Graph theory: final project

Minimal Bisection Problem -Group 2

KOUAMEN Igore Bolton, NAULET Emma, PAITIER Mathias, SOPRANSI Maëlle

ISEN NANTES - CIR3

Due date : 21/12/2023

Number of words : 4742

Table of content

Introduction

The task of effectively dividing the vertices within a given graph into subsets, ensuring that specific conditions are met, stands as a fundamental challenge in algorithmic studies. Beyond its clear theoretical significance, problems related to graph partitioning hold substantial practical relevance across various domains, including computer vision, image processing and VLSI layout design.

In particular, the issue of partitioning a graph into components of equal size, while minimizing the number of connections between these components, plays a pivotal role in parallel computing. To illustrate, to parallelize applications, it is typically necessary to evenly distribute the computational workload among processors, while simultaneously minimizing the inter-processor communication.

Formally, let G = (V, E) be an undirected simple graph such that |V| = 2n. (Note that G has an even number of vertices, and it is not necessarily connected.) The Minimum Bisection Problem (MBP) consists in finding a partition of V, into two sets V1 and V2 such that V = V1 ∪ V2, V1 ∩ V2 = ∅ and |V1| = |V2| = n, that minimises the number of edges x1 x2 with x1 ∈ V1 and x2 ∈ V2.

The MBP is a well-known combinatorial optimization problem in graph theory and computer science and has a history dating back several decades. It was proved that the MBP is NP-hard, which means that it is computationally challenging to find an optimal solution in a reasonable amount of time for large graphs. This result underscored the problem's difficulty and led to the development of approximation algorithms.

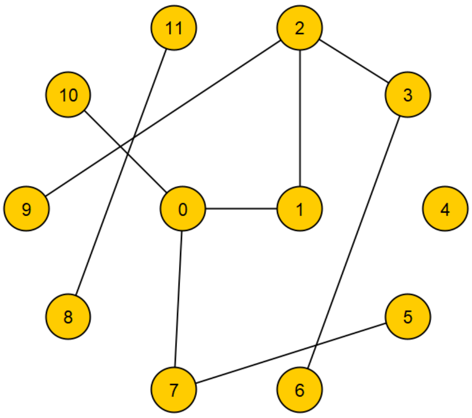
The MBP remains a fundamental problem in graph theory and combinatorial optimization, and its study has not only contributed to theoretical advancements but has also led to practical solutions for optimizing various real-world applications, particularly in the field of computer science and engineering."

There are real-life situations that can be modelled by the MBP, like the transportation planning, in the planning of transportation networks, minimizing the disruption caused by dividing the network into two parts can be critical. This is relevant in scenarios such as urban planning or logistics. The energy distribution is also a situation that can be modelled by the MBP, within the realm of energy distribution, it is crucial to partition a power grid into two segments while minimizing the reduction in power supply. This optimization is essential to maintain a balanced and dependable distribution of electricity.

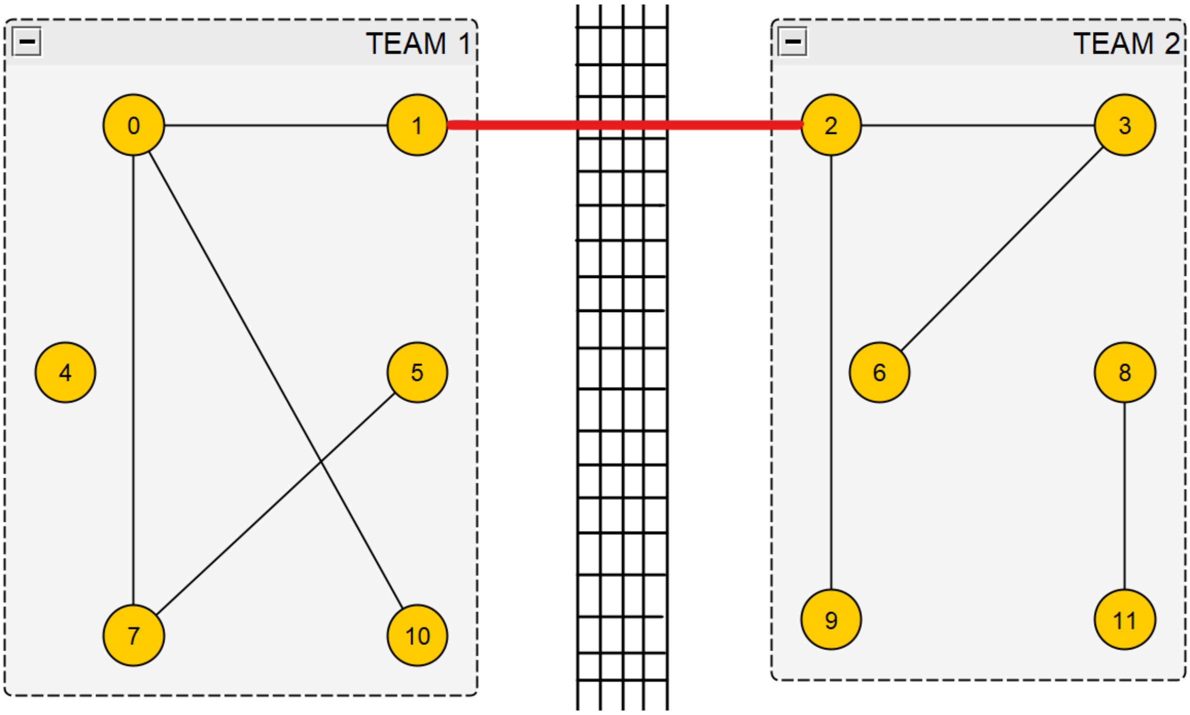
Our original idea for the real-life situations that can be modelled by the MBP is in the sports domain. Let's imagine a team of volleyball players for example. They want to form 2 teams to play a game but some of them have already played against each other and they want to vary the team's composition. We can represent this situation by the MBP problem. The vertices are the players. An edge between 2 vertices means that the 2 players have already played against each other. The goal is to minimize the edges to minimize the number of players who already have played against each other.

Minimal bisection, in this domain, could be used to form balanced and diverse teams while minimizing pre-existing interactions between opponents.

To illustrate, here are 12 players. The edges are set such that 0 has already played against 1, 7 and 10, 8 has already played against 11, 4 has never played against anyone, etc.



We apply the algorithm, and we obtain 2 teams on the field with only 1 edge between them so only 2 players having already played against each other.



For the implementation of our algorithms, we have chosen the C++ language, because it is the language that we have studied the most and that we master the best.

Here are the different tools used to make the project: to code the algorithms we used CLion; to model graphs we used the application yEd; to model graphics we used GnuPlot. For all the graphics we used Maëlle's computer, it has a Ryzen 9 processor and a RAM of 32 Go.

To have a better organization in the project, we divided the different tasks in the group, at first, we recovered a class we had done in TD to create a graph with these basic functions associated. Maelle coded what to make random graphs to perform tests. Emma and Igor started by working on the exact algorithm, while Maelle was working on the constructive heuristic and Mathias on the local search heuristic and then on the Tabu Search meta-heuristic. In addition, Mathias managed the creation of out files in the main, while Emma and Maelle of the in files. Then, Maelle and Mathias have made the implementation to compute the time complexity of each algorithm. Finally, Maelle has made the run experiments to observe and compare the performance of our algorithms with charts. Maelle and Emma wrote the introduction of our report and Mathias wrote the readme document and the conclusion.

Exact algorithm to solve the MBP

Explanation

We implemented a recursive algorithm to find the minimum cut of a graph using an exact algorithm. The `**ExactAlgorithm**` function generates all possible subsets of vertices and computes the cut for each pair of subgraphs.

The `**Exact_Main**` function initializes the process and returns the result, which includes the best cut and corresponding subgraphs.

Therefore, we will describe to you what our exact algorithm does.

It takes a graph represented by an adjacency list (**GraphAdjacencyList G**), a set of vertices (vector<int> vertices), and some parameters to track the minimum cut (**min_cut and best**).

The function generates the power set of vertices using recursion. It starts with an empty subset (**curr**), and for each vertex, it includes or excludes it from the current subset.

When the size of the current subset reaches half of the total number of vertices **(n/2)**, it computes the corresponding complementary subset **(deduire_subgraph2(vertices, curr))** and forms two subgraphs.

It then computes the cut (common edges) between these two subgraphs using the function **calculEdgeCommun**.

If the current cut is smaller than the minimum cut found so far **(min_cut),** it updates the minimum cut and the best subset **(best).**

The recursion continues until all possible subsets are considered.

The **Exact_Main** function initializes the exact algorithm process for finding the minimum cut of a graph. We will then explain in detail how it works.

The function starts by creating a **'vertices'** vector which contains the indices of the vertices of the graph, from 0 to **G.V - 1**.

It initializes the variables **min_cut** (the minimum cut) and **'best'** (the best subset of vertices).

Then, it calls the **ExactAlgorithm** function to perform the exact algorithm on the graph **G**, the vertex **vertices**, the minimum cut **min_cut,** and the best subset best.

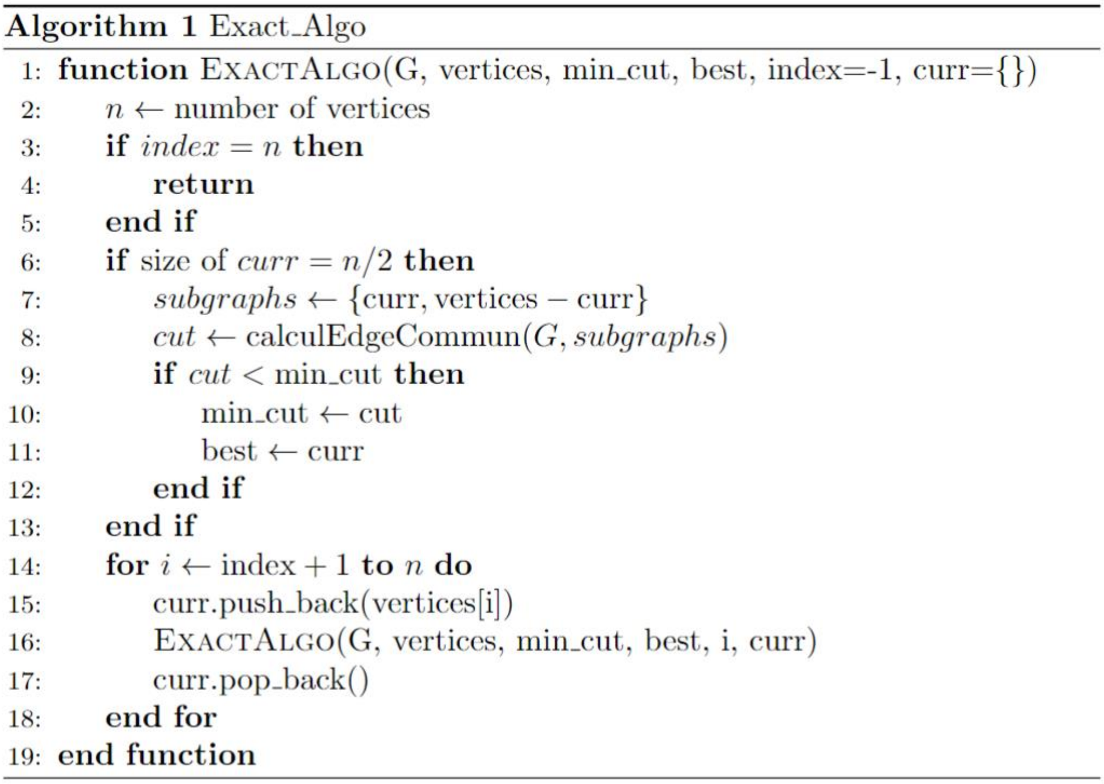
After running the algorithm, it infers the second subgraph from the best subset found by calling the **deduce_subgraph2** function.

Finally, it stores the **best** subset and the second subgraph in results vector **res** and returns this vector.

The **Exact_Main** function coordinates the execution of the exact algorithm and returns the results of the minimum cut and the corresponding subgraphs.

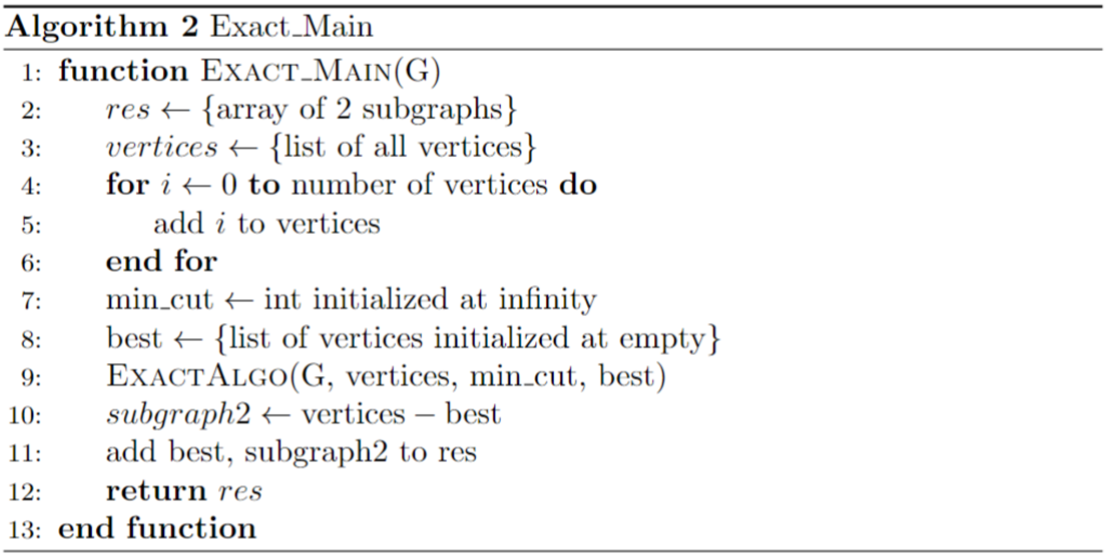
Pseudo-code

Here is the corresponding pseudo-code.



