Group members : Etienne Lemonnier - Charlotte Mougenot - Guillaume Loquet - Sébastien Demousselle - Alexandre Duplouy

Class - Group : CIR3 - Team 1

Module : Graph Theory

Module teacher : Leandro Montero

Document title : Graph Theory Project

Sending date and hour : May 17th - 12pm

Words number : 2583

• By sending this document to the teacher, we certify that this report is ours and we beware of plagiarism and referencing rules.

Contents

1 Introduction

1.1 Maximum Edge Weight Clique Problem

In graph theory, a clique is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent (Wikipedia). In this project, edges have weights: a number is associated for each edge of a graph. The goal of the project is to find the clique which has the greatest weight, which is the sum of all interconnected edges of a clique.

1.2 Applications

This problem can be applied to groups of people where vertices are people and edges are points of interest. For example, if two people share Computer Science, Art and Litterature in points of interest, the weight of the edge between the two people will be 3 (number of points of interest). The maximum edge clique weight of a population represent the group of people where everyone have points of interest with each person of the group, and the points of interest are multiple. Then, the problem can detect the greatest group of friends of a population. Therefore, this problem can be applied to social networks, where cliques are group of users. The goal here is to maximise these cliques by recommending centers of interest to pair of users.

1.3 Test parameters

Moreover, for all the instances, we have used "Intel(R) Core(TM) i9-8950HK CPU @ 2.90GHz 2.90 GHz" (without overclocking), that runs approximately between 12% and 16%.
To generate all graphs, we use linear regression method to find the theoretical complexity coefficient (it is indicated in first in the complexity formula). Remember, this coefficient is unique to the tests on the same computer. Then, we can compare the time execution that we found to the worse theoretical time complexity.

2 Exact algorithm

2.1 Pseudo-code

Algorithm 1 Find the exact maximum weight clique of a graph using backtracking improvement

```
procedure EXACTMAIN(adjacencyList)
    maxClique ← empty structure with weight, array of vertices and size of array
    actualClique ← empty array of vertices
    verticesToTest ← empty array of vertices
    for each vertex in adjacencyList do
        for each neighbour of each vertex do                 ▷ reject unexisting edges
            if w(vertex, neighbour) > maxClique.weight then
                update(maxClique)                             ▷ updating cliques of two vertices
            end if
            clear actualClique
            clear verticesToTest
            put vertex in actualClique
            put neighbour in actualClique
            put all other vertices in verticesToTest
            maxClique ← exactVisit(adjacencyList, maxClique, actualClique, verticesToTest)
        end for
    end for
    return maxClique
end procedure

procedure EXACTVISIT(adjacencyList, maxClique, actualClique, verticesToTest)
    if verticesToTest is empty then                           ▷ recursion stop condition
        return maxClique
    end if
    for each vertex in verticesToTest do
        put vertex in actualClique
        if actualClique is a clique then
            if w(actualClique) > maxClique.weight then
                update(maxClique)
            end if
            remove vertex from verticesToTest
            maxClique ← exactVisit(adjacencyList, maxClique, actualClique, verticesToTest)
        end if
        remove vertex from actualClique
    end for
    return maxClique
end procedure
```

The goal of this algorithm is to browse all the existing set of vertices of a graph, and to return the one that is the clique of maximum weight. This algorithm browses a tree of solutions, where the root is a set of zero vertices, and more we go deep in the tree, more the sets of vertices are large. In the part ”exactMain”, we only start with edges that exist, because if we take a non-existing edge, we will never get a clique if we add additional vertices. Then, in ”exactVisit”, we call recursivity only if we get a clique (backtracking). This algorithm browses all the sets of vertices, but cuts off when a set of vertices is not a clique anymore (a same set of vertices with additional vertices will not be browsed). On the worst case (a complete graph), the tree of solutions is complete.

