n^2)$

$O(2^n)$

Pseudo-code of exactAlgorithmVisit

Complexity :

For the exact algorithm, the worst case would be to browse the entire graph to find the best solution. We could find ourselves in this case, for example, if we have a graph of the type:



The worst case for the exact algorithm

For this graph, we will have a base case with a number of edges equal to 3. But when we go through the branches, we will not know the number of edges common to the two subgraphs before arriving at the bottom of the graph.

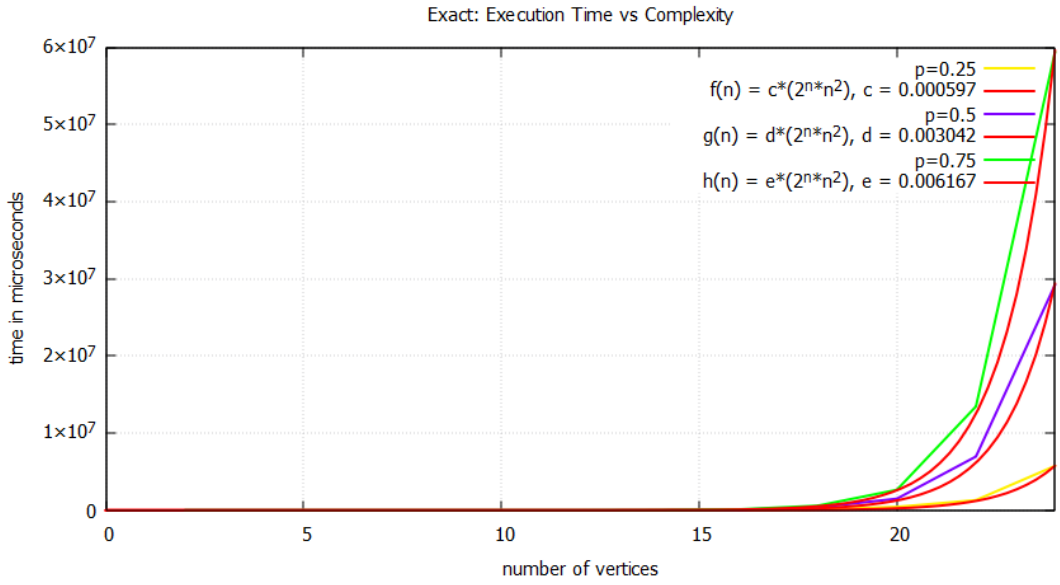
In the first algo, when we initialize the variable min, we have a complexity of $O(n^2)$ because we use two loops, one that traverses the sub-array A and the other to traverse B to retrieve the number of edges between the two sub-arrays. To this complexity is added the call of the exactAlgorithmVisit function.

For the second algo, in each *if* corresponding to the path of the left or right sub-tree, we first have a complexity of $O(n^2)$ to calculate the number of edges between the two groups. Then we have the recursive call which corresponds to the path of each sub-tree, which has a maximal complexity of $O(2^n)$ which corresponds to the number of possible combinations. In total we have a complexity of $O(n^2 \times 2^n) + O(n^2 \times 2^n) = O(2 \times n^2 \times 2^n) = O(n^2 \times 2^n)$ for the exactAlgorithmVisit function.

In the end we have:

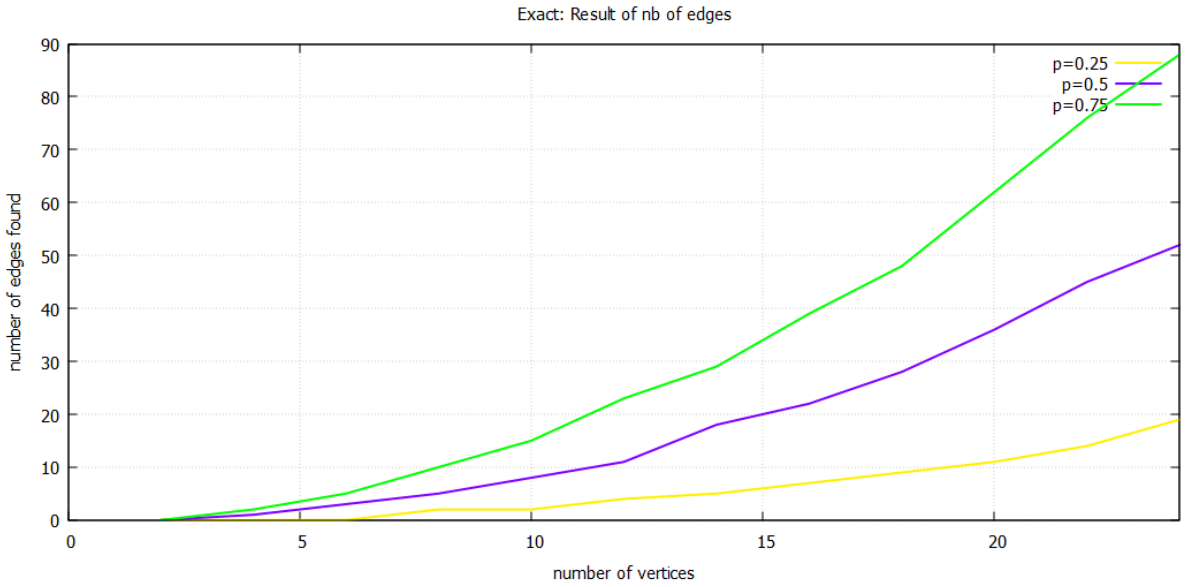
$$O(n^2) + O(n^2 \times 2^n) = O(n^2 \times 2^n)$$

Analyze of the performances of the algorithm :



1 Curves of the exact algorithm – Execution time vs Complexity

We can observe different curves of the execution time in microseconds of the exact algorithm for three chosen probability. Indeed, it was chosen that a vertex is connected to another by an edge have a probability of 25%, 50%, and 75%. We can also observe the complexities of the three probability curves, and we notice that they correspond well to what has been said above. The execution time of the algorithm follows the curve of the complexity $O(n^2 \times 2^n)$. Also, we see that the higher the probability of edge, the longer the execution time increases.



2Curves of the exact algorithm – Result of number of edges

The exact algorithm being the one that allows to obtain the minimum possible number of edges that there can be between two subgroups of vertices of the same size in a graph. This graphic represents the number of edges found for each n. And as for the execution time, the more the probability of edge of the graph increases, the more the number of edges found increases.

Constructive Heuristic Algorithm

Why Constructive Heuristic :

While the exact algorithm is great to obtain the best solution, it has a major flaw, it is extremely slow. Therefore, we can improve the time complexity but at the cost of precision using the Constructive Heuristic Algorithm (CHA). The CHA works by defining a solution based on some criteria but without seeking for an exact solution, it aims for an approximate solution. The main goal is to earn some time in tasks where precision is not what matters the most, but time is.

In the case of the MBP, the exact algorithm is unusable on graphs with more than 50 vertices. In consequence it is acceptable to have a solution that is close to the actual solution but that runs in a fraction of the exact solution times.

Let's take a look at the case of a war mentioned above. In a war you need to be able to take a decision rapidly. While the exact algorithm can give you the best way to split the country in two by destroying the less connection possible, it might take until the end of the war to give you a solution, while having to destroy more connection but knowing it at the start of the war is more useful and usable.

What criteria :

To be able to code the CHA, we first need to decide which criteria to use. In the case of the MBP, we decide to get the vertex of maximal degree and to put it in the group where it creates the less links between the two groups. After that, we take the x, x being half the vertex's degree, vertices that are not neighbors of this vertex and of minimal degree, that are not already put in one of the two groups, and we put them in the other group, if it's not full.

Why those criteria :

Firstly, we choose a vertex of maximal degree to help improve the precision of the algorithm. Since we put the non-neighborhood vertices, having a greater degree allow to put more vertices in the other group that will not create link with it.

Secondly, we decide in which group we put that vertex to take the best decision at this moment by reducing the number of edges linking the two groups.

Finally, we put x vertices in the other group of minimal degree that are not neighbor of the vertex because taking the minimal degree help avoiding creation of links between the two groups without having to check each vertex and get the ones that minimize the links.

That help reducing the time needed to run the algorithm. We could have chosen to put in the same group half the neighbors of maximal degree, but it would have reduced the precision because doing the previous steps with greater degree vertex give a more precise response as seen before. And we only take half of the vertices to help handle special cases were doing so is actually not a good idea furthermore, it avoids filling one group only using one vertex.