Algorithm 1 Exact_Algo

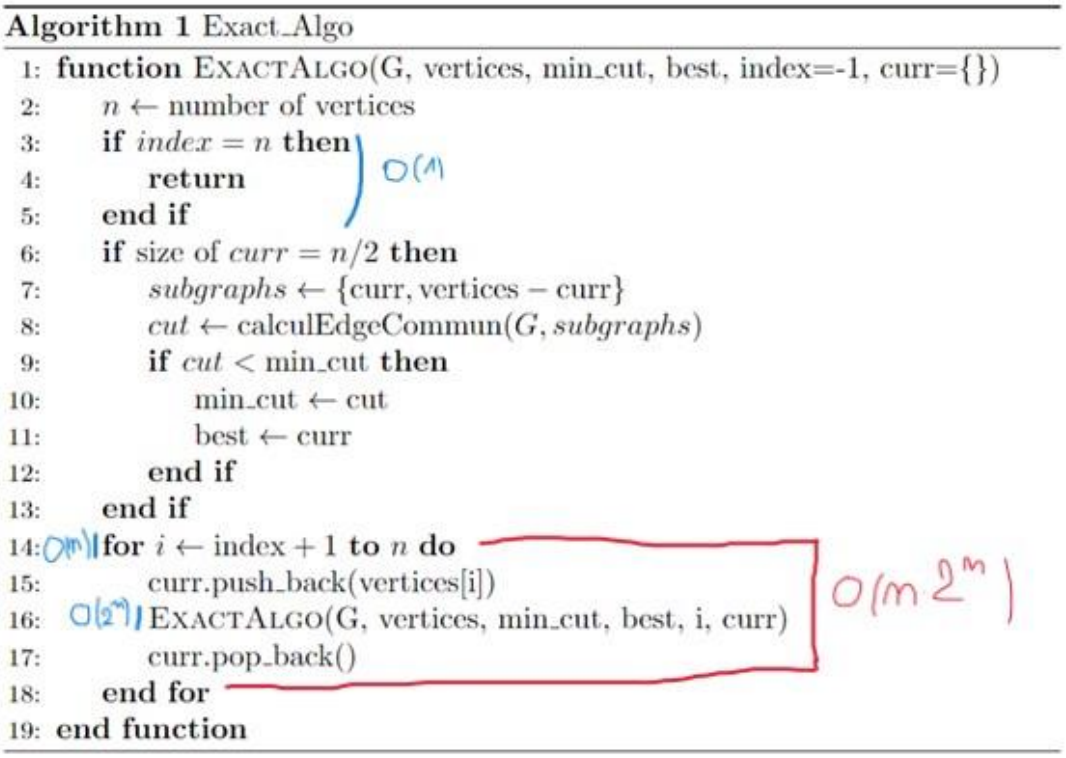
```
function ExactAlgo(G, vertices, min_cut, best, index=-1, curr={})
    $n$ ← number of vertices
    if $index = n$ then
        return
    end if
    if size of $curr = n/2$ then
        $subgraphs$ ← {curr, vertices − curr}
        $cut$ ← calculEdgeCommun($G, subgraphs$)
        if $cut$ < min_cut then
            min_cut ← cut
            best ← curr
        end if
    end if
    for $i$ ← index + 1 to $n$ do
        curr.push_back(vertices[i])
        ExactAlgo(G, vertices, min_cut, best, i, curr)
        curr.pop_back()
    end for
end function
```



Algorithm 2 Exact_Main

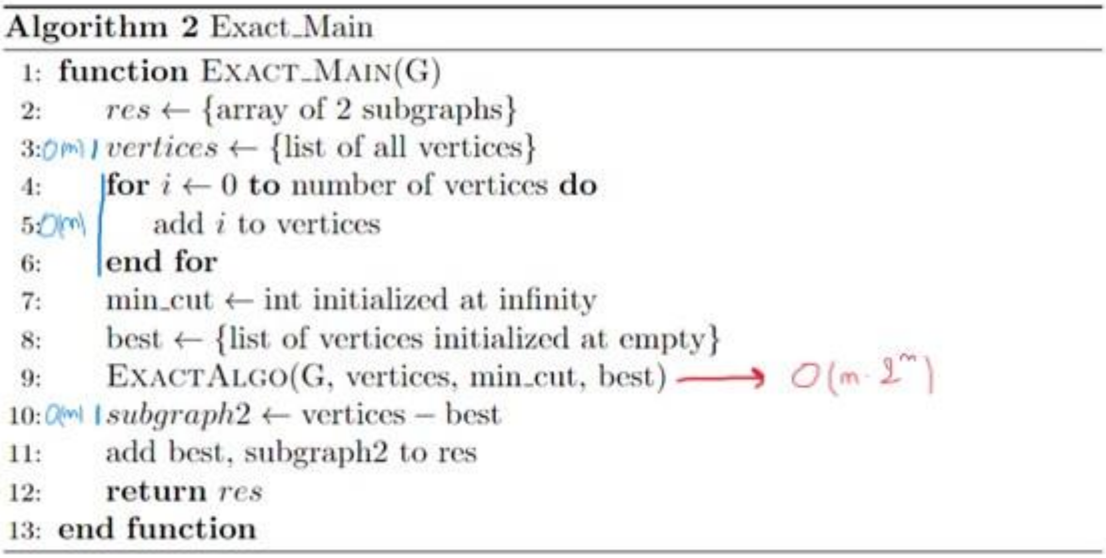
```
function Exact_Main(G)
    $res$ ← {array of 2 subgraphs}
    $vertices$ ← {list of all vertices}
    for $i$ ← 0 to number of vertices do
        add $i$ to vertices
    end for
    min_cut ← int initialized at infinity
    best ← {list of vertices initialized at empty}
    ExactAlgo(G, vertices, min_cut, best)
    $subgraph2$ ← vertices − best
    add best, subgraph2 to res
    return $res$
end function
```

Complexity



Algorithm 1 Exact_Algo

```
function ExactAlgo(G, vertices, min_cut, best, index=-1, curr={})
    $n \leftarrow$ number of vertices
    if $index = n$ then      } O(1
        return
    end if
    if size of $curr = n/2$ then
        $subgraphs \leftarrow$ {curr, vertices − curr}
        $cut \leftarrow$ calculEdgeCommun($G, subgraphs$)
        if $cut <$ min_cut then
            min_cut ← cut
            best ← curr
        end if
    end if
O(m) for $i \leftarrow$ index + 1 to $n$ do                      O(m 2^m)
        curr.push_back(vertices[i])
O(2^m)  ExactAlgo(G, vertices, min_cut, best, i, curr)
        curr.pop_back()
    end for
end function
```



Algorithm 2 Exact_Main

```
function Exact_Main(G)
    $res \leftarrow$ {array of 2 subgraphs}
O(m) $vertices \leftarrow$ {list of all vertices}
    for $i \leftarrow 0$ to number of vertices do
O(m)    add $i$ to vertices
    end for
    min_cut ← int initialized at infinity
    best ← {list of vertices initialized at empty}
    ExactAlgo(G, vertices, min_cut, best) ⟶ O(m·2^m)
O(m) $subgraph2 \leftarrow$ vertices − best
    add best, subgraph2 to res
    return $res$
end function
```

The **Exact_Algo** has two choices (include or exclude) and explores all possible combinations. The total number of subsets is 2^n, where n is the number of vertices. This step contributes O(2^n) to the overall complexity.

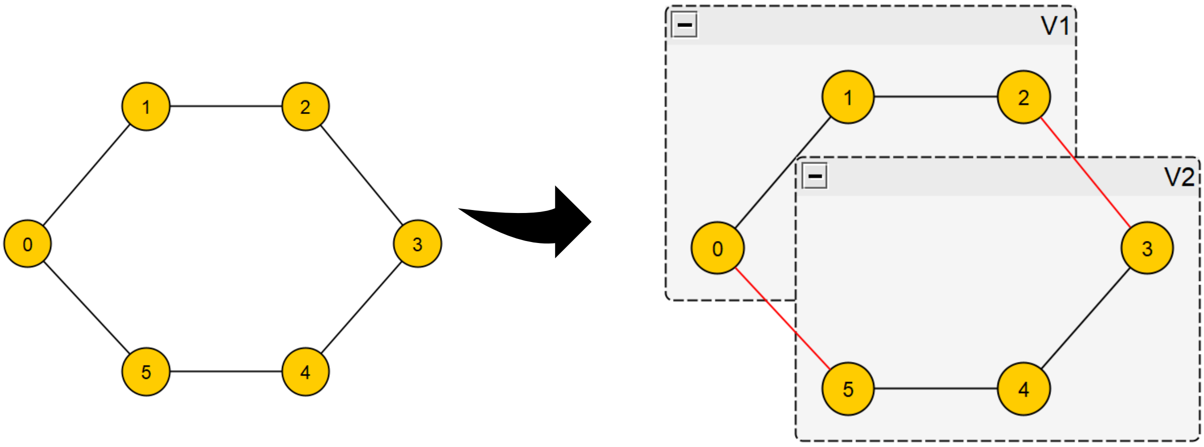
In each recursive call where the size of the current subset is n/2, the algorithm computes the cut between the two subgraphs. The cut computation itself involves checking common edges between two subgraphs, and the complexity depends on the size of the subgraphs. However, since this reduction computation is performed for each subset, it does not modify the overall exponential complexity.

Indeed, it appears that the recursive generation of the power set dominates the complexity with a complexity of O(n x 2^n), so it is making the overall complexity of the algorithm exponential. It should be noted that algorithms involving power set generation tend to have exponential complexity and can become impractical for large input sizes.

In total, the complexity of the exact algorithm is the complexity of the recursive calls which is O(n x 2^n), so an exponential time complexity.

Solution

Let's do an example! If we take this graph with 6 vertices, such that:



We are looking for all the possible sets of size = n/2 = 3.

All the possible sets starting with 0 are: 012, 013, 014, 015, 023, 024, 025, 034, 035, 045.

All the possible sets starting with 1 are: 123, 124, 125, 134, 135, 145.

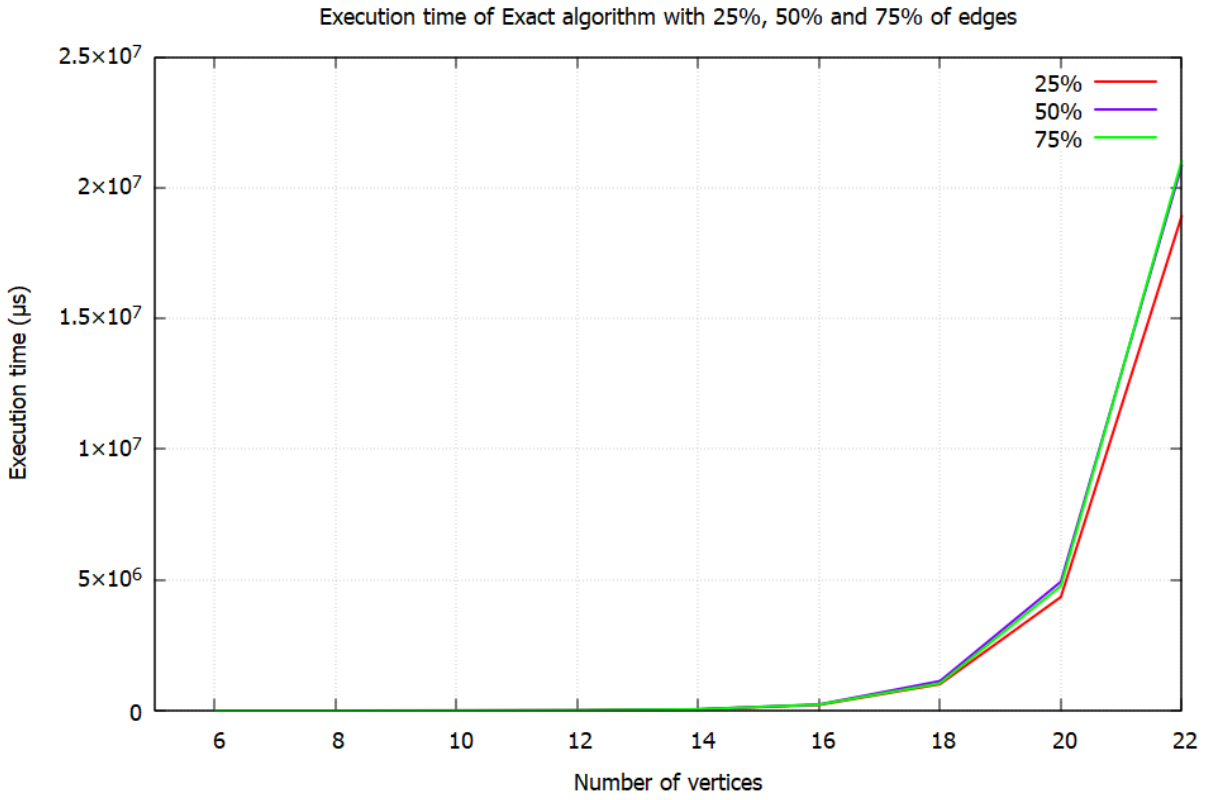
All the possible sets starting with 2 are: 234, 235, 245.

For each partition corresponding, we calculate the number of edges between set1 and set2.

We found that all possibilities give the same answer to the problem: the minimum is 2 edges. So we keep the following set: set1 = 012 and set2 = 345.

Execution time

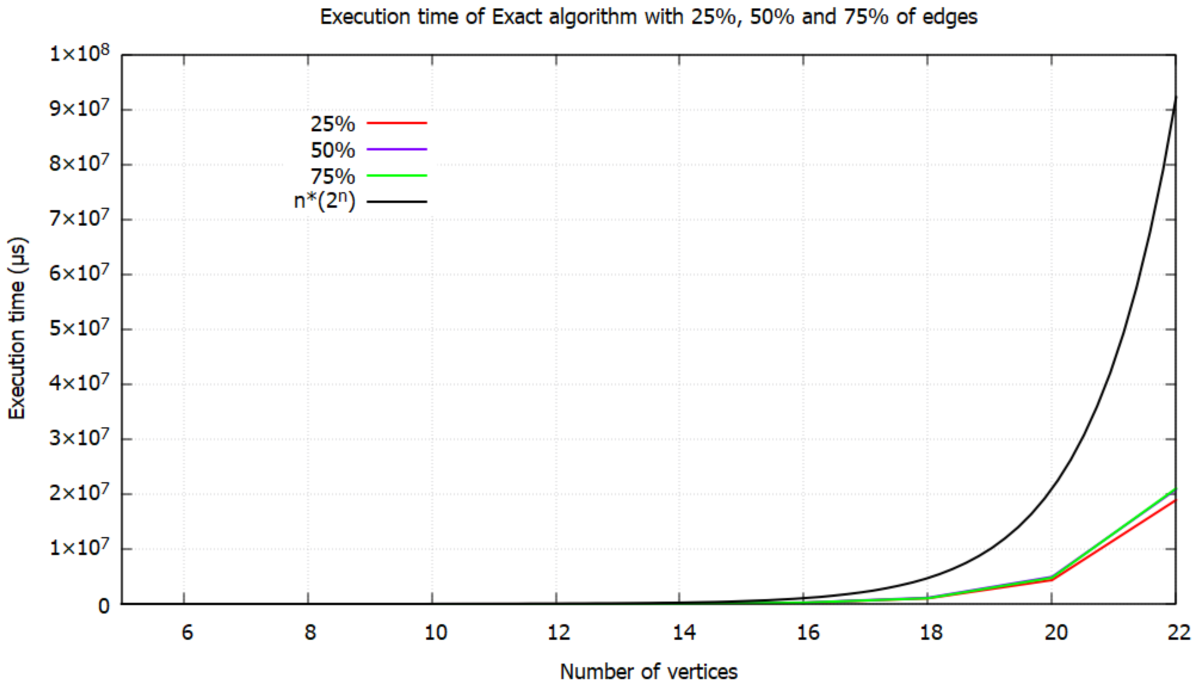
This is the execution time (in microseconds) of the exact algorithm depending on the number of vertices. We have tested it from 6 vertices to 22 vertices for 3 cases of different edge probability: 25%, 50% and 75%.