2.2 Time complexity

On the worst case, this algorithm browses all combinations of vertices of all sizes from 1 to n (2 to n in reality because cliques of one vertex have no weight). Also, to calculate the weight of a clique, it costs k (increment from the sum), because we need to add from the previous clique weight the weight of all edges from the vertices of the old clique to the vertex tested. So, time complexity is:

$$T(n) = \sum_{k=1}^{n} k \cdot \binom{n}{k} = \sum_{k=1}^{n} \frac{k \cdot n!}{k! \cdot (n-k)!}$$

$$T(n) = \sum_{k=1}^{n} \frac{(n-1)! \cdot n}{(k-1)! \cdot ((n-1)-(k-1))!} = n \cdot (\sum_{k=0}^{n} \binom{n-1}{k-1} \cdot 1^{k-1} \cdot 1^{(n-1)-(k-1)} - 1)$$

Binomial theorem $\rightarrow T(n) = n \cdot ((1+1)^n - 1) \in \boxed{O(n \cdot 2^n)}$

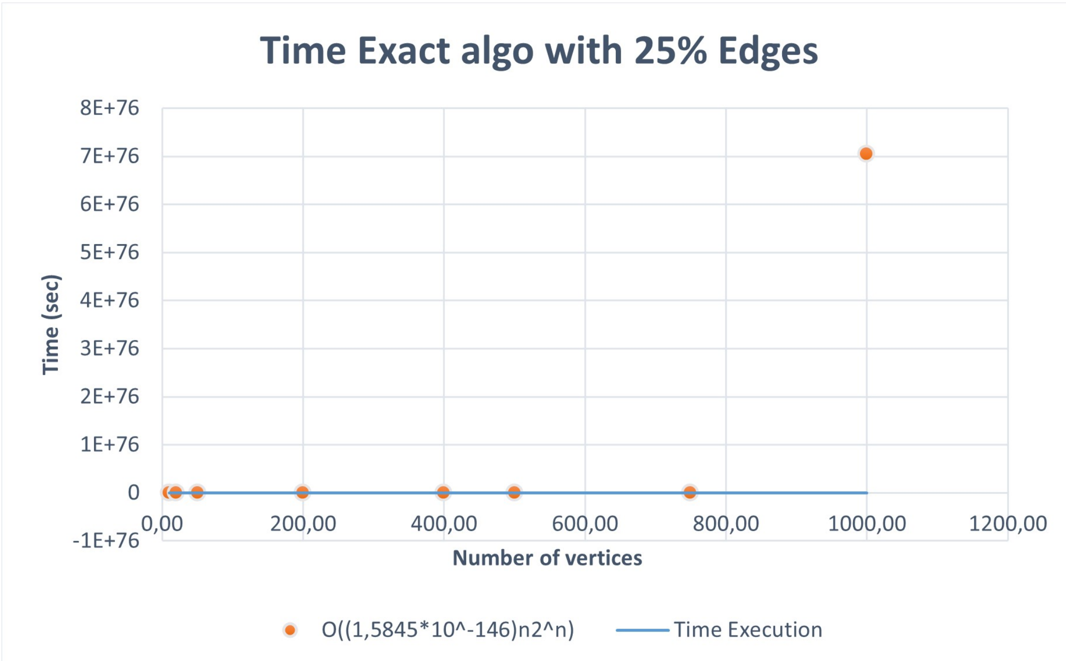
Therefore, worst-case time complexity of the exact algorithm is exponential.