Pseudo-code :

Algorithm 1 constructiveHeuristic(Graph G)

```
1: n size of G
2: result array of 2 arrays
3: degree array of pairs of n values
4: used array of boolean of n values
5: for each vertex in G do degree.push_back(degree of vertex, vertex)
   used[i] ← false
6: end for
7: Sort of degrees
8: for each i from 0 to n − 1 do
9:     g ← degrees[i].second
10:    if not used[g] then
11:        edges0 ← number of edges linking result[0] and g
12:        edges1 ← number of edges linking result[1] and g
13:        if size of result[1] = n/2 then result[0].push_back(g)
14:        else
15:            if edges0 ≥ edges1 and size of result[0] < n/2 then
16:                result[0].push_back(g)
17:                count ← 0
18:                deg ← degrees[i].first
19:                j ← n − 1
20:                while j > i and count < deg/2 do
21:                    h ← degrees[j].second
22:                    if size of result[1] < n/2 and not used[h] then
23:                        if there isn't an edge between g and h then
24:                            result[1].push_back(h)
25:                            count + +
26:                            used[h] ← true
27:                        end if
28:                    end if
29:                    j − −
30:                end while
31:            else
32:                result[1].push_back(g)
33:                count ← 0
34:                deg ← degrees[i].first
35:                j ← n − 1
36:                while j > i and count < deg/2 do
37:                    if size of result[0] < n/2 and not used[h] then
38:                        if there isn't an edge between g and h then
39:                            result[0].push_back(h)
40:                            count + +
41:                            used[h] ← true
42:                        end if
43:                    end if
44:                    j − −
45:                end while
46:            end if
47:        end if
48:        used[g] ← true
49:    end if
50: end for
51: return result
```

3Pseudo-code constructive heuristic

The principle of the CAH pseudo-code using our criteria is quite simple. First you create an array of two arrays to store the resulting two groups, you also create an array that contain pairs of vertex's degree and the vertex then you sort it accordingly to the degree to be able to iterate throw vertices of max degree. You also need an array called used that tell you if you already have put a vertex in the resulting solution, this array is initialized with false.

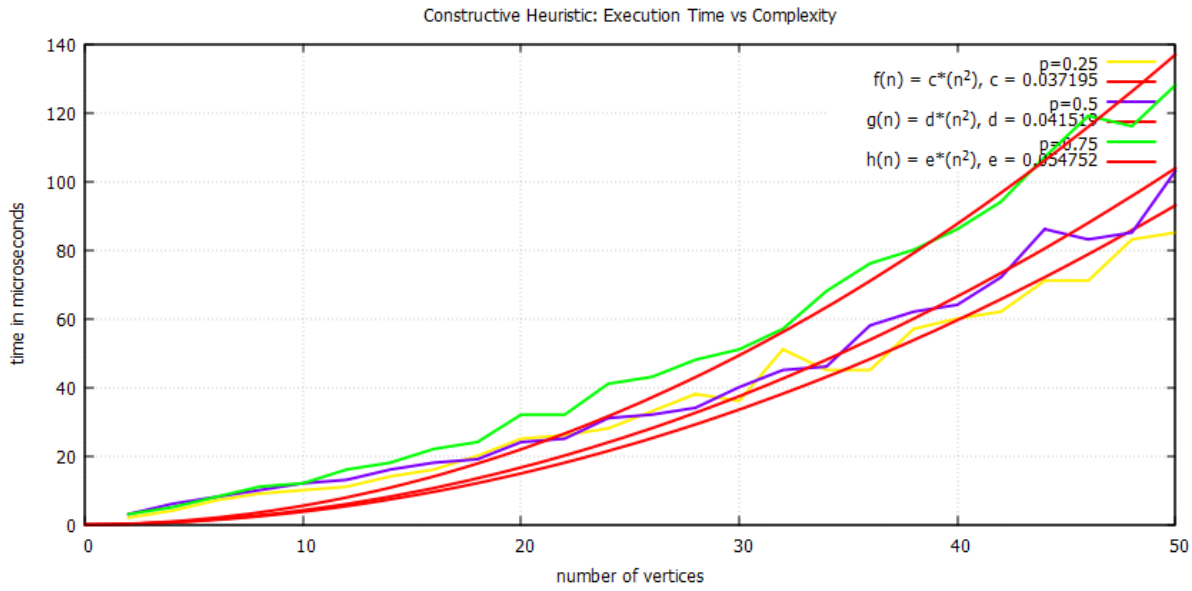
Then, iterate through each one of the vertices in the degrees array. For each one of those vertices, that are ordered by maximal degree, if the vertex has not already been used, put it in the appropriate group according to the second degree seen early, which is where it creates the less links with the other group. Next, take half its degree vertices of minimal degree that are not linked with it and are not used and put them in the other group.

Complexity :

As we can see in the pseudo-code, to make the CHA, we need to, first, get a descending ordered array of the vertices by the vertex's degree, which is in $O(n \times log(n))$. Then, we loop over each vertex, which is in $O(n)$, and for each vertex that has not already been use we put half of its degree vertices into the other group, which is $O(n)$, since the maximum degree possible is n-1 and getting the degree of a vertex is done in $O(n)$. So, the total complexity is $O(n \times log(n) + n \times n) = \mathrm{O}(\mathrm{n}^2)$.

The time complexity in the worst case is $\mathrm{O}(\mathrm{n}^2)$.

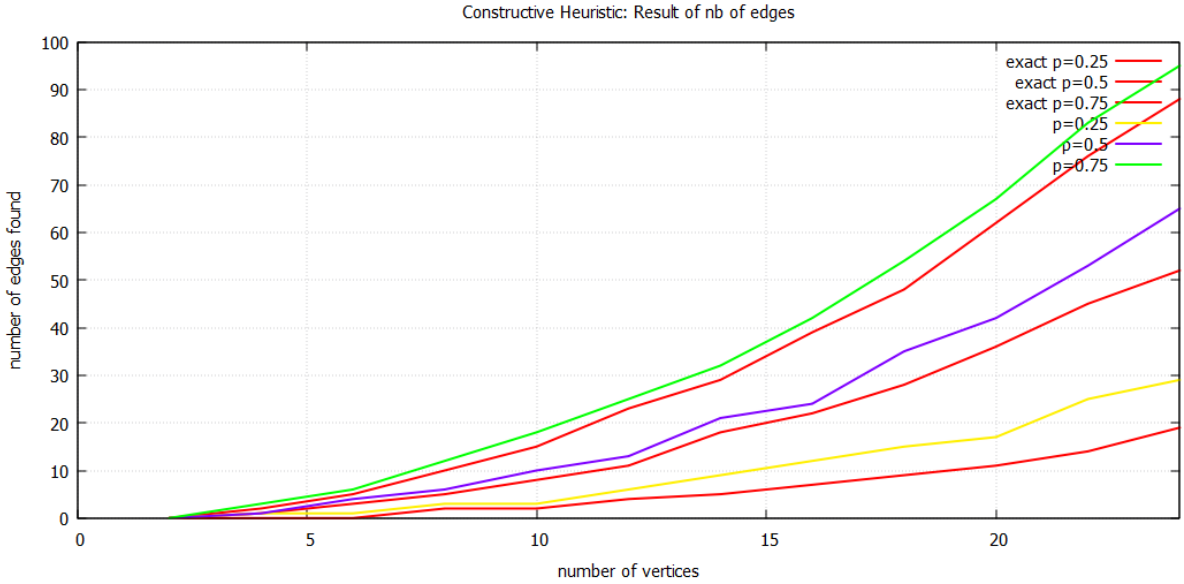
Analyze of performance of the algorithm :



4Curves of the constructive heuristic – Execution time vs Complexity

The above graphic shows us how the CHA performs in time on average compared to the worst case.

As we can see, the algorithm has a time complexity that isn't growing as fast as the worst case. The above graph also shows that the time needed to execute the CHA depends on the density of the graph.



5Curves of constructive heuristic - Result of number of edges

On this graphic we can observe the evolution of the number of edges for the found solutions according to n. The exact algorithm finding the best solution, we compare this algo and the constructive thanks to the curves found for each probability of edge. We can directly notice that the curves for the same probability do not overlap, demonstrating that the heuristic constructive is an algorithm that doesn't always find the best solution but tries to get close while being as fast as possible.

We can also observe that the greater is the probability, the closer is the constructive heuristic to the exact algorithm.

Local Search Heuristic Algorithm

The LocalSearch (is name localHeuristic in our project) is an optimization algorithm that explores the space of solutions by moving from one solution to another in the neighborhood, in the hope of finding a better solution. So, let's see how our Local Search works for our MBP.