We can see that the execution time of the algorithm is exponential, mainly due to the exponential nature of the recursive function. The more the graph has vertices, the more it takes time to execute the algorithm and the growth is exponential. As for the number of edges, it doesn’t have an impact at all on the execution of the exact algorithm.

This means that the algorithm's performance can be problematic for large graphs. Indeed, from the step with 24 vertices, the algorithm takes many minutes to find the solution. Algorithmic improvements or heuristic approaches may be required to handle this problem efficiently.

In addition, we can compare the execution times for all percentage of edges to our previously calculated temporal complexity:



As we can see, the actual results are better than the theorical complexity estimated before. We can explain this because we don’t really take every set of vertices in our algorithm.

Constructive heuristic to solve the MBP

Explanation

Our algorithm to solve the Minimum Bisection Problem with a constructive heuristic, works in four steps.

The first step is to take the highest degree vertex in all the vertices of the graph and put it in the first subgraph. If two vertices have the same degree, we take the first found.

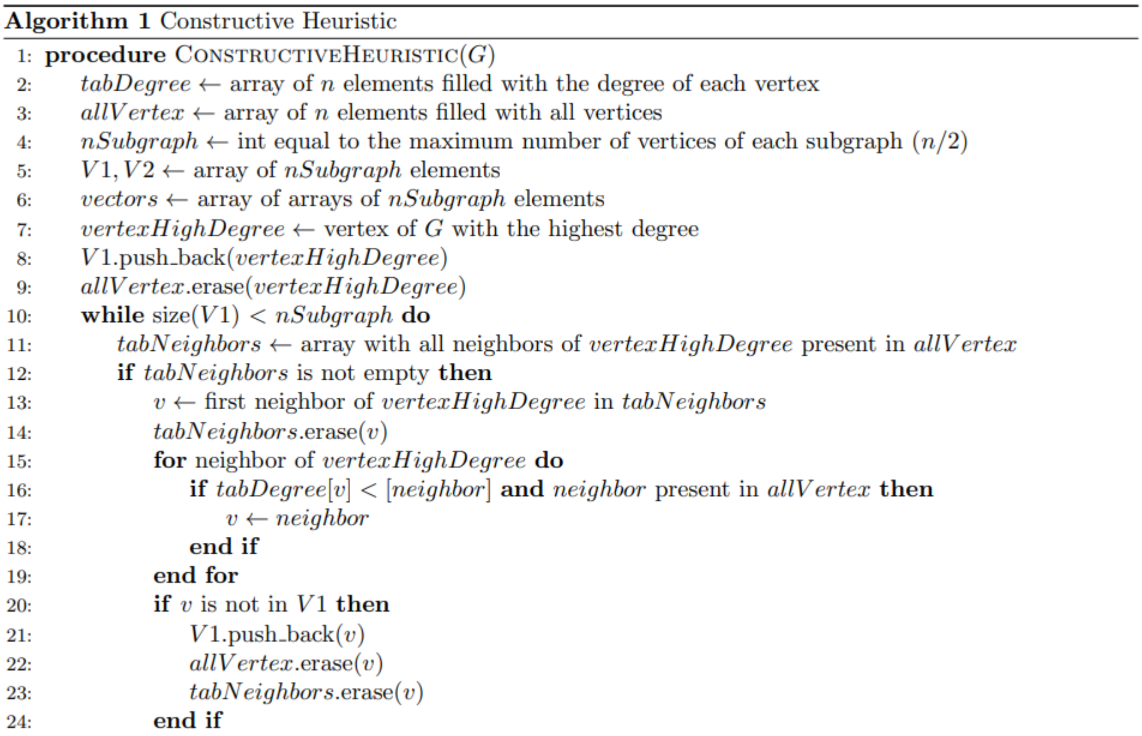
The second step consists in looking at all the neighbours of the vertex of higher degree in the graph, in all the neighbours, among them, we take that of higher degree, and to place it in the first subgraph. This step is repeated if the first subgraph has not reached its maximum size (equal to half the number of vertices of the graph), it stops when the higher-degree vertex has no neighbours, then we move to the next step.

The third step is to take the vertex with the highest degree already present in the first subgraph and compare these neighbours to put those of higher degree in the first subgraph until the first subgraph is filled. Then if needed, take another vertex with the highest degree in the first subgraph and repeat the same step. This step stops when the first subgraph reached its maximum size.

Finaly, the fourth and last step, is to look all vertices that are not in the first subgraph and to put them in the second subgraph.

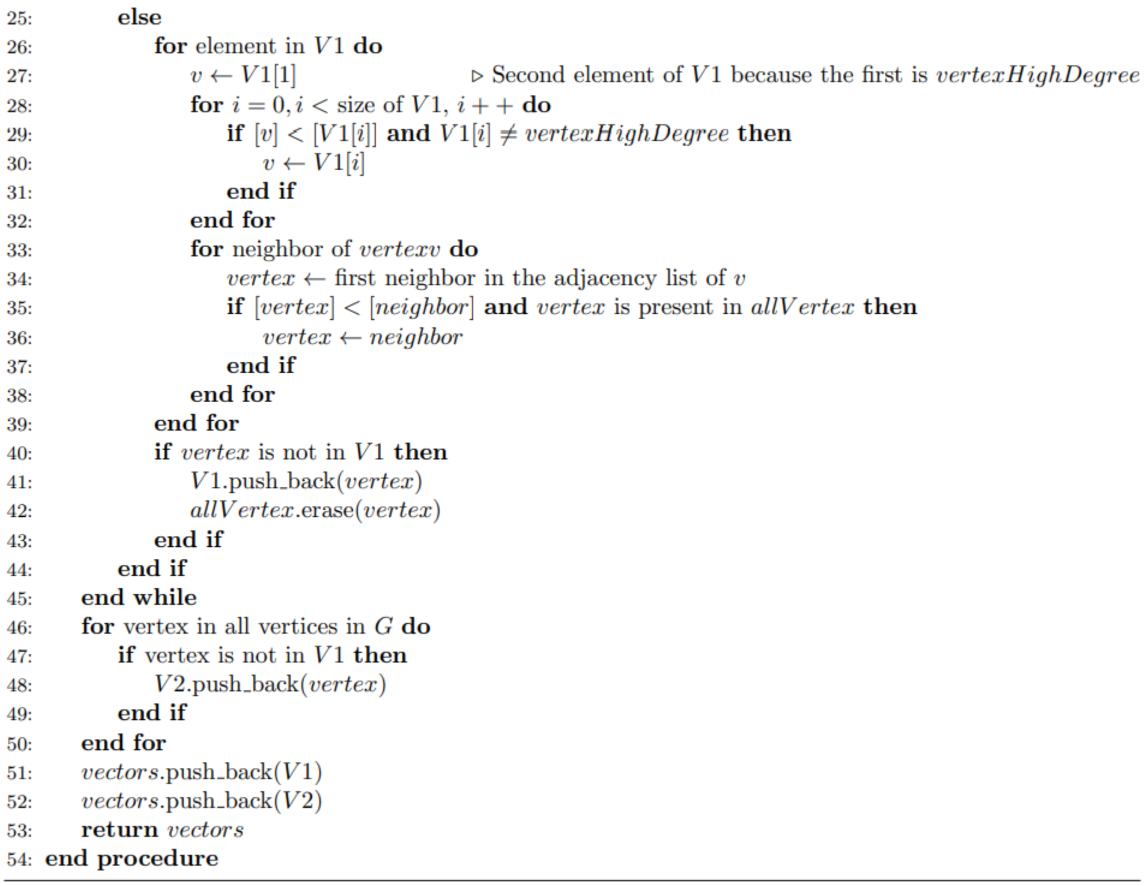
Pseudo-code

We can transcribe these steps through the following pseudo code:



Algorithm 1 Constructive Heuristic

```
procedure ConstructiveHeuristic($G$)
    $tabDegree \leftarrow$ array of $n$ elements filled with the degree of each vertex
    $allVertex \leftarrow$ array of $n$ elements filled with all vertices
    $nSubgraph \leftarrow$ int equal to the maximum number of vertices of each subgraph $(n/2)$
    $V1, V2 \leftarrow$ array of $nSubgraph$ elements
    $vectors \leftarrow$ array of arrays of $nSubgraph$ elements
    $vertexHighDegree \leftarrow$ vertex of $G$ with the highest degree
    $V1$.push_back($vertexHighDegree$)
    $allVertex$.erase($vertexHighDegree$)
    while size($V1$) $<$ $nSubgraph$ do
        $tabNeighbors \leftarrow$ array with all neighbors of $vertexHighDegree$ present in $allVertex$
        if $tabNeighbors$ is not empty then
            $v \leftarrow$ first neighbor of $vertexHighDegree$ in $tabNeighbors$
            $tabNeighbors$.erase($v$)
            for neighbor of $vertexHighDegree$ do
                if $tabDegree[v] < [neighbor]$ and $neighbor$ present in $allVertex$ then
                    $v \leftarrow neighbor$
                end if
            end for
            if $v$ is not in $V1$ then
                $V1$.push_back($v$)
                $allVertex$.erase($v$)
                $tabNeighbors$.erase($v$)
            end if
```



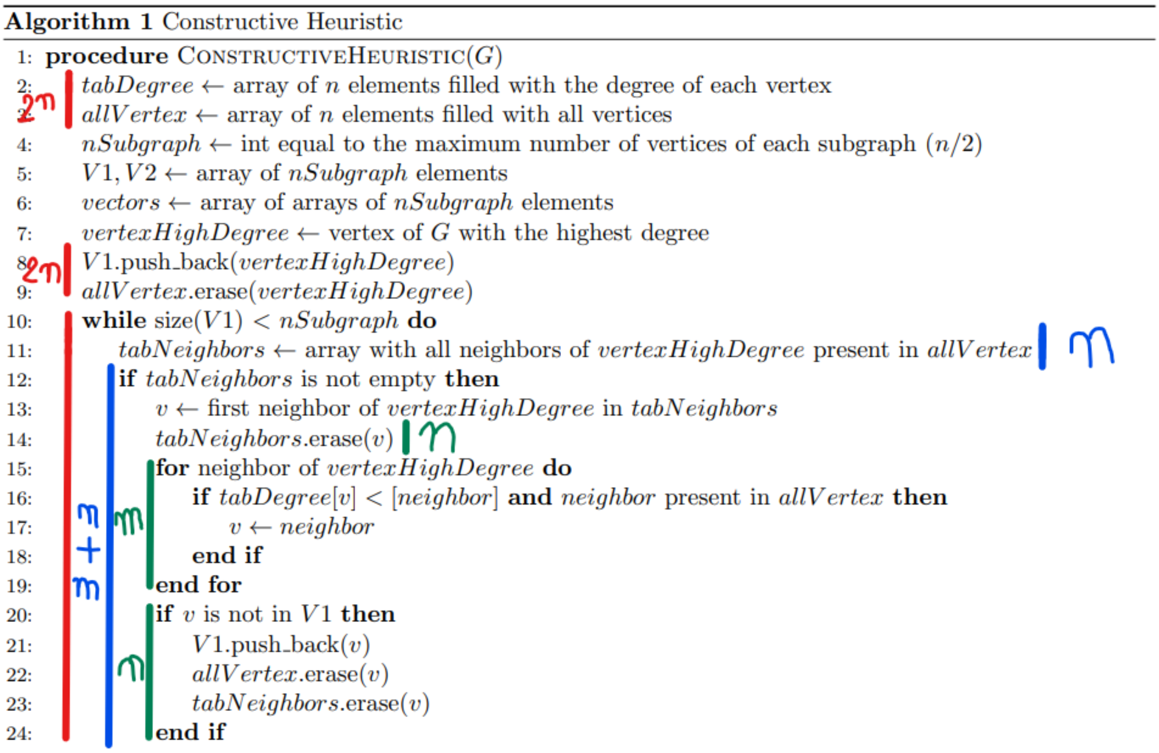
```
        else
            for element in $V1$ do
                $v \leftarrow V1[1]$                ▷ Second element of $V1$ because the first is $vertexHighDegree$
                for $i = 0, i <$ size of $V1$, $i++$ do
                    if $[v] < [V1[i]]$ and $V1[i] \neq vertexHighDegree$ then
                        $v \leftarrow V1[i]$
                    end if
                end for
                for neighbor of $vertexv$ do
                    $vertex \leftarrow$ first neighbor in the adjacency list of $v$
                    if $[vertex] < [neighbor]$ and $vertex$ is present in $allVertex$ then
                        $vertex \leftarrow neighbor$
                    end if
                end for
            end for
            if $vertex$ is not in $V1$ then
                $V1$.push_back($vertex$)
                $allVertex$.erase($vertex$)
            end if
        end if
    end while
    for vertex in all vertices in $G$ do
        if vertex is not in $V1$ then
            $V2$.push_back($vertex$)
        end if
    end for
    $vectors$.push_back($V1$)
    $vectors$.push_back($V2$)
    return $vectors$
end procedure
```

Complexity

Now we calculate the complexity of this pseudo-code, in first, to fill tabDegree and allVertex, the complexity is O(n) each. The functions erase and push_back, as also a complexity of O(n).