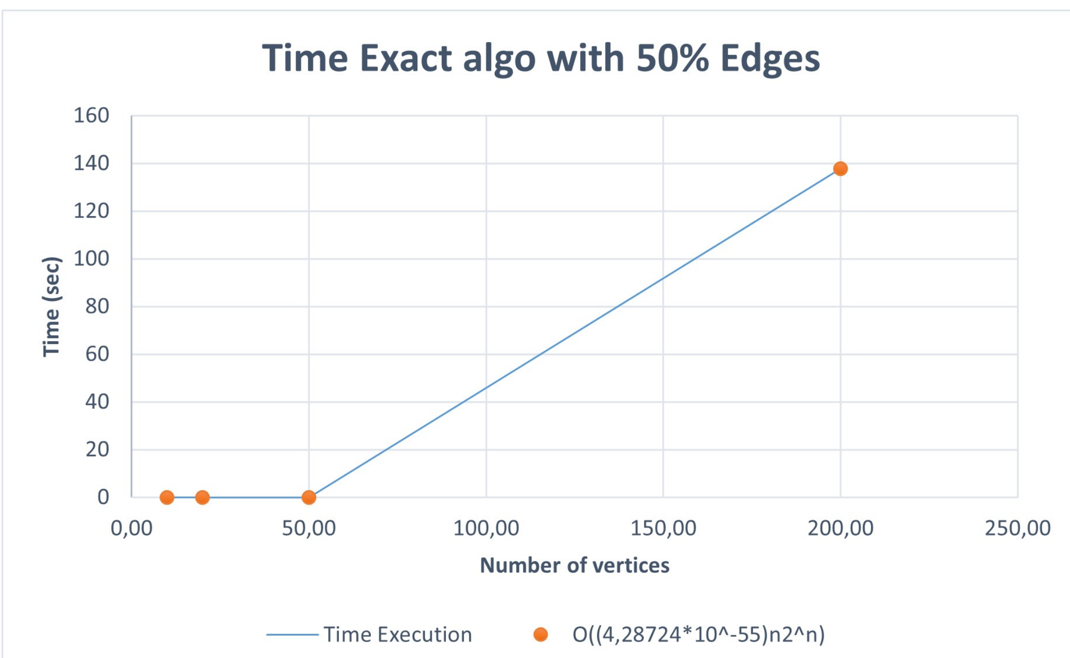
2.3 Instances

At the beginning, we did not improved the algorithm and tried all combinations in any case. We could run only small instances like ten and twenty vertices. Now, thanks to the backtracking improvement, we can run the 50-vertex instances in less than a minute. However, the 400-vertex instances are way too large - it would take days (for the 50% one) or years (for the 75% one) to get the exact solution.

2.4 Experiments

To explain this graph, the complexity in theory is greater than the practice and that's why we find this. In theory the graph represent the worst case, then, the complexity is really high compared to practice.


Time Exact algo with 25% Edges
8E+76
7E+76
6E+76
5E+76
4E+76
3E+76
2E+76
1E+76
0
-1E+76
0,00
200,00
400,00
600,00
800,00
1000,00
1200,00
Time (sec)
Number of vertices
O((1,5845*10^-146)n2^n)
Time Execution


Time Exact algo with 50% Edges
160
140
120
100
80
60
40
20
0
0,00
50,00
100,00
150,00
200,00
250,00
Time (sec)
Number of vertices
Time Execution
O((4,28724*10^-55)n2^n)

3 Constructive heuristic algorithm

3.1 Pseudo-code

Algorithm 2 Find a maximum weight clique of a graph easily. It could be not the best one

```
procedure CONSTRUCTIVEHEURISTICMAIN(adjacencyList, &tableauResult, size, &weight)
    for each vertex in adjacencyList do
        if vertex.degree >= start then
            tabIndexActual ← vertex
        end if
    end for
    for each indexActual in tabIndexActual do
        while true do
            if indexActual is not in tableauResult then        ▷ if it in, we break and while stop
                tableauResult.Insert(tableauResult.begin, indexActual)
            end if
            for each neightbor of indexActual do
                if neighbor.distance > max then        ▷ max initialize at 0 for first iteration
                    indexMax ← neighbor.index
                    max ← neighbor.distance
                end if
            end for
            if indexMax is in tableauResult then
                for each neightbor of indexActual do
                    if neightbor == indexActual then
                        neightbor.weight ← 0
                    end if
                end for
            end if
            for each indexActual in tableauResult do
                for each neightbor of indexActual do
                    if neighbor is in tableauResult&& is not in tableauUsed then
                        weight ← weight + neightbor.weight
                    end if
                end for
                tableauUsed.Insert(tableauUsed.begin(), indexActual)
            end for
        end while
        we get the best weight of clique and we return them in &weight and &tableauResult
    end for
end procedure
```

Note: this algorithm has been **simplified** to facilitate its comprehension, it actually needs more variables to run properly with programming languages.

The main goal of this algorithm is to return a pretty good solution. This solution may not be the best one but it finds it faster than exact algorithm. We start with a vertex who has max neighbors of all vertices in the graph. We check its neighbors and we take the one who have max weight. We continue until we find a vertex already saw. After this, we get a clique and its weight.