The first step is initialization. At this point we can take what we want as an initial solution, however this is not the most optimized. In our project we decided to do this initialization as an argument of our function which permits us to reuse it more easily. For example, in the MetaHeuristic we use a random value to call local search. This will be detailed below in the part dedicated to this algorithm. For our case and our tests, we will look at an initialization with the result of the ConstructiveHeuristic. This gives us already a solid base (the solution is already not bad) that we will improve with the local.

Then come the evaluation which consists in calculating the quality of the current solution using an evaluation function. In the context of MBP, the quality of a solution is measured by the number of edges between the two sets.

On the graph below the evaluation of initialization is represented by the red cross.

The following steps will be performed in a loop.

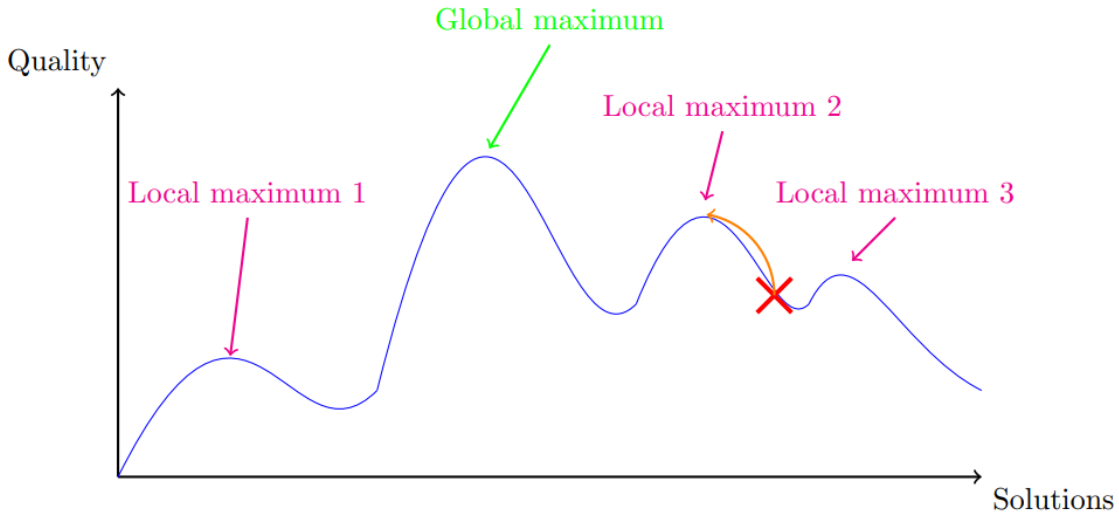
We must determine the neighborhood of the current solution. For the MBP, a typical neighborhood is to exchange a vertex of one set with a vertex of the other set. Each neighbor corresponds to a slight modification of the current solution.

So, we must explore the neighboring solutions by making simple movements and for each one, reassess the quality.

Then it is necessary to select a neighbor that improves the quality of the current solution. The selection criterion can be based on the best possible improvement. We will see the criterion we took later by explaining the pseudo code.

And finally, if the selected neighbor offers an improvement, update the current solution with that neighbor and redo these steps.

Otherwise, it means that the algorithm has reached a local maximum (the displacement performed can be represented by the orange arrow below).

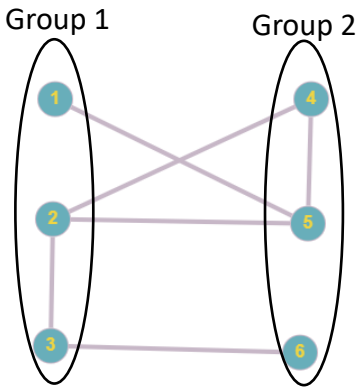


6 Illustration of local search heuristic algorithm

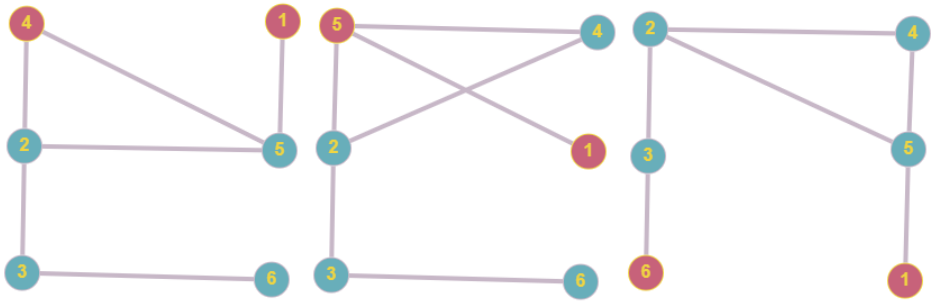
This graphic shows the quality according to the different iterations (solutions). Each iteration comes down to changing both sets.

Let's see more visually how it works.

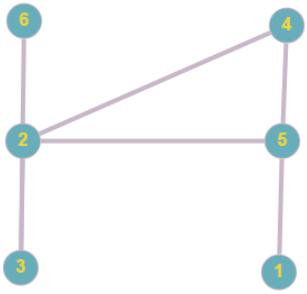
Let's start with this graph.



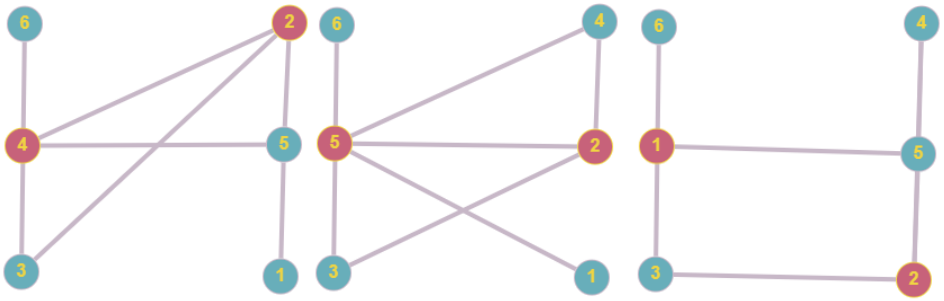
So we take the first vertex of the group 1 here the vertex 1 and compare it by exchanging it to the vertex of the group 2.



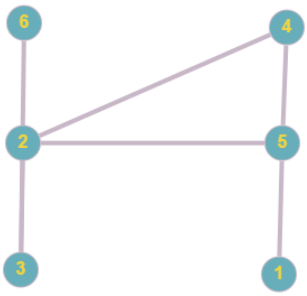
Here the best improvement and swap of 1 and 6 so we keep this exchange.



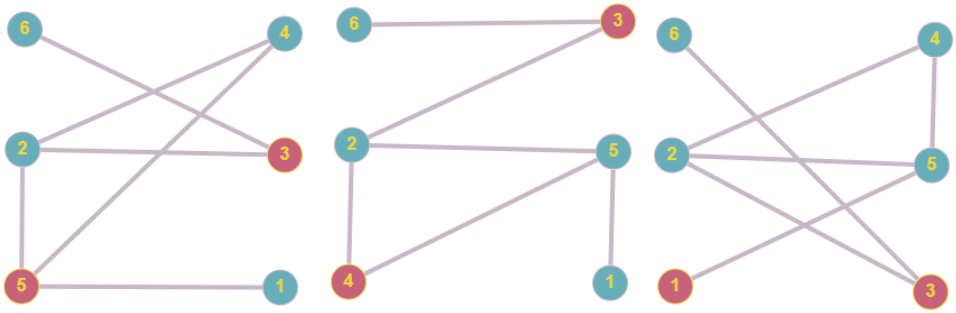
Now we do the same with the second vertex of the group 1.



We see no improvement so we change nothing.



Now we do the same with the last vertex of the group 1.



7Viewing the LocalSearch method

All the solutions are bader and we arrive at the end of our group 1. We see that the solution is improved when we have crossed the first set so we must start again until, during a full course of the group 1, the solution does not improve.