Then, we enter in the while, to fill V1, we fill tabNeighbors with a complexity of O(n), next in the first if, when we check if tabNeighbors is not empty, the complexity is :

$$total = n + \sum_{i=1}^{n} d(v_i) + 2n = \ 3n + m \ \in O(n+m)$$



Algorithm 1 Constructive Heuristic

```
procedure ConstructiveHeuristic($G$)
    $tabDegree$ ← array of $n$ elements filled with the degree of each vertex
    $allVertex$ ← array of $n$ elements filled with all vertices
    $nSubgraph$ ← int equal to the maximum number of vertices of each subgraph ($n/2$)
    $V1, V2$ ← array of $nSubgraph$ elements
    $vectors$ ← array of arrays of $nSubgraph$ elements
    $vertexHighDegree$ ← vertex of $G$ with the highest degree
    $V1$.push_back($vertexHighDegree$)
    $allVertex$.erase($vertexHighDegree$)
    while size($V1$) < $nSubgraph$ do
        $tabNeighbors$ ← array with all neighbors of $vertexHighDegree$ present in $allVertex$
        if $tabNeighbors$ is not empty then
            $v$ ← first neighbor of $vertexHighDegree$ in $tabNeighbors$
            $tabNeighbors$.erase($v$)
            for neighbor of $vertexHighDegree$ do
                if $tabDegree[v]$ < $[neighbor]$ and $neighbor$ present in $allVertex$ then
                    $v$ ← $neighbor$
                end if
            end for
            if $v$ is not in $V1$ then
                $V1$.push_back($v$)
                $allVertex$.erase($v$)
                $tabNeighbors$.erase($v$)
            end if
```

Besides, in the else, when tabNeighbors is empty, we have a complexity equal to:

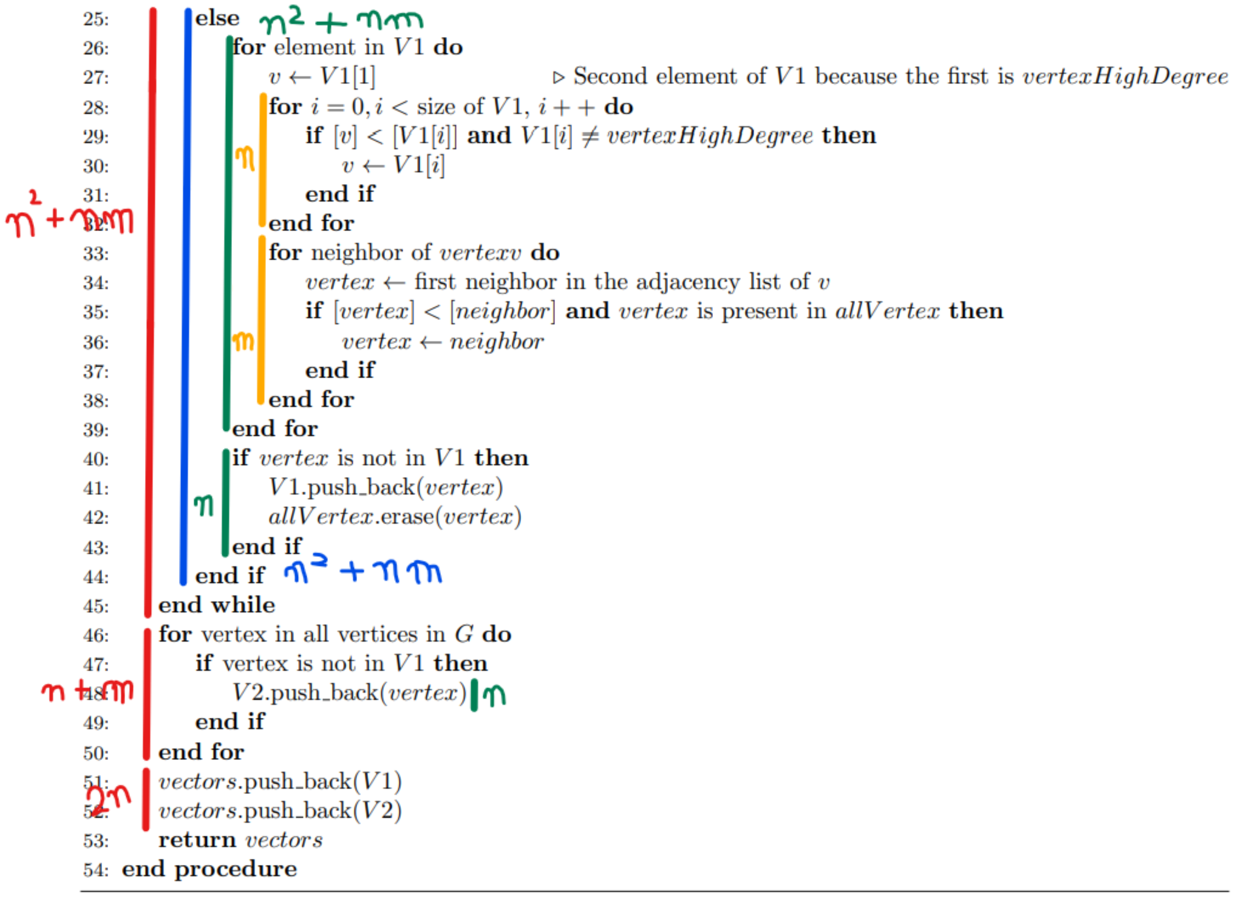
$$total = n * (n + \sum_{i=1}^{n} d(v_i)) + 2n = n * (n + m) + 2n \ \in O(n^2 + nm)$$

So, in the while we have the following complexity:

$$total = (n + m) + (n^2 + nm) \ \in O(n^2 + nm)$$

After the while, when we fill V2, we check all vertices of the graph, so the complexity is O(n+m).

Finally, we push V1 and V2 in the vector final, as well as the complexity is O(n).



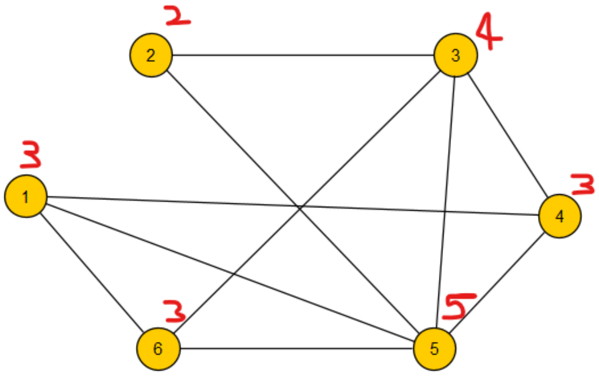
```
    else   $n^2 + nm$
        for element in $V1$ do
            $v \leftarrow V1[1]$                ▷ Second element of $V1$ because the first is $vertexHighDegree$
            for $i = 0, i <$ size of $V1$, $i++$ do      $n$
                if $[v] < [V1[i]]$ and $V1[i] \neq vertexHighDegree$ then
                    $v \leftarrow V1[i]$
                end if
            end for      $n^2 + nm$
            for neighbor of $vertexv$ do
                $vertex \leftarrow$ first neighbor in the adjacency list of $v$
                if $[vertex] < [neighbor]$ and $vertex$ is present in $allVertex$ then      $m$
                    $vertex \leftarrow neighbor$
                end if
            end for
        end for
        if $vertex$ is not in $V1$ then
            $V1$.push_back($vertex$)      $n$
            $allVertex$.erase($vertex$)
        end if
    end if   $n^2 + nm$
end while
for vertex in all vertices in $G$ do
    if vertex is not in $V1$ then
        $V2$.push_back($vertex$)   $n$      $n + m$
    end if
end for
$vectors$.push_back($V1$)      $2n$
$vectors$.push_back($V2$)
return $vectors$
end procedure
```

To conclude, the function for the constructive heuristic has a complexity of:

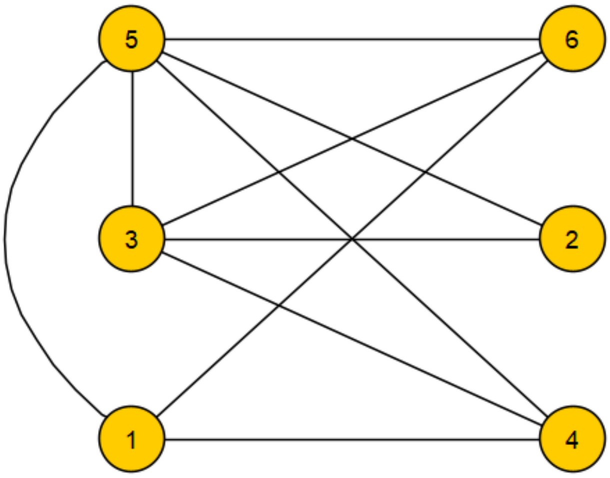
$$total = 4n + (n^2 + nm) + (n + m) + 2n \in \boldsymbol{O(n^2 + nm)}$$

Solution

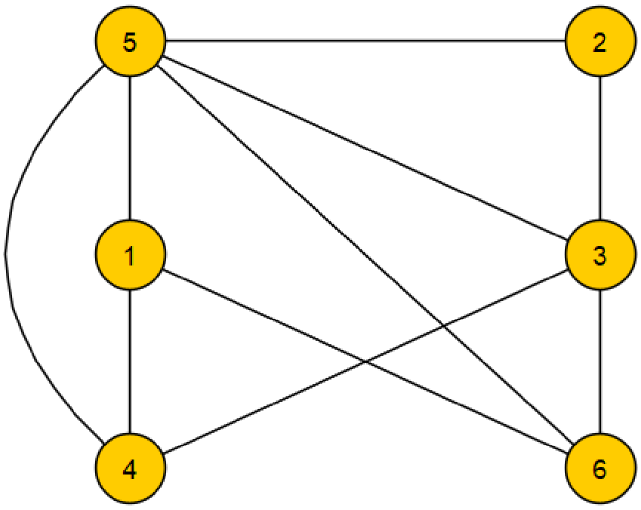
There are some cases, with some graphs where the solution obtained thanks to our constructive heuristic is not the optimal solution. In example, the next graph G with 6 vertices (whose degree of each vertex is marked in red) and 10 edges:



When we execute our constructive heuristic algorithm on this graph. First, we take the vertex with the highest degree: 5. Then we take a neighbour of 5 with the highest degree and can add 3 to the first subgraph. Finaly, the neighbour of 5 and 3 with the highest degree is 1. So the algorithm return V1= {5,3,1} and V2 = {6,2,4}, and when we calculate the number of edges between V1 and V2, edgeCommuns = 8. In fact, we get:



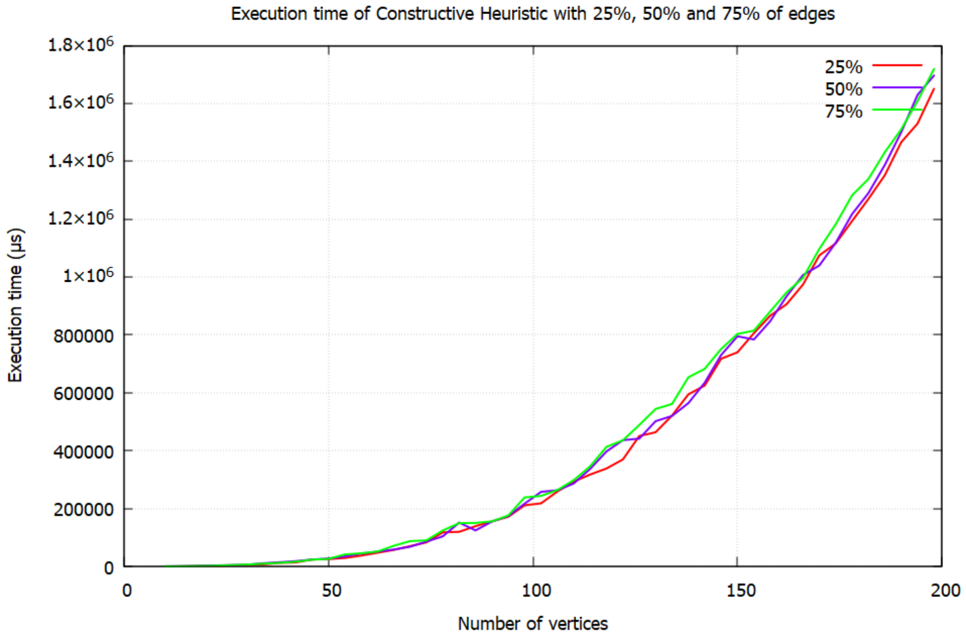
However, we see that there is a more optimal solution, with the number of edges between V1 = {5,1,4} and V2 = {2,3,6} is only 5 edges.



Indeed, we noticed that when the graph is almost complete and the number of vertices is not high, the solution obtained is not the most optimal. We can see that also, in the time complexity. The constructive heuristic algorithm has a complexity of $(n^2 + nm)$, as well as, when the number of vertices is not high, but the number of edges is high, the constructive heuristic has a higher time complexity.

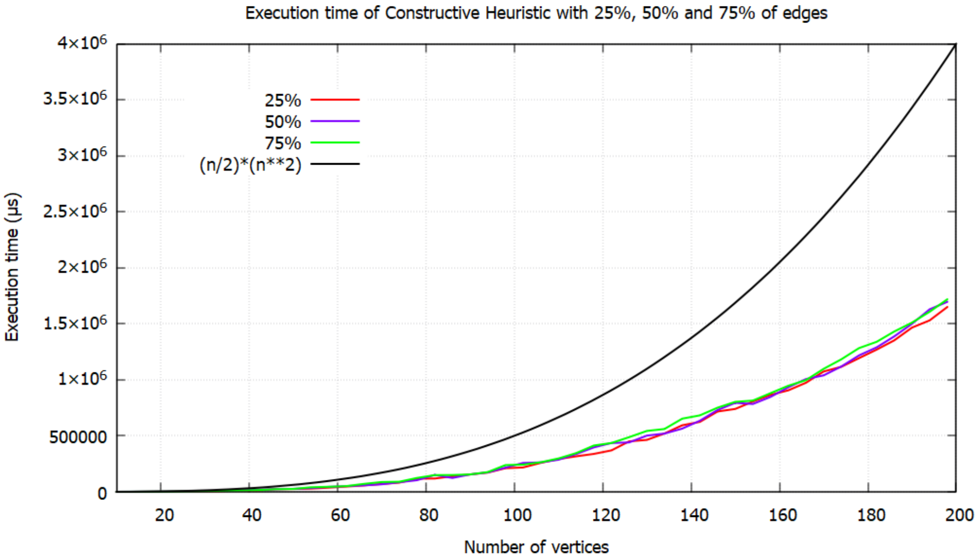
Execution time

This is the execution time (in microseconds) of the constructive heuristic algorithm depending on the number of vertices. We have tested it from 10 vertices to 200 vertices with a step of 4 for each iteration, for 3 cases of different edge probability: 25%, 50% and 75%.



We can note that the three curves, corresponding to the three cases of percentage of edges, are very close to each other. In fact, the percentage of edges, when the number of vertices is high, doesn't have much influence on this algorithm. We can globally observe that the algorithm is slightly slower with a percentage of edges equal to 75, but it is negligible.