3.2 Time complexity

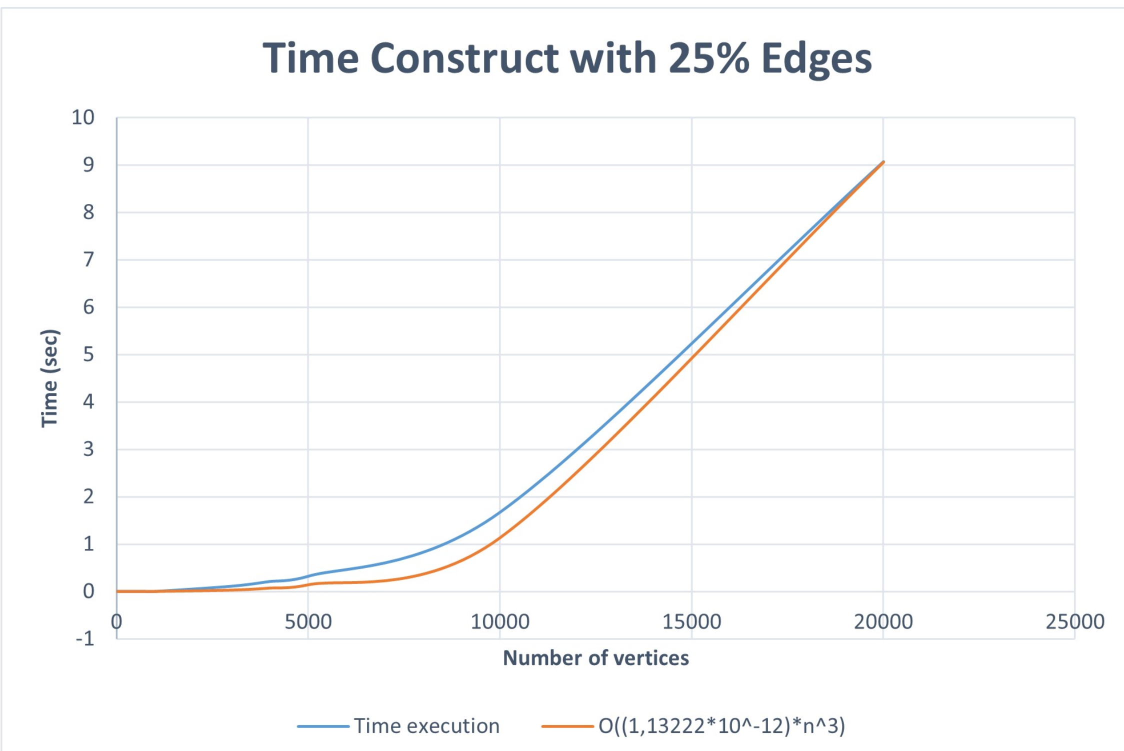
On the worst case, the graph is complete, it means that all vertices have maximum neighbors. The algorithm will run to check all links with other vertices and take the first one with maximum weight. The algorithm will continue until he get a clique. If we have many vertices with maximum neighbors, algorithm will test all vertices and select the best clique. It will be the best for a complete graph but in majority for a not complete graph, the clique will be a good clique but not the best. So, time complexity is:

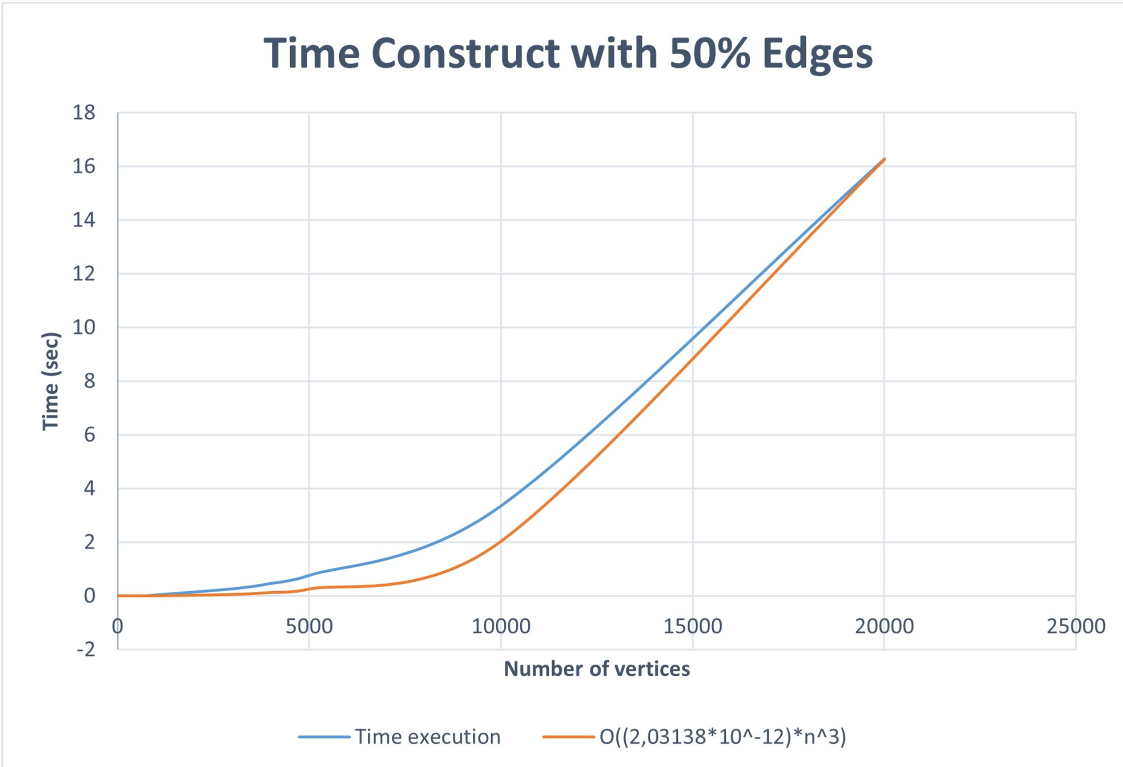
$$T(n) = n + n * ((n-1) + (n-1) + (n * (n-1))) \in \boxed{O(n^3)}$$