Pseudo code :

Algorithm 1 Local Heuristic

```
1:  function LocalHeuristic(Graph, InitialSolution)
2:      result0 ← InitialSolution[0]
3:      result1 ← InitialSolution[1]
4:      var, var2, i, j ← 0, 0, 0, 0
5:      best ← −1
6:      NM1 ← NumberOfEdgesBetweenTwoSets
7:      lire ← False
8:      while ¬lire do
9:          for each k in result0 do
10:             for each w in result1 do
11:     1           var ← result0[k]
12:     1           result0[k] ← result1[w]
13:     1           result1[w] ← var
14:                 if NumberOfEdgesBetweenTwoSets < NM1 then
15:     2               best ← NumberOfEdgesBetweenTwoSets
16:     2               i ← k
17:     2               j ← w
18:                 end if
19:     3           var ← result0[k]
20:     3           result0[k] ← result1[w]
21:     3           result1[w] ← var
22:             end for
23:             if best ≠ −1 then
24:     4           var2 ← result0[i]
25:     4           result0[i] ← result1[j]
26:     4           result1[j] ← var2
27:             end if
28:         end for
29:         if NM1 > best then
30:             lire ← True
31:         end if
32:     end while
33:     return {result0, result1}
34: end function
```

8 Pseudo-code of Local search heuristic

Explanation:

1) These three lines of code swap the values of result0[k] and result1[w], allowing to test a new configuration in the space of the neighboring solutions.

This is part of the local search process, where different configurations are explored to assess whether they lead to an improvement of the current solution.

2) This part of the code determines whether the current configuration, obtained by exchanging the elements at the k index of result0 with the element at the w index of result1, offers an improvement over the current configuration. If so, the number of the best configuration found so far is updated, and the indices of the exchange are retained in *i* and *j*.

3) These three lines of code are just used to rewrite the swap elements above after testing if the set configurations were better.

4) This section of the code puts the best configuration improvement found, and it terminates the local search process if no improvement is detected.

Complexity:

The temporal complexity of the pseudo-code can be explained as follows:

- The While loop has a variable complexity depending on the proximity of the local minimum so it cannot be deduced. In the best case we find ourselves directly on the global minimum and 1 single iteration is enough unlike the worst case which is that the only maximum is the global minimum so the local minimum and the last iteration due to the facts that we start at the top of the curve of the solutions. On average we will therefore take *k* the number of iterations.

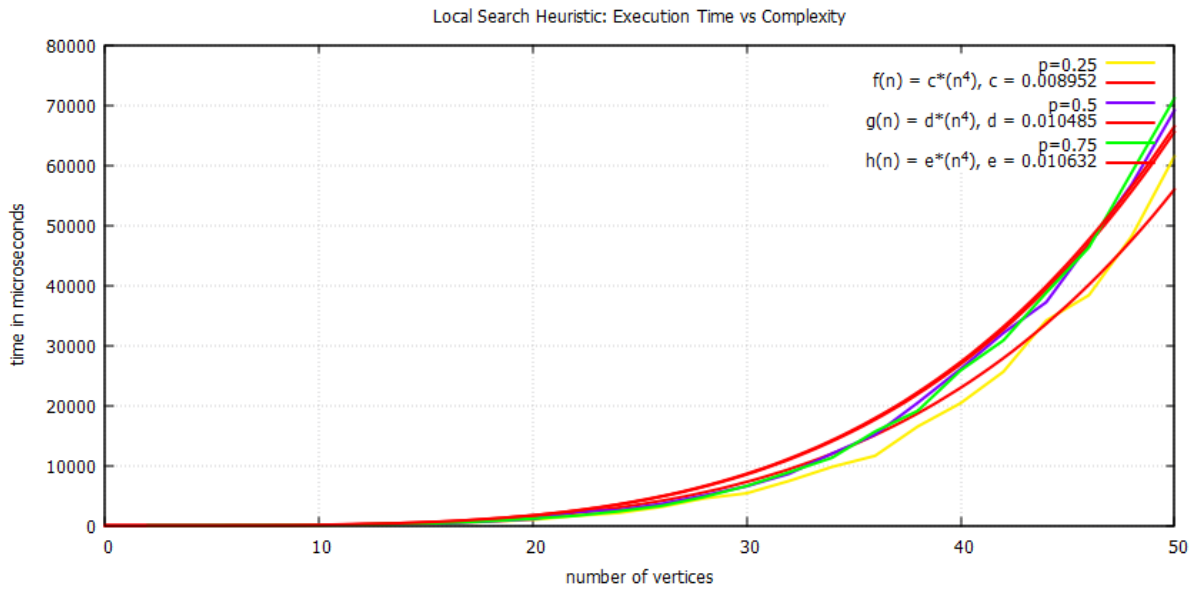
- The two nested for loops have a complexity of $O\left(\left(\frac{n}{2}\right)^2\right)$, where n is the number of vertices. So $O(n^2)$.

- The complexity of the NumberOfEdgesLinkingTwoGroups function is also $O(n^2)$. because it has 2 for nested loop.

This makes us on global $O(n^2) \times O(n^2) \times O(k) = O(k \times n^4)$.

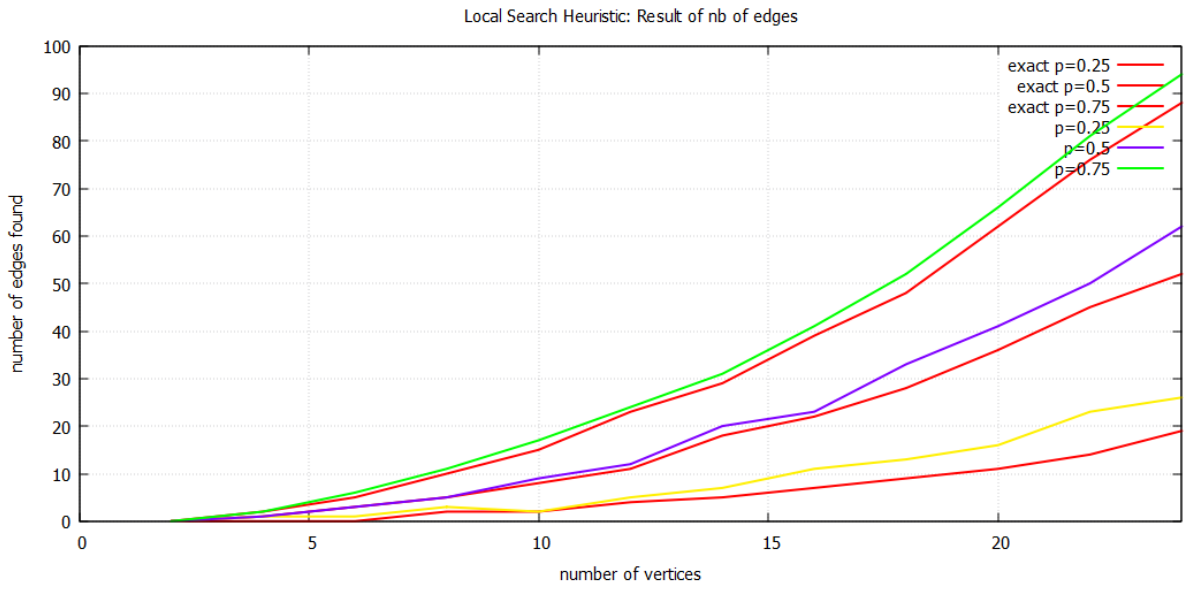
We notice that k is negligible and that our curves resemble an $O(n^4)$.

Analyze of performance of the algorithm :



9 Curves of local search heuristic - Execution time vs complexity

This graphic shows how the execution time of the local search algorithm changes according to the number of vertices for a density probability of 25%, 50%, and 75%. We notice that the algorithm corresponds to a complexity of $O(n^4)$.



10 Curves of local search heuristic - Result of number of edges

We can observe that although local search heuristic is an improved algorithm based on constructive, as said earlier, it finds a local minimum. There are thus possibilities where the local minimum is not the global minimum, characterized on the graph by the fact that the curves of the local search algorithm do not overlap with those of the exact.

Meta-Heuristic Algorithm

Operating of the algorithm:

Following the localSearch algorithm, we undertook to partially resume this same algorithm by adding randomization. Thus, the meta-heuristic algorithm implemented in our project relies on the use of a local search with, in addition, a tabu list. The main objective of this meta-heuristic is to explore the space of solutions more efficiently to find better quality solutions. It combines local research and global exploration to balance the exploitation of current solutions and the exploration of new research spaces.

First, the process begins by randomly generating an initial solution. This solution is then used as a starting point for the local search algorithm, which consists in exploring the neighbors of a given solution to look for local improvements. However, to avoid getting stuck in local optimum, we use a tabu list that stores recently visited solutions. This allows us to explore new regions of the solution space. Then using an acceptance criterion, when searching locally, a neighboring solution is accepted if it improves the cost of the current solution.

However, even if a solution does not improve cost, it can be accepted according to defined criteria (to avoid local optima). Finally, metaheuristics seeks a balance between diversification (exploration of new regions) and intensification (exploitation of current solutions). The tabu list promotes diversification by avoiding revisiting recently explored, while local research intensifies exploration around promising solutions.