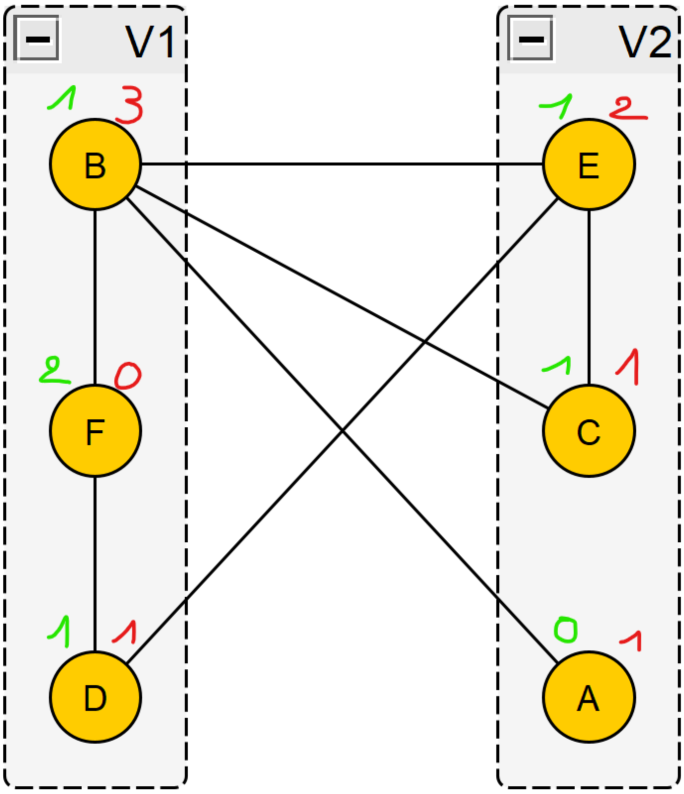
In addition, we can compare the execution times for all percentage of edges to our previously calculated temporal complexity:



Local search heuristic to solve the MBP

Explanation

For the local search, we first take the constructive solution. We have the original graph G with two vectors V1 and V2 that contains the index of the vertices. The idea is to swap vertices x1 from V1 with vertices x2 from V2. However, it would be dumb to swap every vertex to check if the result is better than the previous one. To minimise the number of swaps we configure a condition to make them. To illustrate the condition, let's use the following graph obtained after using the constructive algorithm.

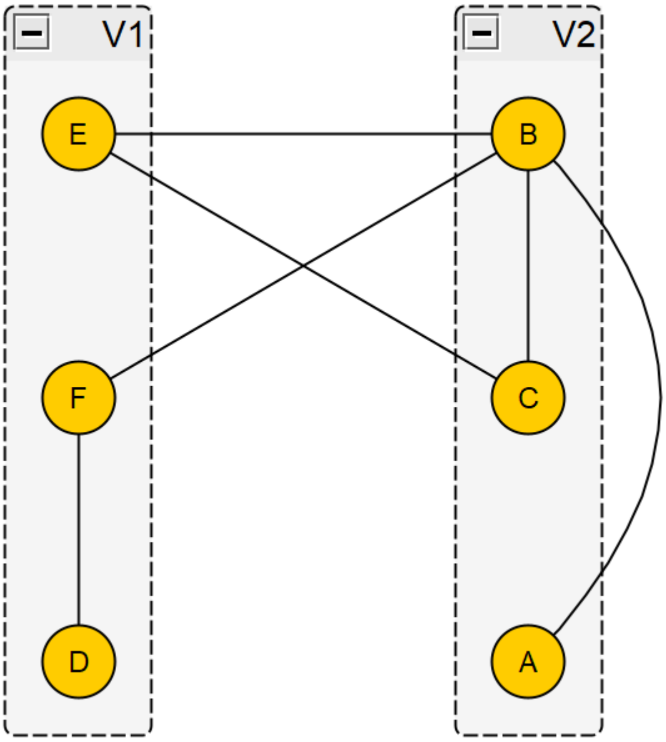


On the left we have the vertices in V1 and on the right we have the vertices in V2. In red we have what we will call "out degree" and in green the "in degree". The idea is the same as for the undirected graph. The in degree of a vertex is the number of edges that a vertex has inside his own subgraph, V1 or V2. For example, F has an in degree of 2 because he shares an edge with B and D that are in the same subgraph, V1. The out degree is the number of edges that a vertex has outside his own subgraph. For example, B is inside V1, so his out degree is the number of edges that he shares with vertices from V2, here with E, C and A. So, the out degree of B is 3.

Finaly, the condition to swap to vertices is: if the in degree of x1 and x2 is inferior to the out degree of x1 and x2, minus one if the edge between x1 and x2 exist, we can swap x1 and x2. If we respect that condition, the number of edges between V1 and V2 will decrease.

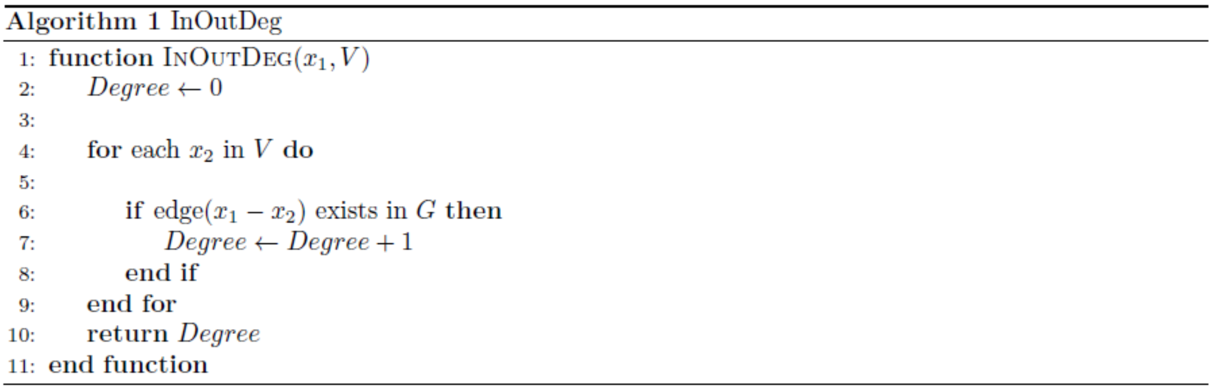
We obtained these conditions because we want to maximize the indegree of each V1 and V2 and minimize the outdegree. So, we don't want to have an outdegree superior to the indegree.

After running the algorithm, we obtain the following graph after the first iteration and there is no upgrade available with the local search.



Pseudo-code

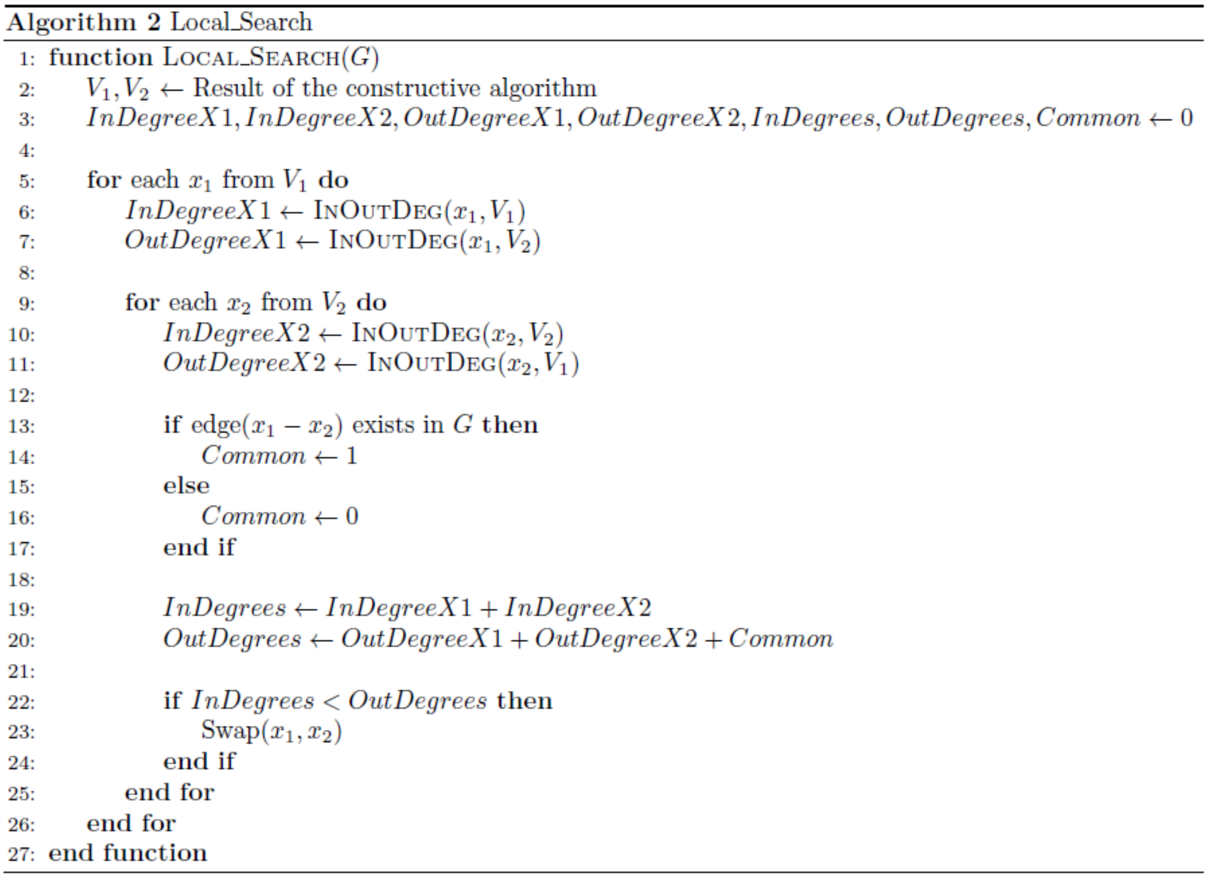
With this we can create the following pseudo-code. The function InOutDeg compute the indegree or the outdegree of a vertex. The difference between them is made with the input vector. If x1 is in V, it will automatically compute the indegree. Otherwise, if x1 is not in V, it will automatically compute the outdegree.



Algorithm 1 InOutDeg

```
function INOUTDEG($x_1$, $V$)
    $Degree \leftarrow 0$

    for each $x_2$ in $V$ do

        if edge($x_1 - x_2$) exists in $G$ then
            $Degree \leftarrow Degree + 1$
        end if
    end for
    return $Degree$
end function
```

We will use it in the main local search algorithm bellow, corresponding the description before.



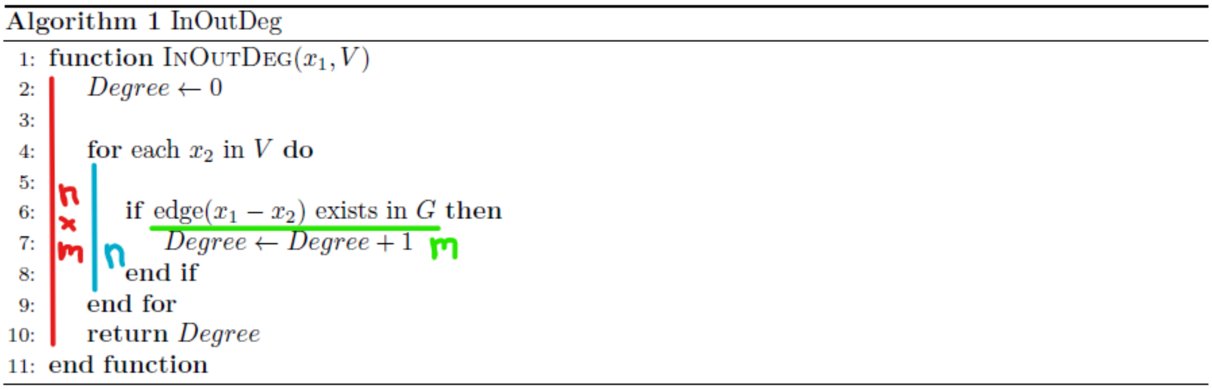
```
Algorithm 2 Local_Search
function LOCAL_SEARCH($G$)
    $V_1, V_2 \leftarrow$ Result of the constructive algorithm
    $InDegreeX1, InDegreeX2, OutDegreeX1, OutDegreeX2, InDegrees, OutDegrees, Common \leftarrow 0$

    for each $x_1$ from $V_1$ do
        $InDegreeX1 \leftarrow$ INOUTDEG($x_1, V_1$)
        $OutDegreeX1 \leftarrow$ INOUTDEG($x_1, V_2$)

        for each $x_2$ from $V_2$ do
            $InDegreeX2 \leftarrow$ INOUTDEG($x_2, V_2$)
            $OutDegreeX2 \leftarrow$ INOUTDEG($x_2, V_1$)

            if edge($x_1 - x_2$) exists in $G$ then
                $Common \leftarrow 1$
            else
                $Common \leftarrow 0$
            end if

            $InDegrees \leftarrow InDegreeX1 + InDegreeX2$
            $OutDegrees \leftarrow OutDegreeX1 + OutDegreeX2 + Common$

            if $InDegrees < OutDegrees$ then
                Swap($x_1, x_2$)
            end if
        end for
    end for
end function
```

Complexity

Now we compute the complexity of this pseudo-code, in first, we do it for the InOutDeg function. We add 1 to the total degree if the edge between X1 and X2 exist. To verify this we use a function in O(d(x1)) because we use an adjacency list. We do it for each X2 in V2, in O(n).

$$InOutDeg = \sum_{i=1}^{n/2} d(x_i) = n \times d(x_i) \in O(n \times d(x_i))$$

When we wrote the complexity, we made a mistake and used m instead of d(x1). So, considerate that in the following pseudo-code, when we estimate the complexity in O(m) it actually is in O(d(x_i)).