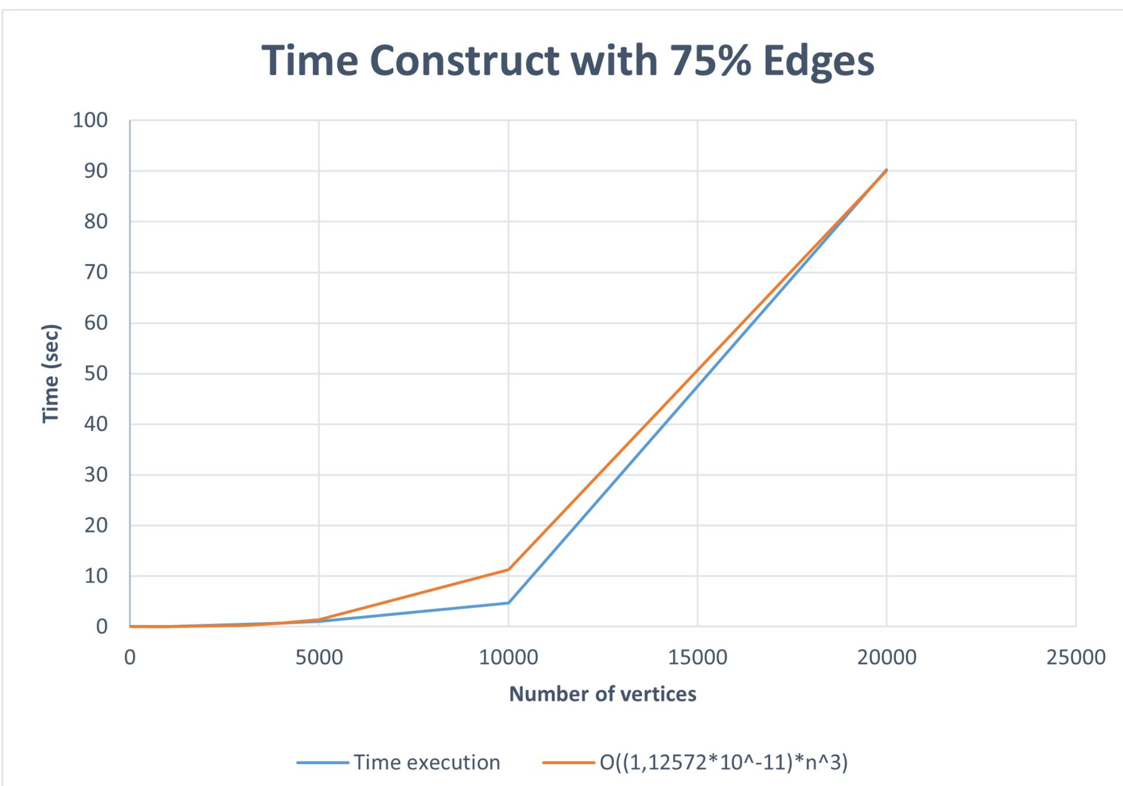
3.3 Instances

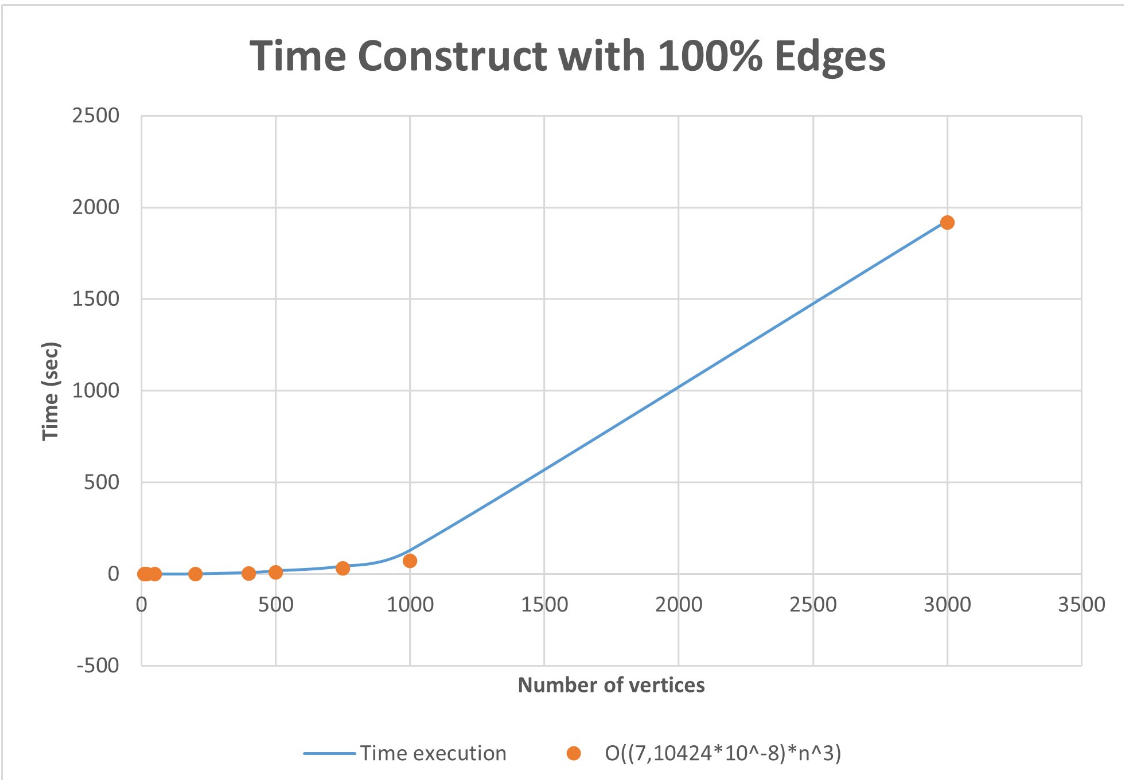
At the beginning, the first version of this algorithm takes the first vertices who have max of neighbors. It means that if another vertex has the same number of vertices, it will not be tested. Now, with the second version we test all vertices who have max of neighbors and we see which clique that has the max weight and we return the clique and its weight. For instances under 5,000 vertices the time is really short (approximately 0 seconds) and when we have more than 5,000 vertices we start to have more time execution but it is still really fast. The percentage has a role in how much time will run our algorithm. If the percentage of edges is 100%, all vertices have max neighbors, so the time execution will be really long because it will test all vertices.