The meta-heuristic algorithm follows a main loop that repeats until one of the stop criteria is reached (either a maximum number of iterations or a maximum number of failures). At each iteration of the loop, a new solution is randomly generated, then a local search phase is launched. During local search, the common solution is improved by exploring potential neighbors. Neighboring solutions are evaluated in terms of cost, and the best of them is chosen as a common solution. This procedure is repeated with different initial solutions to explore a wider range of solutions. The tabu list is used to avoid getting stuck in local optima. It maintains a history of visited solutions and imposes temporary prohibitions on returning to these solutions. This encourages the algorithm to explore new regions of the solution space.

Let's take a simple example, consider a simple graph with 6 vertices, and initialize our metaheuristic with an initial random solution. The vertices are numbered from 0 to 5, and we want to divide them into two groups to minimize the number of edges between the groups.

The meta-heuristic algorithm begins by generating a random initial solution. Suppose the first solution generated is:

- Group 1: {0, 1, 2}

- Group 2: {3, 4, 5}

The metaheuristic then undertakes local research to improve the quality of this solution. It explores potential neighbors by swapping vertices between groups while maintaining a tabu list to avoid revisiting the solutions explored. Thus, it could follow the followings iterations:

- Iteration 1:
  Common Solution: Group 1 = {0, 1, 3}, Group 2 = {2, 4, 5}
  Metaheuristics explores different permutations of vertices to improve the solution.
- Iteration 2:
  Common Solution: Group 1 = {0, 1, 5}, Group 2 = {2, 4, 3}
  Similarly, with a new vertex permutation is evaluated.
- Iteration 3:
  Common Solution: Group 1 = {4, 1, 5}, Group 2 = {2, 0, 3}
  Exploration continues with different permutations.
- ... and so on until the acceptance criteria are violated (iter_max or nb_fail_max)

During this local search process, the metaheuristic uses a taboo list of peer groups already visited to avoid revisiting solutions already explored. For example, if the permutation {0, 1, 3} is generated again and is already present in the tabu list, it will not be re-evaluated immediately. Finally, as mentioned, the main loop continues until a stop criterion is reached, such as a maximum number of iterations (iter_max) or a maximum number of failures (nb_fail_max). These constants will always be predefined outside the algorithm.

Pseudo-code :

Algorithm 1 Generate Random Solution

```
1:  function GENERATERANDOMSOLUTION
2:      randomOrdering ← array of size n
3:      for i ← 0 to n − 1 do
4:          randomOrdering[i] ← i
5:      end for
6:      randomOrdering // Random mixing of vector elements
7:      group1, group2 ← empty vectors
8:      for i ← 0 to n − 1 do
9:          if i < n/2 then
10:             PUSH_BACK(group1, randomOrdering[i])
11:         else
12:             PUSH_BACK(group2, randomOrdering[i])
13:         end if
14:     end for
15:     return {group1, group2}
16: end function
```

11 Pseudo-code of GenerateRandomSolution (Call in metaheuristic algorithm)

The generateRandomSolution() function operates in several steps to generate a random initial solution. This function will be called by each iteration of the meta-heuristic algorithm to retrieve a new pair of vertices. First of all, it initializes a vector called randomOrdering of size n, assigning each element of this vector a value corresponding to its index, ranging from 0 to n-1. Then, it uses the Fisher-Yates mixing algorithm via the shuffle function to randomly swap randomOrdering elements, creating a random order of the indices. Thereafter, the function creates two vectors, group1 and group2, which will be used to store the indices of the elements distributed in two distinct groups. A for loop is used to browse the randomOrdering elements, adding each index to group1 if the index is less than half the size of randomOrdering, otherwise the index is added to group2. Ultimately, the function returns a vector containing the two generated groups, group1 and group2, thus forming a random initial solution for the problem under consideration.

This approach of random generation of initial solutions is often used as part of metaheuristic algorithms to diversify the exploration of the space of solutions, thus contributing to improving the search for quality solutions in optimization contexts.

Algorithm 2 Metaheuristic

```
1: function METAHEURISTIC(iter_max, nb_fail_max)
2:     constructive ← CONSTRUCTIVEHEURISTIC()
3:     bestSolution ← LOCALHEURISTIC(constructive)
4:     bestCost ← GETNUMBEROFEDGESLINKINGTWOGROUPS(bestSolution[0], bestSolution[1])
5:     tabuList ← empty list
6:     iter ← 0
7:     nb_fail ← 0
8:     while iter < iter_max and nb_fail < nb_fail_max do
9:         randomSolution ← GENERATERANDOMSOLUTION()
10:        currentCost ← GETNUMBEROFEDGESLINKINGTWOGROUPS(randomSolution[0], randomSolution[1])
11:        SORT(randomSolution[0].begin(), randomSolution[0].end())
12:        isTabu ← false
13:        for each solutionRange in tabuList do
14:            if solutionRange.first ≥ randomSolution[0] and solutionRange.second ≤ randomSolution[0] then
15:                isTabu ← true
16:                break
17:            end if
18:        end for
19:        if not isTabu then
20:            optimised_random_solution ← LOCALHEURISTIC(randomSolution)
21:            SORT(optimised_random_solution[0].begin(), optimised_random_solution[0].end())
22:            if randomSolution[0] < optimised_random_solution[0] then
23:                PUSH_BACK(tabuList, {randomSolution[0], optimised_random_solution[0]})
24:            else
25:                PUSH_BACK(tabuList, {optimised_random_solution[0], randomSolution[0]})
26:            end if
27:            randomSolution ← optimised_random_solution
28:            currentCost ← GETNUMBEROFEDGESLINKINGTWOGROUPS(randomSolution[0], randomSolution[1])
29:        end if
30:        if currentCost < bestCost then
31:            bestSolution ← randomSolution
32:            bestCost ← currentCost
33:            nb_fail ← 0
34:        else
35:            nb_fail ← nb_fail + 1
36:        end if
37:        iter ← iter + 1
38:    end while
39:    return bestSolution
40: end function
```

12 Pseudo-code of metaheuristic algorithm

The algorithm is expressed using the metaheuristic function (iter_max, nb_fail_max) which implements an iterative metaheuristic that seeks to improve an initial solution using a local search approach, using the localSearchheuristic algorithm. To this, we add a taboo list (which is a tab) to diversify the exploration of the space of the solutions so as not to repeat twice in the same place.

Initially, the function obtains a constructive initial solution using the constructive function Heuristic(), then applies a local search to this solution via localHeuristic(), storing the result in the bestSolution variable and calculating its initial cost within the bestCost variable. A tabuList list is initialized to record the solutions already verified and therefore it will be considered prohibited. The function then enters a main while loop, running different iterations until the maximum number of iterations (iter_max) is reached or the maximum number of failures (nb_fail_max) is exceeded.

At each iteration in while loop, it generates a new randomSolution random solution using the generateRandomSolution() function and evaluates its cost with getNumberOfEdgesLinkingTwoGroups(). The taboo list is then consulted to determine whether the solution is prohibited or not. If it is, the main loop is quited and a new random solution is generated. Otherwise, a local search is applied to this solution with the localSearchHeuristic algorithm, and the optimized solution is stored in optimized_random_solution. Once this is done, this solution is added to the taboo list after being sorted for simplified comparison within the taboo list. The current solution is updated accordingly.

The cost of the current solution is compared to the best current cost (bestCost). If the current solution is better, it replaces the previous best solution and the failure counter (nb_fail) is reset. Otherwise, the failure counter is incremented. At each iteration, the iteration counter (iter) is also incremented. Once the loop is complete, the function returns the best solution found and repeats this procedure until exiting the while main loop.

Complexity:

We will now try to calculate the temporal complexity of our algorithm. The time complexity of meta-heuristic algorithm largely depends on the specific nature of the algorithm and the operations it performs. However, one can examine the general parts of the metaheuristic to get a general idea of temporal complexity. Thus, relying on the pseudo-code given in the part above.

Within the generateRandomSolution() function which aims to create a random solution, we have:

- A first for loop in order to choose a random solution, which has a linear complexity in relation to the number of vertices, so $O(n)$,

- A second for loop that cuts into 2 groups, a complexity of $O(n)$, because of the for loop that iterates on each vertex.