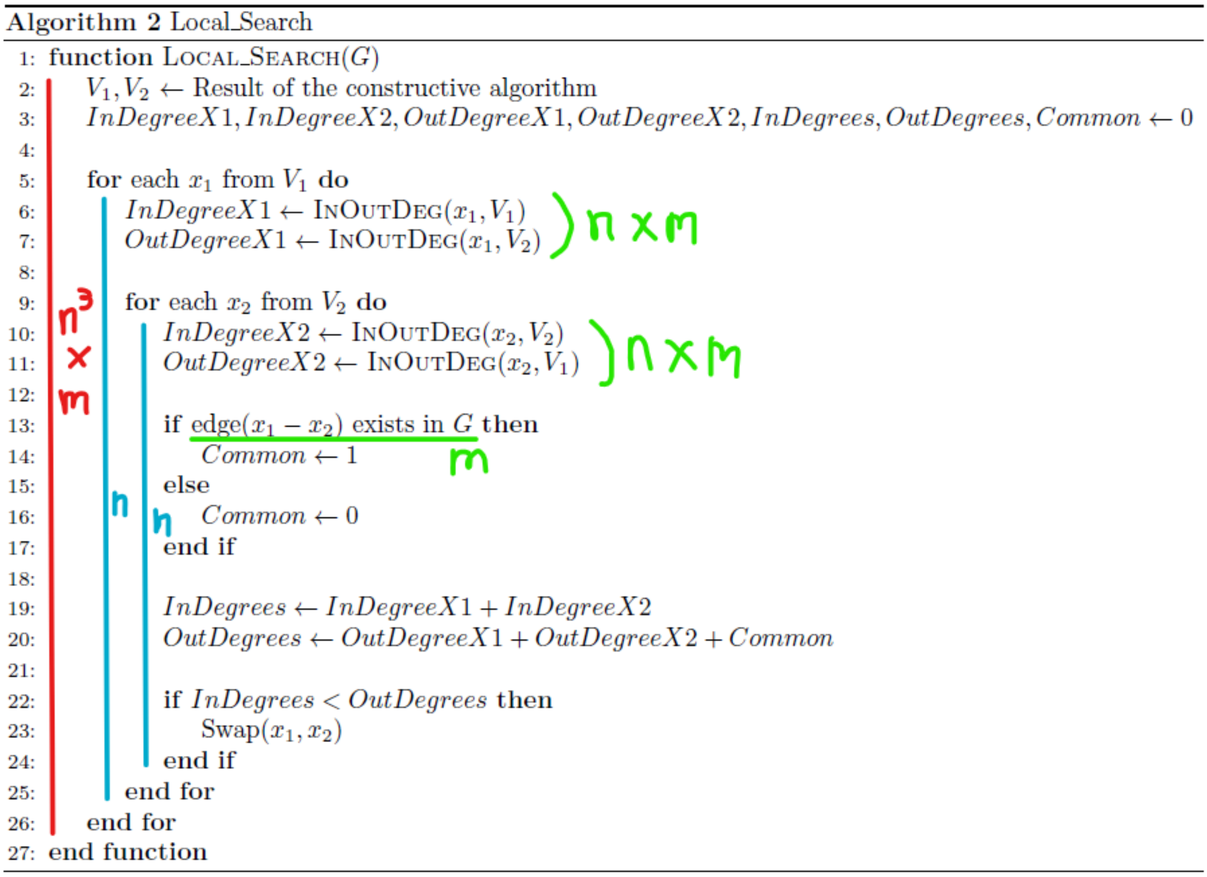
Algorithm 1 InOutDeg

```
function InOutDeg($x_1$, $V$)
    $Degree \leftarrow 0$

    for each $x_2$ in $V$ do

        if edge($x_1 - x_2$) exists in $G$ then
            $Degree \leftarrow Degree + 1$
        end if
    end for
    return $Degree$
end function
```

Now for the main algorithm, for each vertex X1 in V1 we compute the InOutDeg algorithm. Then for each X1, we do for each vertex X2 in V2 the InOutDeg algorithm.

$$Local = \sum_{i=1}^{n/2} (n \times d(x_i) + \sum_{j=1}^{n/2} n \times d(x_i) + d(x_i)) \ \in O(n^3 \times d(x_i))$$

When we wrote the complexity, we made a mistake and used m instead of d(x$_i$). So, considerate that in the following pseudo-code, when we estimate the complexity in O(m) it actually is in O(d(x$_i$)).



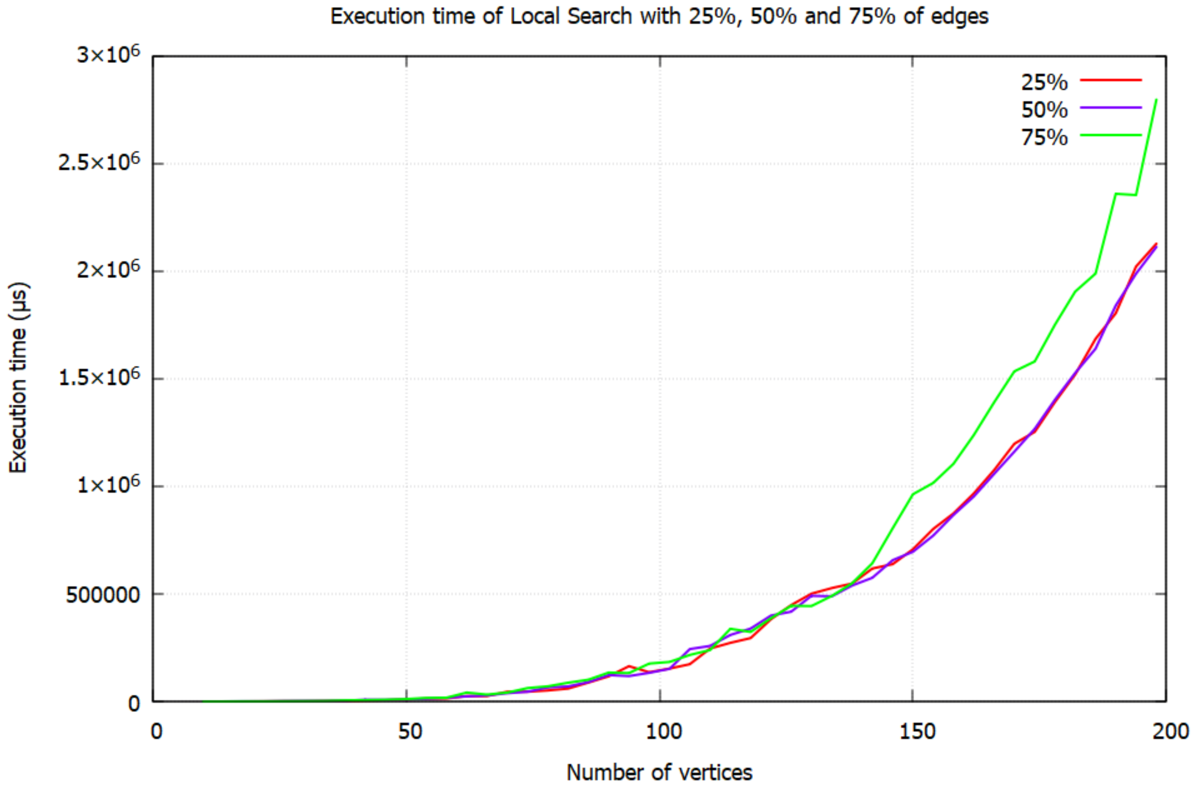
Algorithm 2 Local_Search

```
function Local_Search($G$)
    $V_1, V_2 \leftarrow$ Result of the constructive algorithm
    $InDegreeX1, InDegreeX2, OutDegreeX1, OutDegreeX2, InDegrees, OutDegrees, Common \leftarrow 0$

    for each $x_1$ from $V_1$ do
        $InDegreeX1 \leftarrow$ InOutDeg($x_1, V_1$)
        $OutDegreeX1 \leftarrow$ InOutDeg($x_1, V_2$)

        for each $x_2$ from $V_2$ do
            $InDegreeX2 \leftarrow$ InOutDeg($x_2, V_2$)
            $OutDegreeX2 \leftarrow$ InOutDeg($x_2, V_1$)

            if edge($x_1 - x_2$) exists in $G$ then
                $Common \leftarrow 1$
            else
                $Common \leftarrow 0$
            end if

            $InDegrees \leftarrow InDegreeX1 + InDegreeX2$
            $OutDegrees \leftarrow OutDegreeX1 + OutDegreeX2 + Common$

            if $InDegrees < OutDegrees$ then
                Swap($x_1, x_2$)
            end if
        end for
    end for
end function
```

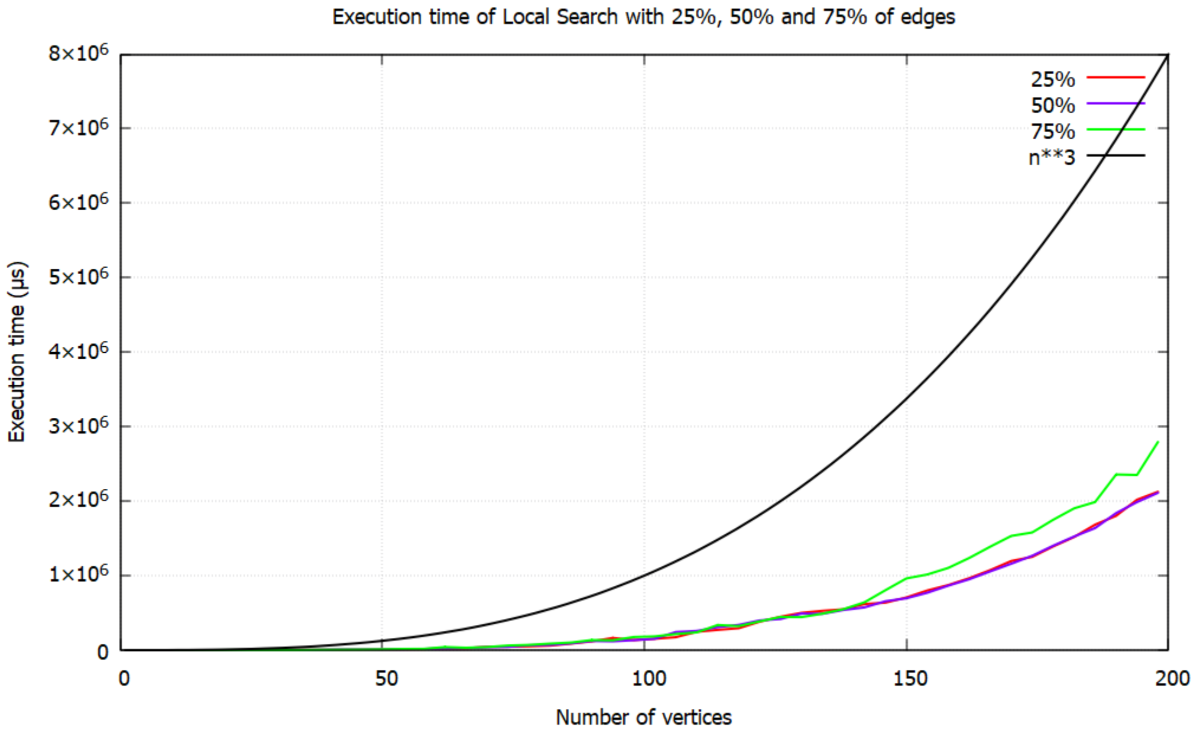
Execution time

For now, let's study the application of the complexity that we estimated before. For each of the curve bellow, we have in abscissa the number of vertices, which in those case goes from 10 vertices to 200 vertices. For each iteration the number of edges increases by 4. Then on the ordinate we have the execution time in microseconds.

First, let's compare the execution time between 3 different cases, depending on the probability of having an edge when we create the graph. As we can see bellow, the execution time are close between each other even though for a probability of 75% of having an edge it is a little bit longer. Indeed, the complexity, O (n^3 x $d(x_i)$), mainly depends on the number of edges rather than the degree of the vertex.



Then when we compare to the theorical complexity obtained before. We use the function f(x) = x^3 and we obtain the following curves.



As we can see, the expectation is beyond the results obtained. We can explain this because the implemented algorithm might have variation that affects its actual complexity.

Tabu Search meta-heuristic to solve the MBP

Explanation

Unlike the local search algorithm, which seeks a local maximum, the Tabu-Search will seek a global maximum for the Minimum Bisection Problem.

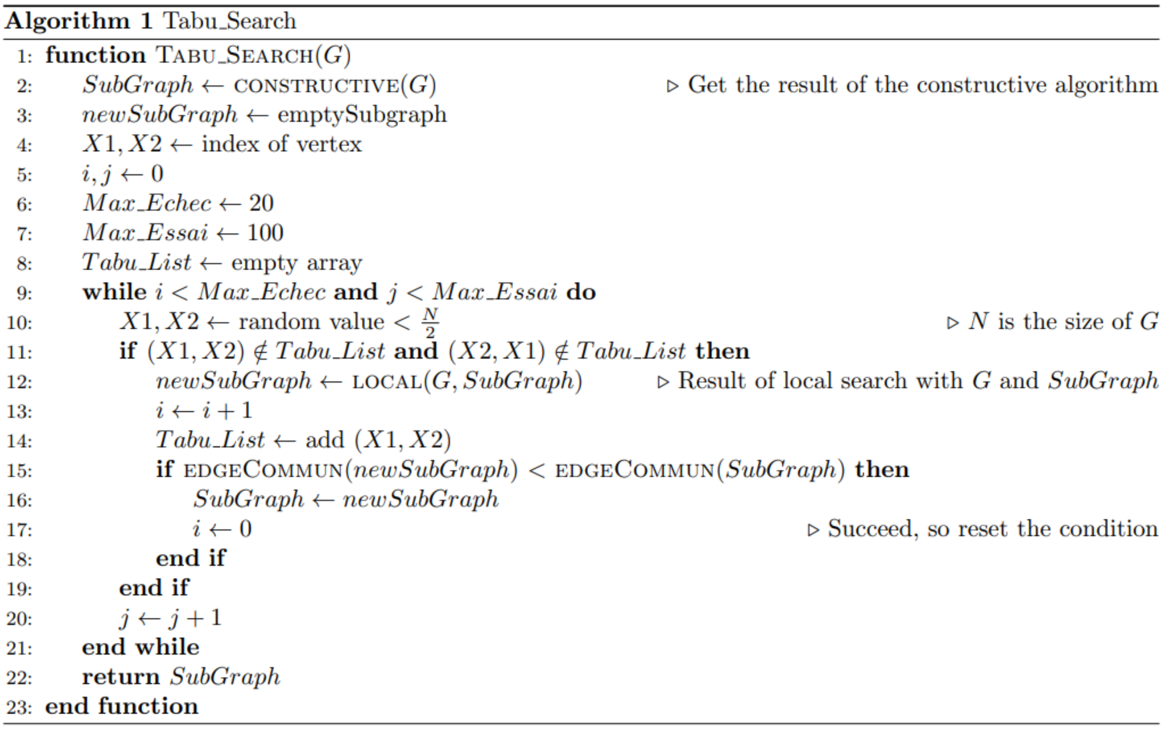
In this algorithm, we start with an initial solution from the constructive algorithm, which will act as the best result, and we define two stopping conditions. The first, Max_Fail, will stop the algorithm after several tests that do not allow to have a better solution, here we choose Max_Fail = 20, we create a counter "i" that will check the condition Max_Fail. The second stop condition, Max_Try, allows to have a maximum number of tests of 100, we create a counter "j", which will check this condition. We chose those values for Max_Fail and

We create the Tabu_List, an array that contains all the vertex pairs that have been exchanged, it is filled as the iterations progress.

As long as one of the two conditions, Max_Fail or Max_Try, is not met, a random vertex is taken in V1 and V2. If they do not belong to the Tabu_List table, we swap them and add the vertex pair exchanged in Tabu_List. We then perform the local search on this new graph. If the result is better than the old one, it becomes the best and "i" is reset because a solution has been found. Finally, we increase "i" and "j" since we are at the end of the test.