Either a total complexity of generateRandomSolution() function is : $0(n + n) = O(2n) = O(n)$

Finally, in the metaheuristic function, we have the following processes with their complexity:

- The main metaheuristic loop runs within a while until a certain number of iterations (iter_max) are reached or a certain number of failures (nb_fail_max) are observed. The two complexities in the worst case are O(n) and O(1) respectively. The complexity depends on these parameters, but it can be approximated to $O(iter\_max)$ in the worst case, with $O(iter\_max) \leq O(n)$. iter_max is defined in the input pararmeters of the algorithm.

We then find the following operations which are not nested:

- The invocation of the generateRandomSolution() function, the operation and complexity of which are described above, so a temporal complexity of $O(n)$.

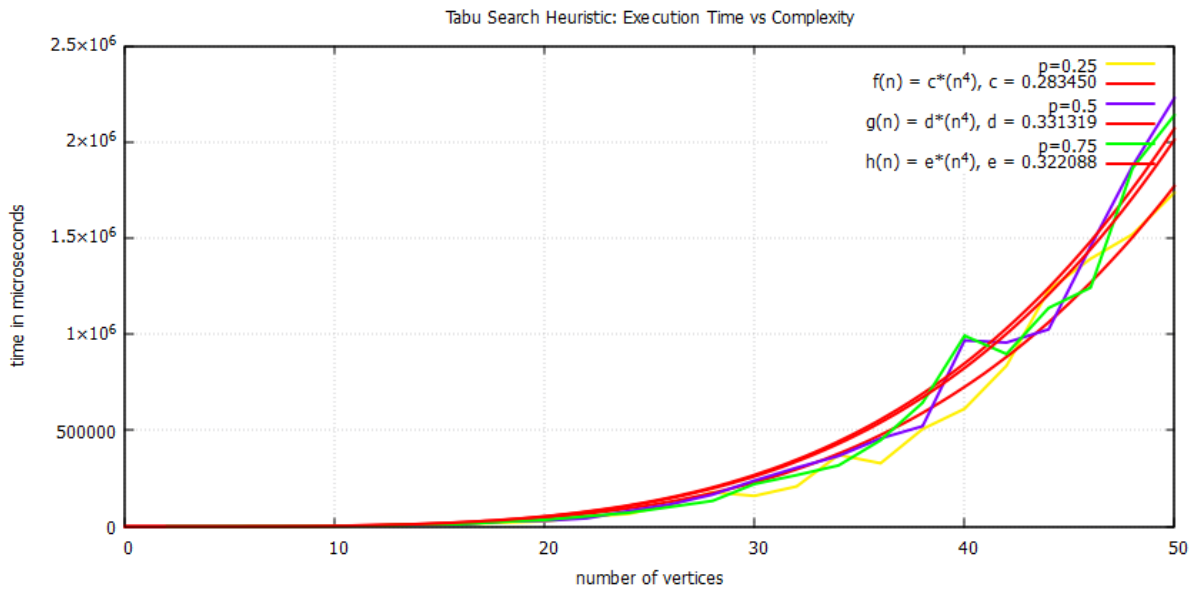
- A first for loop that checks if the pair of randomly selected groups is in the Tabu list, this for loop has a temporal complexity of $O(n)$.

- The localSearchHeuristic call to find a local maximum. However, we have already calculated the temporal complexity of this algorithm in the previous part dedicated to it. We found a temporal complexity for the localSearchheuristic of $O(n^4)$, because we use only constants and not variables.

- Un appel de la fonction sort qui a pour but de trier les groupes par ordre croissant, cette fonction possède une complexité temporelle de $O(n \times log(n))$.

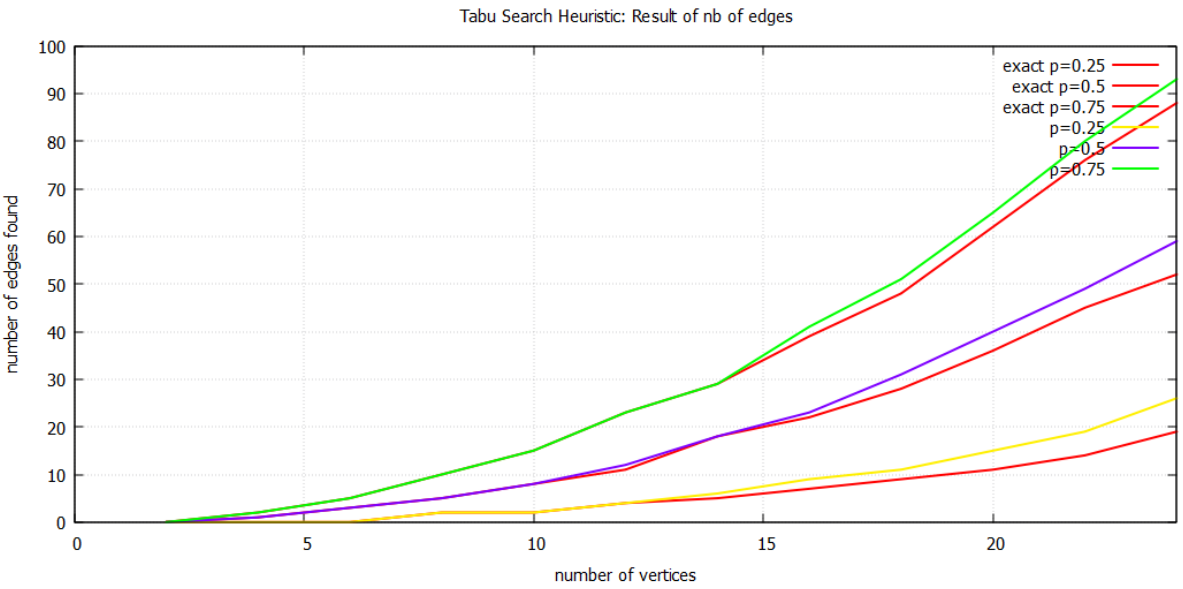
By combining all the complexities, we thus have a total temporal complexity for the metaheuristic algorithm of $O(iter\_max \times (n + n + n^4 + nlog(n))) = O(iter\_max \times n^4)$.

Analyse of performance of the algorithm :



13Curves of tabu search heuristic - Execution time vs Complexity

This graphic represents the evolution of the execution time of the algorithm tabu search heuristic as a function of the number of vertices n. We note that despite the presence of iter_max, the curves follow those representing the complexity $O(n^4)$.



14 Curves of tabu search heuristic - Result of number of edges

This graphic represents the minimum number of edges found based on the vertex number and probability of having a stop for the tabu search algorithm and the exact one. We notice that even if the tabu search allows to go further than the local search using random, it does not always find the best solution. But the curves remain relatively close for a complexity of $O(n^4)$ (against $O(n^2 \times 2^n)$ for the exact).

Choice of parameters & use case of the algorithm:

The meta-heuristic algorithm relies on a judicious selection of parameters to ensure an optimal balance between the exploration of the solution space and the exploitation of local solutions. Their main objective is to save memory space but above all to optimize time. The two main parameters to consider are iter_max and nb_fail_max as well as the max size of the tabu list but the size of tabu list can't be greater than the number of maximum iteration and we estimated it was an acceptable maximum size for the tabu list .

The iter_max parameter is crucial, determining the maximum number of iterations that the metaheuristic will perform. Its value, set at 100 in our implementation, imposes a limit to the search time dedicated to the search for the optimal solution. In other words, the algorithm stops after performing the most iter_max iterations, seeking to converge to the best possible solution in this time frame.

Similarly, the nb_fail_max parameter plays a determining role in controlling the maximum number of consecutive failures tolerated before the algorithm is stopped. A failure is defined as the inability to generate a new solution that improves the current cost. Fixed at 20 in our implementation, nb_fail_max acts as a critical threshold, signaling a potential stagnation of the algorithm when this number of failures is reached. We thus assume that if the limit nb_fail_max is reached means that it is unlikely to find a better solution (unless you want to test all the possible cases but would increase the execution time and so time complexity.

These parameters choices (iter_max = 100 and nb_fail_max = 20) emerged from an empirical exploration, aiming to find an optimal compromise between the exploration of neighboring solutions and the exploitation of local solutions, while having a final complexity not too high. Indeed, the higher one of the parameters, the higher the temporal complexity.

Moreover, in practice, this algorithm can be adapted to specific instances of the problem by adjusting these parameters according to the characteristics of the input graph. For example, larger iter_max values may be preferred for more complex instances that require further exploration, while adjusting nb_fail_max controls the scan level as required.

Thus, our meta-heuristic stands out for its flexibility, allowing users to customize the behavior of the algorithm according to the specific requirements of their problem instances. This deliberate choice of parameters reflects the desire to offer an adaptable and efficient solution in various optimization contexts.

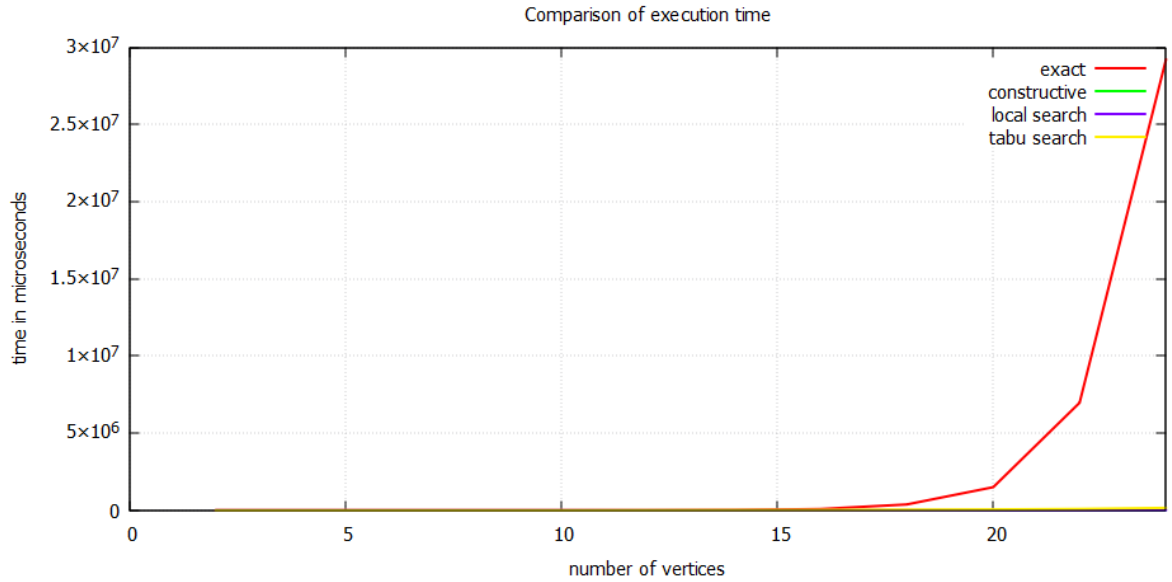
In the end this algorithm can prove to be interesting and efficient for graphs with a number n of very large vertices. And conversely, it may seem an efficient month when the number of vertices becomes small.

Moreover, the worst case would be to start with the smallest case and with each new iteration or some iterations, that we find a better case. this would therefore never cause us to exit the main loop while since the criterion nb_fail_max would never be exceeded. We would therefore be limited to having to achieve the maximum number of iterations that was given as input parameter. Therefore, the best case would be to directly find the best solution and thus exit the main loop thanks to the criterion nb_fail_max. Hence the importance of choosing values to these criteria that are judiciously chosen.

Comparison of the algorithms:

So now we will be able to compare the execution time and the results obtained between our different algorithms using the data that we obtain.

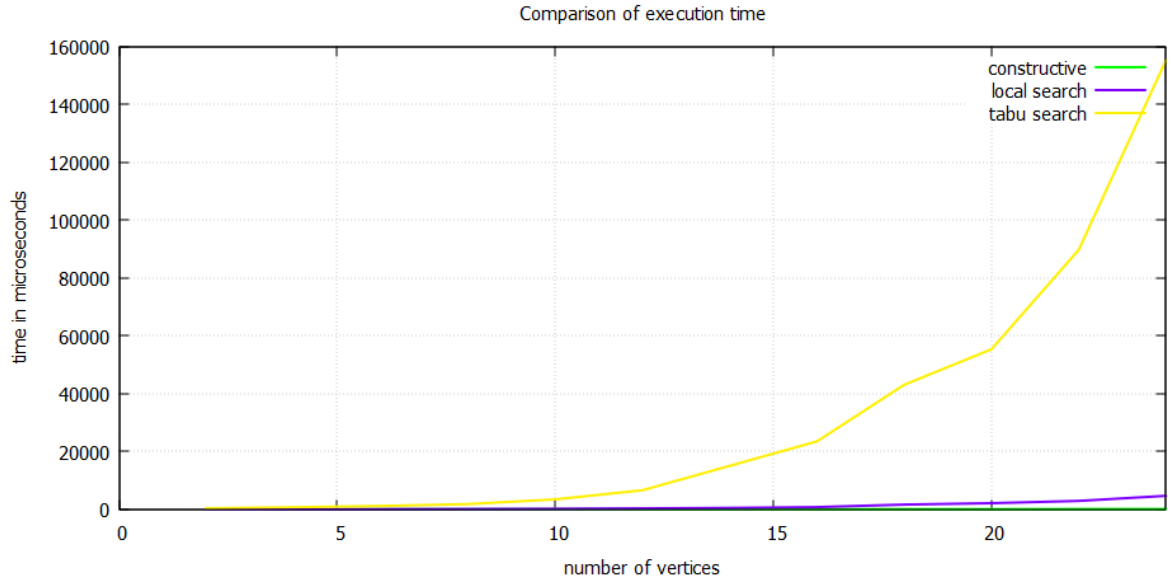
First, we can see the execution time in microseconds based on the number of vertices for the four algorithms :



15 Comparison of execution time - 4 algorithms

We can very quickly notice that the execution time of the exact algorithm is much longer than its competitors, see incomparable. This is because despite our improvements, because of the difficulty of the problem, our algorithm that always finds the exact solution has a bad complexity.

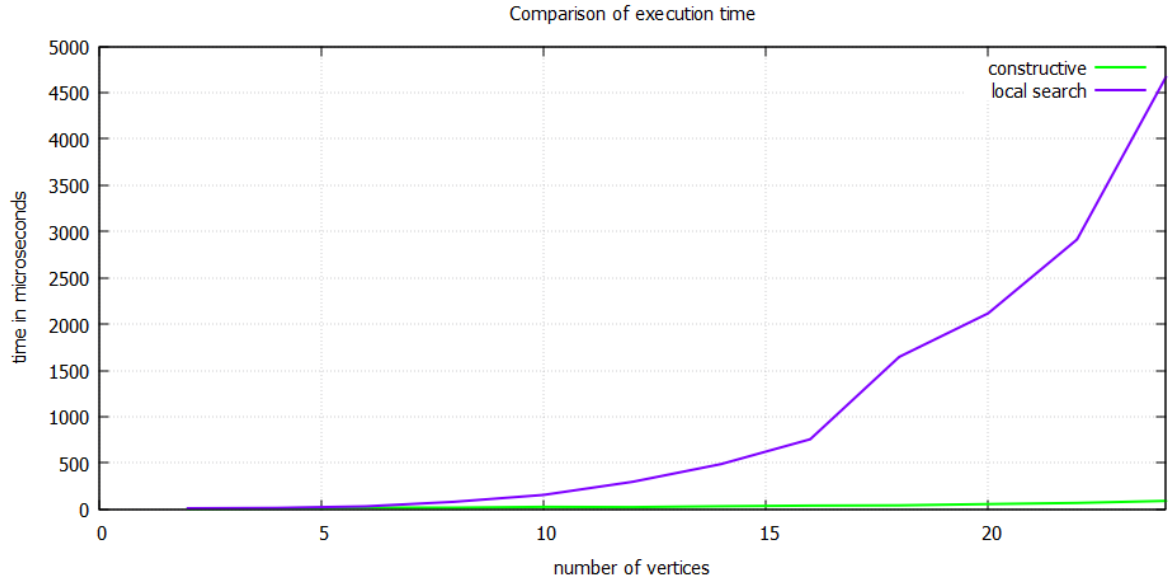
So, we will compare the execution time of the three other algorithms :



16 Comparison of execution time - Constructive, Local search, Tabu search

We notice that as seen above, the tabu search heuristic algorithm is also much slower than the other two. This is due to the fact that the complexity of the tabu search is $O(iter\_max \times n^4)$ while the local search algorithm has a complexity of $O(n^4)$. Here *iter_max*(=100) is superior to n, so this is why tabu search is slowest than local search. If we have $n >> iter\_max$, *iter_max* is derisory in front of n, so the tabu search algorithm and the local search algo will approximately have the same execution time.