Pseudo-code

Here is the pseudo-code corresponding to the Tabu search algorithm. Note that Max_Fail is written Max_Echec and Max_Try is written Max_Essai



Algorithm 1 Tabu_Search

```
function TABU_SEARCH($G$)
    $SubGraph \leftarrow$ CONSTRUCTIVE($G$)            ▷ Get the result of the constructive algorithm
    $newSubGraph \leftarrow$ emptySubgraph
    $X1, X2 \leftarrow$ index of vertex
    $i, j \leftarrow 0$
    $Max\_Echec \leftarrow 20$
    $Max\_Essai \leftarrow 100$
    $Tabu\_List \leftarrow$ empty array
    while $i < Max\_Echec$ and $j < Max\_Essai$ do
        $X1, X2 \leftarrow$ random value $< \frac{N}{2}$                        ▷ $N$ is the size of $G$
        if $(X1, X2) \notin Tabu\_List$ and $(X2, X1) \notin Tabu\_List$ then
            $newSubGraph \leftarrow$ LOCAL($G, SubGraph$)    ▷ Result of local search with $G$ and $SubGraph$
            $i \leftarrow i + 1$
            $Tabu\_List \leftarrow$ add $(X1, X2)$
            if EDGECOMMUN($newSubGraph$) < EDGECOMMUN($SubGraph$) then
                $SubGraph \leftarrow newSubGraph$
                $i \leftarrow 0$                               ▷ Succeed, so reset the condition
            end if
        end if
        $j \leftarrow j + 1$
    end while
    return $SubGraph$
end function
```

Complexity

As said before, when we wrote the complexity, we made a mistake and used m instead of d(x1). So, considerate that in the following pseudo-code, when we estimate the complexity in O(m) it actually is in O(d(v_i)).

The constructive algorithm is used to obtain the initial solution. It's complexity is O(n**2 + d(v)), d(v) the degree of a vertex.

The main loop of the algorithm ends at worst when j >= Max_Try, so it's complexity is O(1).

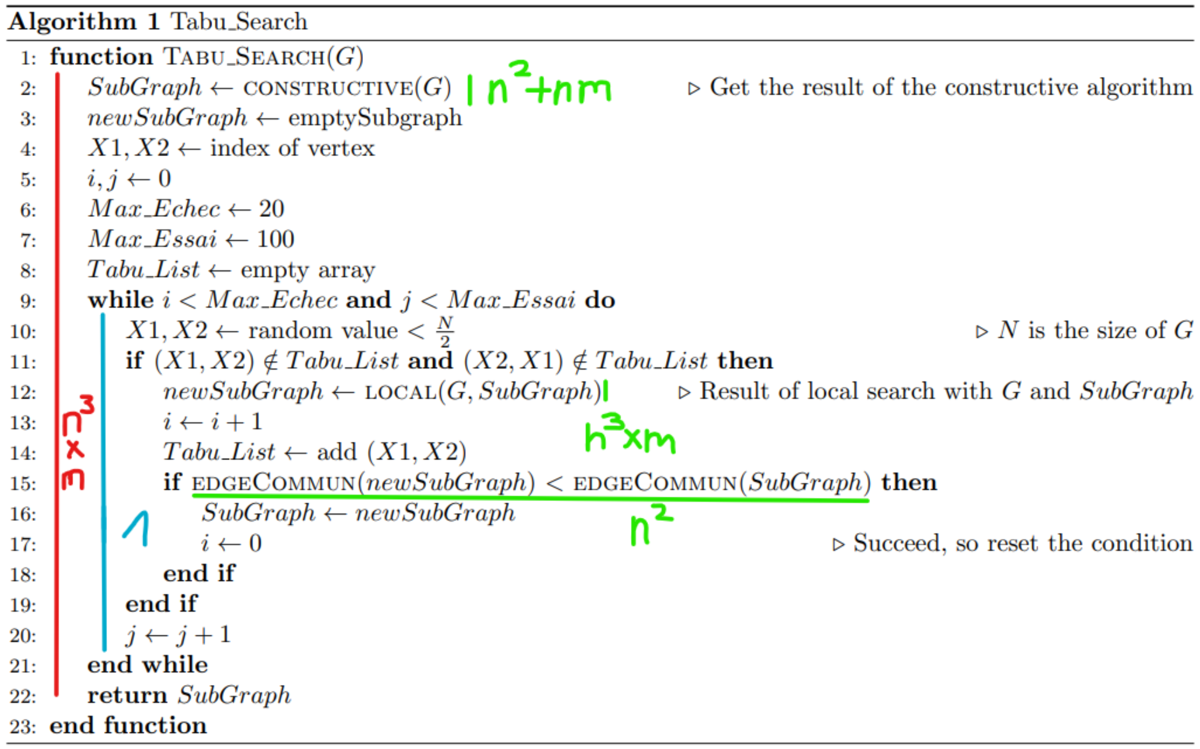
Checking if (X1, X2) or (X2, X1) is in the Tabu list is O (1) because we go through the list only once.

The local search complexity used in the loop is O (n**3 x d(v_i)).

Computing the number of edges there is between the two subgraph is O(n**2) because for each vertex in V1 we check if there is an edge between the vertex and every vertex in V2.

So, the total complexity of the Tabu search is:

$$Tabu \in O\left(n^2 + n * d(x_i)\right) + O(1) * O(n^3 \times d(x_i)) \in \boldsymbol{O}(\boldsymbol{n^3} \times \boldsymbol{d}(\boldsymbol{x_i}))$$



Algorithm 1 Tabu_Search

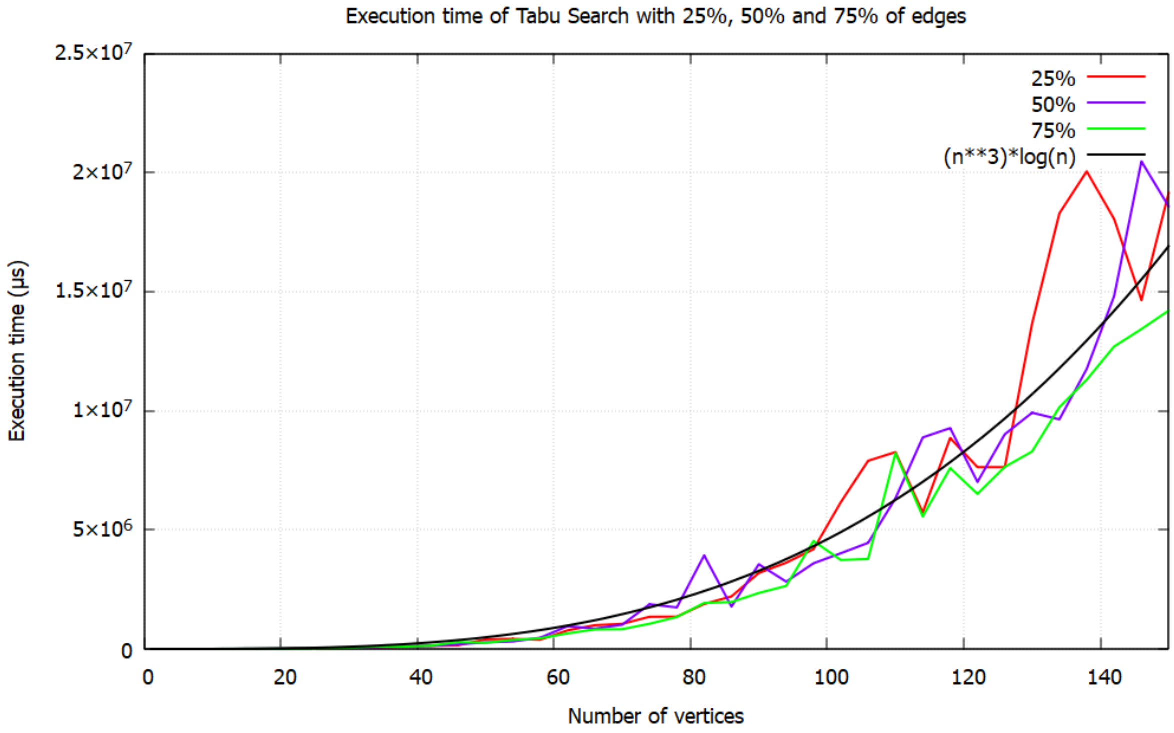
```
function TABU_SEARCH($G$)
    $SubGraph \leftarrow$ CONSTRUCTIVE($G$)   | $n^2+nm$        ▷ Get the result of the constructive algorithm
    $newSubGraph \leftarrow$ emptySubgraph
    $X1, X2 \leftarrow$ index of vertex
    $i, j \leftarrow 0$
    $Max\_Echec \leftarrow 20$
    $Max\_Essai \leftarrow 100$
    $Tabu\_List \leftarrow$ empty array
    while $i < Max\_Echec$ and $j < Max\_Essai$ do        $n^3 \times m$
        $X1, X2 \leftarrow$ random value $< \frac{N}{2}$        ▷ $N$ is the size of $G$
        if $(X1, X2) \notin Tabu\_List$ and $(X2, X1) \notin Tabu\_List$ then        $1$
            $newSubGraph \leftarrow$ LOCAL($G, SubGraph$)|        ▷ Result of local search with $G$ and $SubGraph$
            $i \leftarrow i + 1$        $n^3 \times m$
            $Tabu\_List \leftarrow$ add $(X1, X2)$
            if EDGECOMMUN($newSubGraph$) < EDGECOMMUN($SubGraph$) then        $n^2$
                $SubGraph \leftarrow newSubGraph$
                $i \leftarrow 0$        ▷ Succeed, so reset the condition
            end if
        end if
        $j \leftarrow j + 1$
    end while
    return $SubGraph$
end function
```

Execution time

We will study the time complexity that we estimated before with many examples. For each of the curve bellow, we have in abscissa the number of vertices, which in those case goes from 10 vertices to 15 vertices. For each iteration the number of edges increases by 4. Then on the ordinate we have the execution time in microseconds.

As we can see, all the curves obtained revolve around the curve of the function f(x) = (n**3) * log(n). But we can suppose that on average, the degree of a vertex is around log(n) because for n=100, on average a vertex can't have only 2 neighbours. Because, even if the probability of creating an edge is 25%, on average a vertex should have 25 neighbours which is clearly greater than 2.

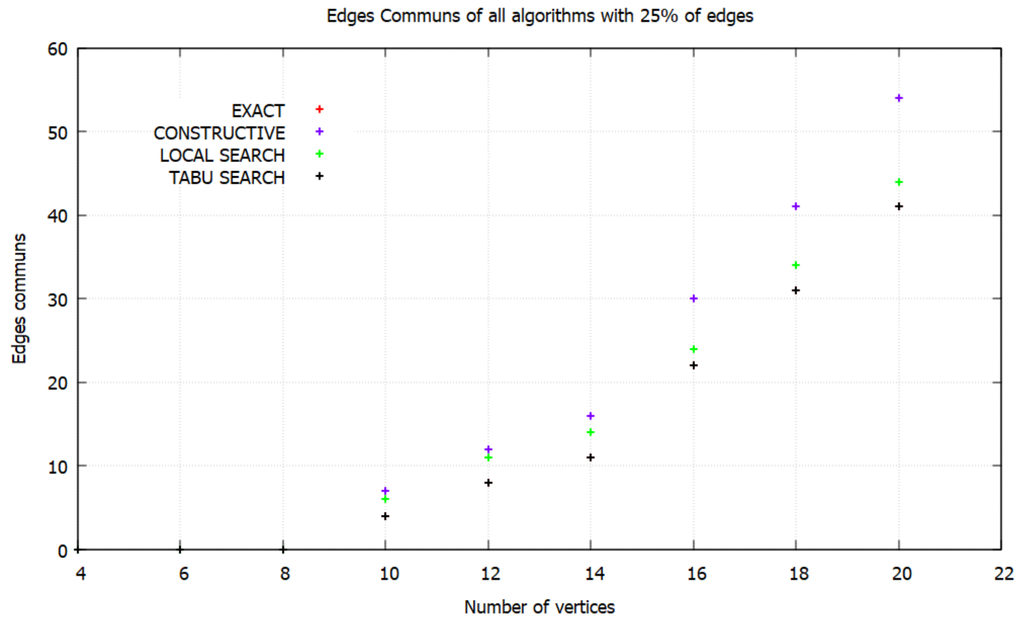
So, the values in each case are around O ((n**3) * log(n)) but we can't find any clue to explain it.



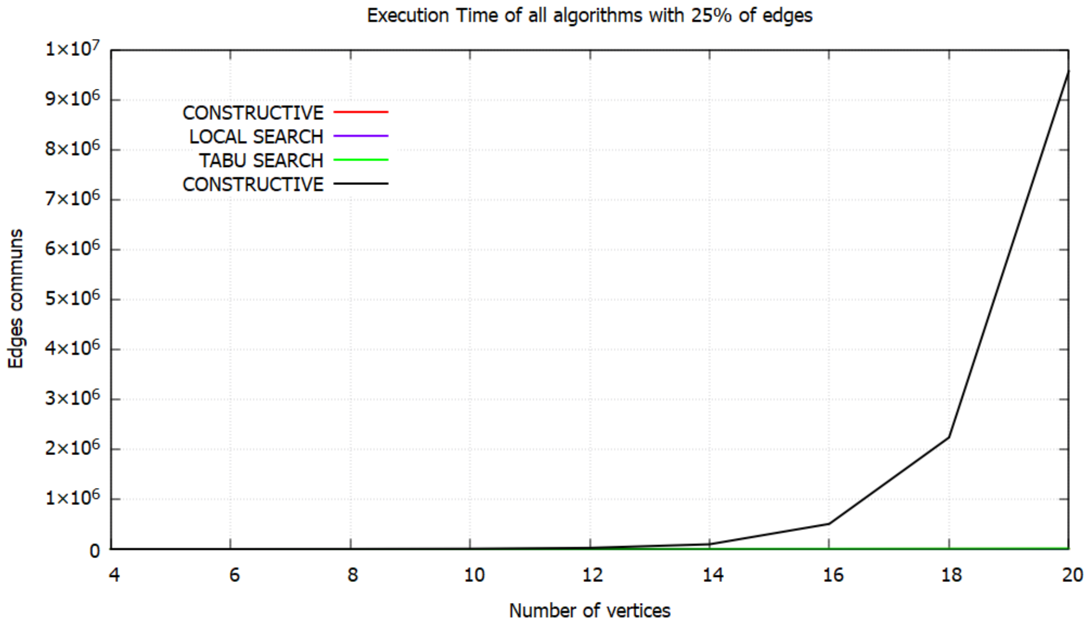
The fluctuation might be done because of the double break condition. We have a higher time of execution when we stop because we have reached the maximum iteration amount than when we stop because of the maximum failure iteration.

Comparison of all algorithms

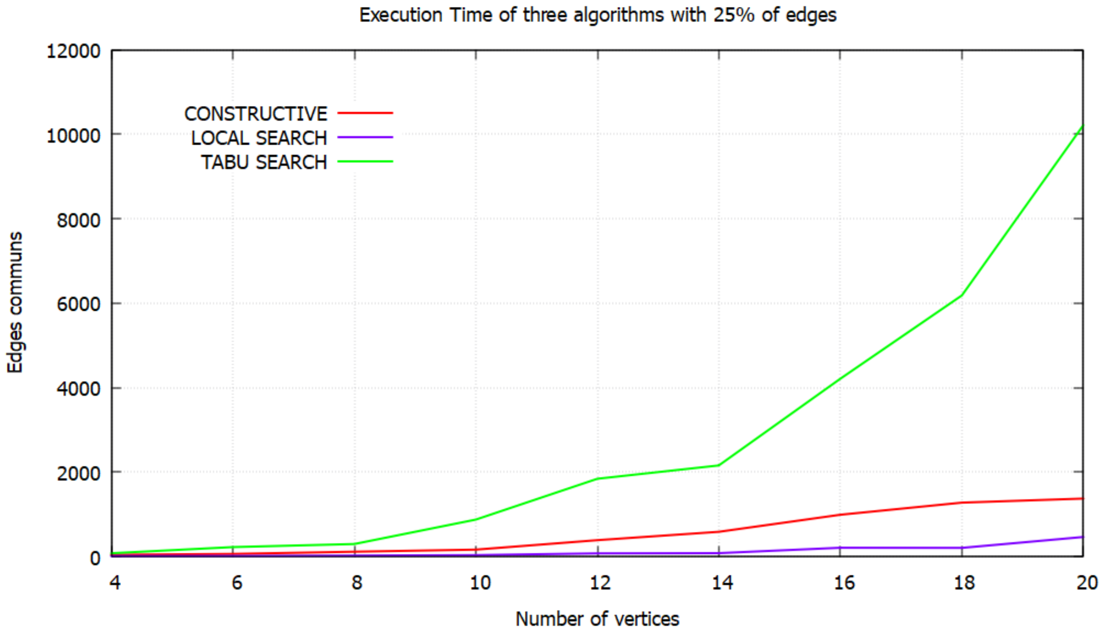
For the comparison between the four algorithms we first compared with a graph with 20 vertices and 25% of edges :



On this graphical, that compares the number of common edges according to the different algorithms, we can observe that the best result is obtain by the Exact algorithm and the Tabu Search (which are confused on the graphical). Moreover, we can observe that the Local Search algorithm obtained better results than the Constructive Heuristic algorithm.

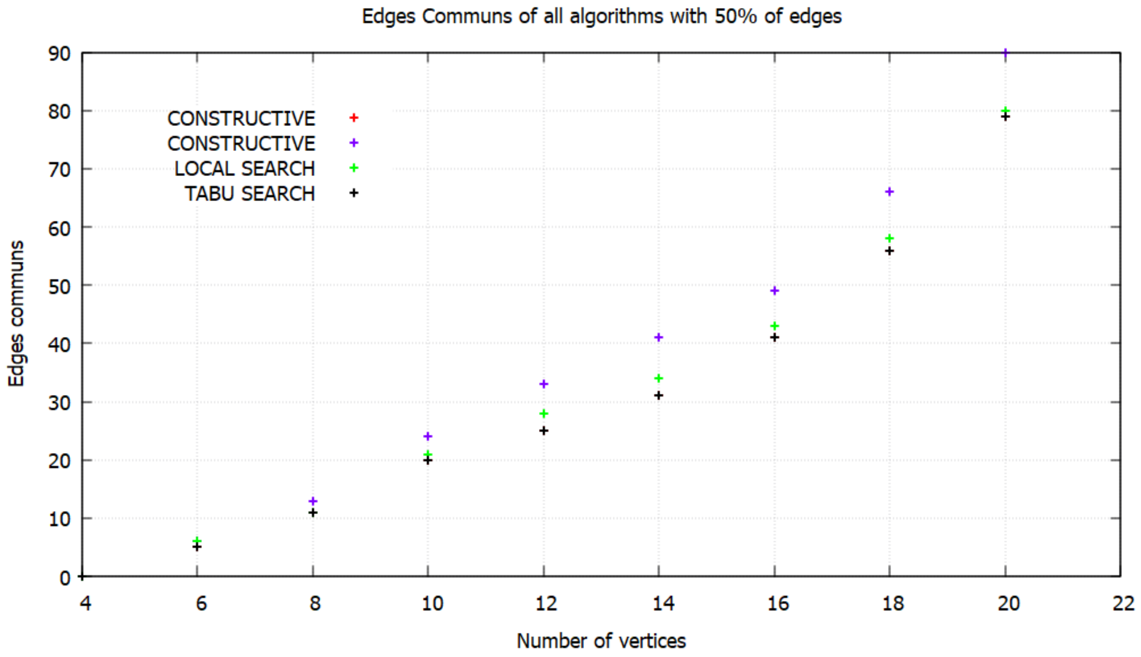


On this graphical that compares the execution time of each algorithm, we can notice that the Exact algorithm has a much higher execution time than other algorithms, so for the rest of the comparisons we will remove it from the graphical to better observe the behavior of the three others. So, we get this graphical:

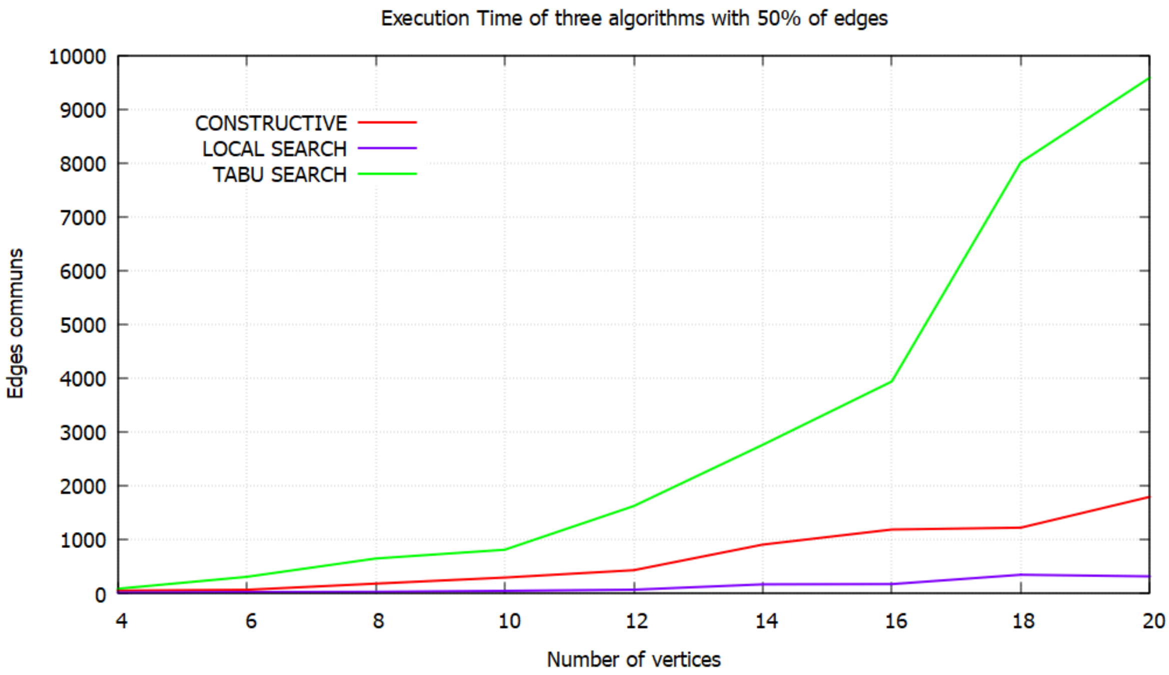


On this graphical, we can note that the Tabu Search has the highest execution time compared to the other two, because he uses the other two algorithms multiple times, but as noted previously it obtained better results for the number of communal edges. However, the Local Search and the Constructive heuristic are about five times faster than the Tabu Search.

To continue the comparison between the four algorithms we now compared with a graph with 20 vertices but with 50% of edges :



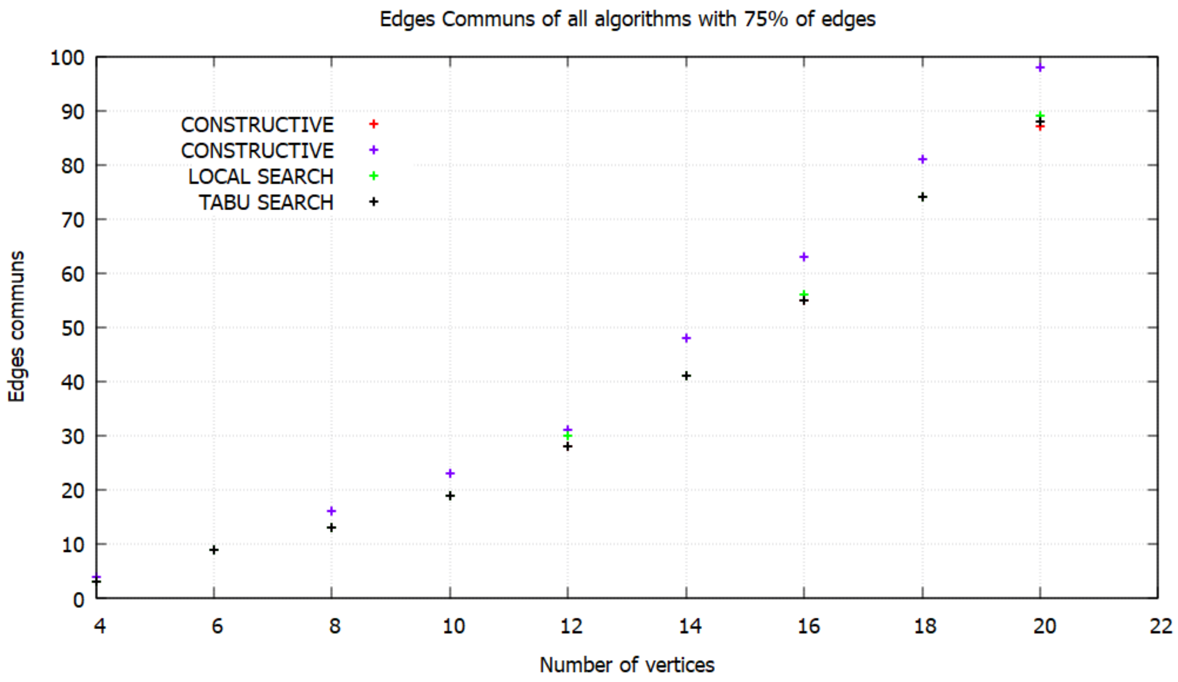
On this graphical, we can observe that the result of the edges commons between all algorithms are identical to the previous graphical proportionally adapted to the numbers of edges, the Tabu Search and the Exact get better results than the others. However, with a percentage of edges higher the Local search is getting closer to the two best.



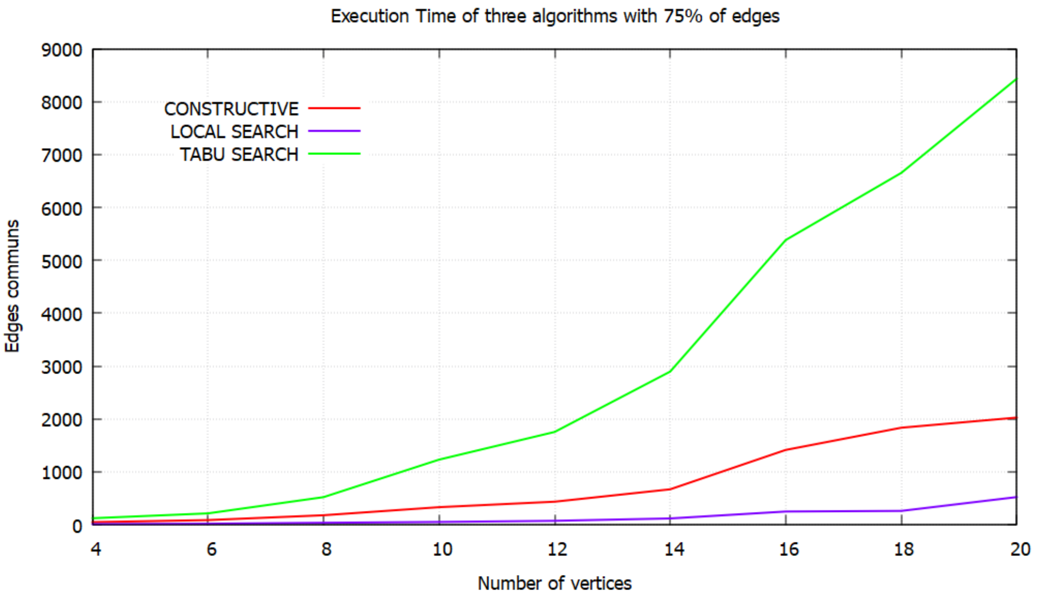
On the other hand, the execution time of the three algorithms remains about the same, even if the percentage of edges has increased.

We can deduce that the Local Search thus becomes more interesting with a higher percentage of edges in the graph, because it gets a better result in the number of commons edges with an execution time about the same. Unlike the Constructive Heuristic which does not evolve especially with a percentage of different edges. But, for now, the Tabu Search remains the best algorithm in terms of results, even if it remains longer at execution than the other two.

Finally, the comparison between the four algorithms we now compared with a graph with 20 vertices but with 75% of edges:



We can notice in this graphical that the results between all algorithms are very close, the Local Search algorithm gets almost the same result or the same result than the Exact and the Tabu Search algorithms for this percentage of edges in the graph.



Also, the execution time of all algorithms always remain about the same. So, the Local Search thus becomes very interesting with a percentage of edges to 75% in the graph.

Conclusion

During this project, we made in total four algorithms to solve the minimum bisection problem. The first algorithm, exact, check all possibilities. As we saw, it will always find the optimal solution, but the main problem is its execution time. For a graph with more than 20 vertices, the execution time feel like it takes for ever, above all when we compare to the other algorithms.

Secondly, we created the constructive algorithm. This algorithm takes a group of vertices, starting with the vertex with the highest degree. Then we had the neighbours in the subgraph with the highest degree. It allows us to maximize the connections in each subgraph. As seen before, the constructive algorithm gives us a solution about 10% worse than the optimal solution. However, it is the second faster algorithm from all.

Then, we created the local search algorithm. First, it takes the result of the previous algorithm. Then it will smartly swap vertices in V1 and V2. We swap only if the two vertices create more edges between the two subgraph V1 and V2 than in their own subgraph. As seen in the analysis part, the local search is the fastest algorithm, and it gives us a result really close to the best solution.

Finally, the Tabu search is an algorithm that can find the optimal solution in an amount of time more reasonable than the exact algorithm. It first takes an initial solution with the constructive algorithm. Then we randomly swap vertices in the subgraphs and use the local search algorithm. The algorithm stops when we reach the try or fail break condition. However, the algorithm is a bit slower than the two others because in the Tabu search uses both algorithms and multiple times.

To conclude, this project was hard to complete in time because of our poor organization due to the heavy workload in other courses. We could have improved on some points such as the complexity analysis. Moreover, we did not manage to communicate properly between us until few weeks, which slowed a lot the resolution of the MBP and made us lose time in the merge of our algorithms.