Now let's compare the local and constructive heuristic algorithms :



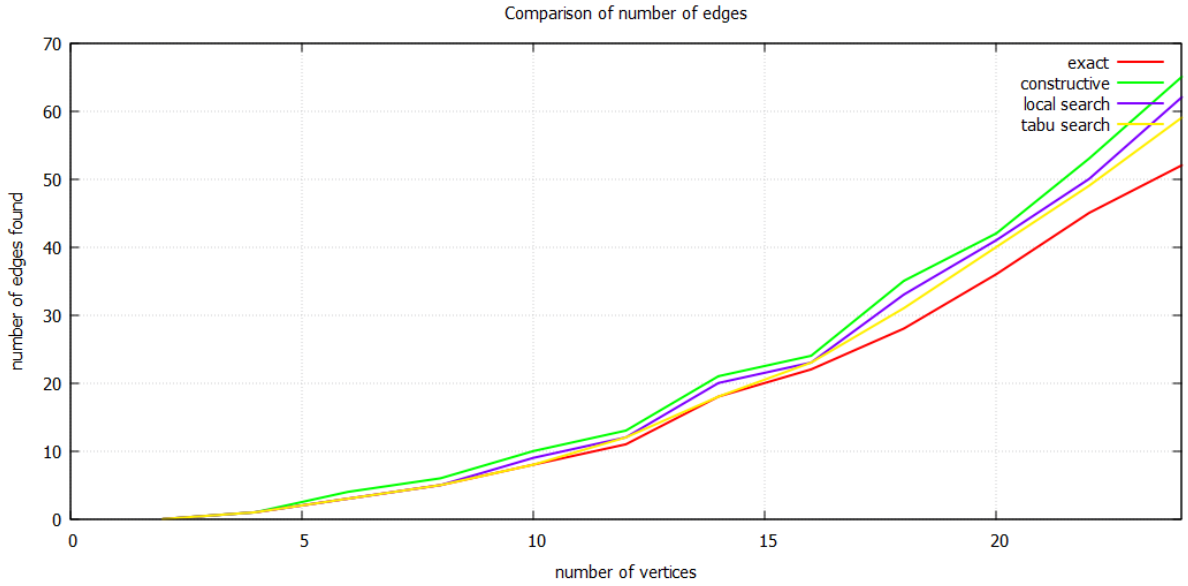
17 Comparison of execution time - Constructive, local search

We can notice that the constructive heuristic algo is slower than the local heuristic. This is due to the complexity where the constructive is $O(n\text{^}2)$ and the local search is $O(n^4)$.

So, the more n increases, the more time difference between the four algorithms increases. For example, for n=6, the exact algo takes 0.04ms, the constructive 0.018ms, the local search 0.031ms and the tabu search 1.152ms. Note that the search tabu is initially slower than the exact but the more n increases, the more important in front of *iter_max*. But for n=20, the execution time of the exact algo is 1.495s while that of the constructive heuristic is 0.056ms, that of the local search is 2.117ms, and that of the tabu search is 55.296ms.

So, we can say that the exact algo is the slowest followed by tabu search algo, local search algo, and constructive algo.

Now that we have been able to compare the different algo according to the execution time and highlight the best ones at the time level, we will compare them the number of edges found between two subgroups of vertices.



18 Comparison of number of edges - 4 algorithms

We can notice that the most efficient algorithm is the exact one that finds at least a lower number of edges than other algorithms. We can notice that the most efficient algorithm is the exact one that finds at least a lower number of edges than other algorithms. Similarly, the tabu search result is at least lower than that of the local search which is at least lower than that of the constructive. This is because the exact always finds the best possible result so no other result for a given n can be inferior to it. And for the others, the tabu search algo. is an improvement of the local search algo. which is itself an improvement of the constructive algo. We can finally say that although the exact is the slowest, it always finds the best possible result while the constructive heuristic is the fastest but with the worst result of the four.

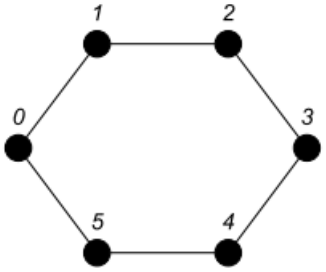
Executions of the algorithms:

We will now see the execution of the different algorithms and be able to compare them by seeing particular graphs. We will see first the graphs test1 and test2 that were given to us :

Test 1 :

“test1.in”:

6 6
0 1 0 5 1 2 2 3 3 4 4 5



“test1_exact.out” :

6 2
0 1 2
3 4 5

“test1_constructive.out” :

6 2
5 4 3
1 0 2

“test1_local_search.out” :

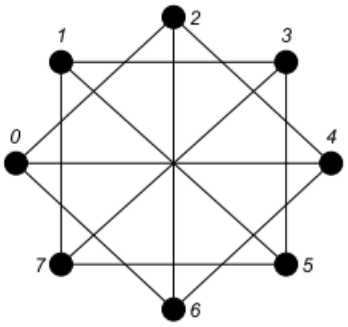
6 2
5 4 3
1 0 2

“test1_tabu_search.out” :

6 2
5 4 3
1 0 2

We can see that in this example, all algorithms have the same minimum number of edges between the two groups of vertices. Indeed, despite the results of the two different groups, this is due to the peculiarity of the graph where all the vertices have a degree 2.

Test 2 :



“test2.in”:

8 12
0 2 0 4 0 6 2 4 2 6 4 6 1 3 1 5 1 7 3 5 3 7 5 7

“test2_exact.out” :

8 0
0 2 4 6
1 3 5 7

“test2_ constructive.out” :

8 0
7 1 5 3
0 6 2 4

“test2_local_search.out” :

8 0
7 1 5 3
0 6 2 4

“test2_ tabu_search.out” :

8 0
7 1 5 3
0 6 2 4

We can notice that the graph is not connected. Indeed, it consists of two sub-graphs of size $\frac{n}{2}$. Thus, the exact to find 0 as a solution and the constructive then. The local search and tabu search finding no better one renders the same thing as the constructive heuristic.

Now we will see new tests :

Test 3 :

“test3.in”:

6 5
0 1 1 2 2 3 3 4 4 5

“test3_ exact.out” :

6 1
0 1 2
3 4 5

“test3_ constructive.out” :

6 2
4 3 2
0 5 1

“test3_ local_search.out” :

6 1
4 3 5
0 2 1

“test3_tabu_search.out” :

6 1
4 3 5
0 2 1

This is an example where constructive heuristic is not effective. Indeed, in this graph, the simplest to have the least edges in the two groups is to separate in the middle. But the constructive does not and therefore finds as solution 2 proving as we saw on the comparisons earlier that it is not the most effective.

Conclusion

To conclude, we saw four different algorithms that each try to solve the Minimum Bisection Problem. The fact that each of the algorithms has its advantages and weaknesses without any being able to solve the MBP accurately and quickly shows that this problem is indeed an NP-Hard problem.

So, each algorithm is effective in some cases:

- For the exact algorithm, it is very effective when n is small. Indeed, it always gives the minimum number of possible edges between two subgroups of vertices. And from $n = 22$ or $n = 24$, this algorithm becomes much slower than other algos because it has a complexity of $O(n^2 \times 2^n)$. It is useful when you have a small n or when you absolutely want to have the best possible result.

- For the constructive heuristic algorithm, it is the fastest of the four but also the one with the worst results. Thus, it is useful for large companies, and one can afford to have a margin of error on the results obtained and want to be fast.

- For the local heuristic algorithm, it always finds a better or equal result than the constructive, because it is taking its result as input, but it is a little slower. It allows to find a local minimum starting from the minimum find by the constructive. But like constructive heuristic, it does not always find the best result, the local minimum is not always the global minimum. It is therefore useful when one wishes to have a more precise result than the constructive with a large n and an accepted margin of error on the result.

- The tabu search heuristic algorithm is the one with the best results of the other heuristic algorithm. But it is at the same time the slowest, so it is useful when we wish, for a large number of vertices, to find the best possible result. Indeed, the exact is so slow for a large n, that the tabu search allows to have a result closest to the optimal result but with only a complexity of $O(n^4)$.

Finally, this project allowed us to study an NP-Hard problem where the algorithm that finds the exact solution has a bad complexity. Thus, for different use cases, different algorithms have been created, allowing to have a suboptimal result faster.

During this project, we encountered some difficulties. First, we had to calculate the time complexity of our algorithms, particularly for the exact-heuristic and the localSearch-heuristic, on which the meta-heuristic depends.

In the localSearch, it was difficult to understand the complexity of the while loop, and we hesitated between O(1), O(n) and O(m). Furthermore, we couldn't decide to what extent the algorithm's constructiveHeuristic criterion should make the algorithm efficient in finding the solution. This is why we had to change the criterion several times, and therefore the algorithm. Time management also proved to be a major constraint in this project. We were certainly a little too late in writing the report, which created a heavy workload in the few days before the project was due.