3.4 Experiments




Time Construct with 50% Edges
Time (sec)
18
16
14
12
10
8
6
4
2
0
-2
0
5000
10000
15000
20000
25000
Number of vertices
Time execution
O((2,03138*10^-12)*n^3)


Time Construct with 75% Edges
Time (sec)
100
90
80
70
60
50
40
30
20
10
0
0
5000
10000
15000
20000
25000
Number of vertices
Time execution
O((1,12572*10^-11)*n^3)



Time Construct with 100% Edges
Time (sec)
2500
2000
1500
1000
500
0
-500
0
500
1000
1500
2000
2500
3000
3500
Number of vertices
Time execution
O((7,10424*10^-8)*n^3)

4 Local search heuristic algorithm

4.1 Pseudo-code

Algorithm 3 Find an approximated maximum weight clique using local search heuristic

Note: this algorithm has been **simplified** to facilitate its comprehension, it actually needs more variables to run properly with programming languages.

```
procedure localMain(adjacencyList, maxClique)
    for increment in n vertices of adjacencyList do
        maxClique ← localVisit(adjacencyList, maxClique)
        if maxClique length has not increased then        ▷ a local maximum is reached
            return maxClique
        end if
    end for
    return maxClique
end procedure

procedure localVisit(adjacencyList, maxClique)
    tempClique ← empty array of vertices
    verticesToTest ← empty array of vertices
    copy maxClique.array to tempClique
    put all vertices that are not in maxClique.array in verticesToTest
    for each vertex in tempClique do
        remove vertex from tempClique
        for each vertexTest1 in verticesToTest do
            put vertexTest1 in tempClique
            if tempClique is still a clique then
                for each vertexTest2 greater than vertexTest1 in verticesToTest do
                    put vertexTest2 in tempClique
                    if tempClique is still a clique and w(tempClique) > maxClique.weight then
                        update(maxClique)
                    end if
                    remove vertexTest2 from tempClique
                end for
            end if
            remove vertexTest1 from tempClique
        end for
    end for
    return maxClique
end procedure
```

The main goal of this algorithm is to return a pretty good solution that may not be the best one, but using a better complexity than the exact algorithm. We start with a solution, then we remove a vertex of this solution, to finally add two more that are not in the solution. The algorithm verifies if the first vertex we add to the solution creates a clique, in the other case we do not continue adding a new vertex, because it will not be a clique anymore. At the end of "localVisit", the function should return a solution that has one more vertex in it. In the other case, we have reached a local maximum, that means that the algorithm can no longer find a better solution, so it stops.

4.2 Time complexity

On the worst case, the graph is complete, that means that if we start with a solution of two vertices, we will need to call "localVisit" $n-2$ times to get a local maximum, that is here the global maximum. Then, in "localVisit", the algorithm browses all the vertices of the maximum clique, to remove them. We will name the size of this clique k. Then, the algorithm takes two vertices to add among the $n-k$ remaining. Finally, to calculate the new clique weight, we remove the weight caused by the removed vertex and we add the weight caused by the two new vertices on the clique, and that costs approximately $3k$ in complexity. So, time complexity is:

$$T(n) \approx \sum_{k=1}^{n} k \cdot \binom{n-k}{2} \cdot 3k = \sum_{k=1}^{n} 3k^2 \cdot \frac{(n-k)!}{2! \cdot (n-k-2)!}$$

$$T(n) = \sum_{k=1}^{n} 3k^2 \cdot \frac{(n-k) \cdot (n-k-1)}{2}.$$

Considering that $n > k$:

$$T(n) \approx n^2 \cdot \sum_{k=1}^{n} k^2 = n^2 \cdot \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

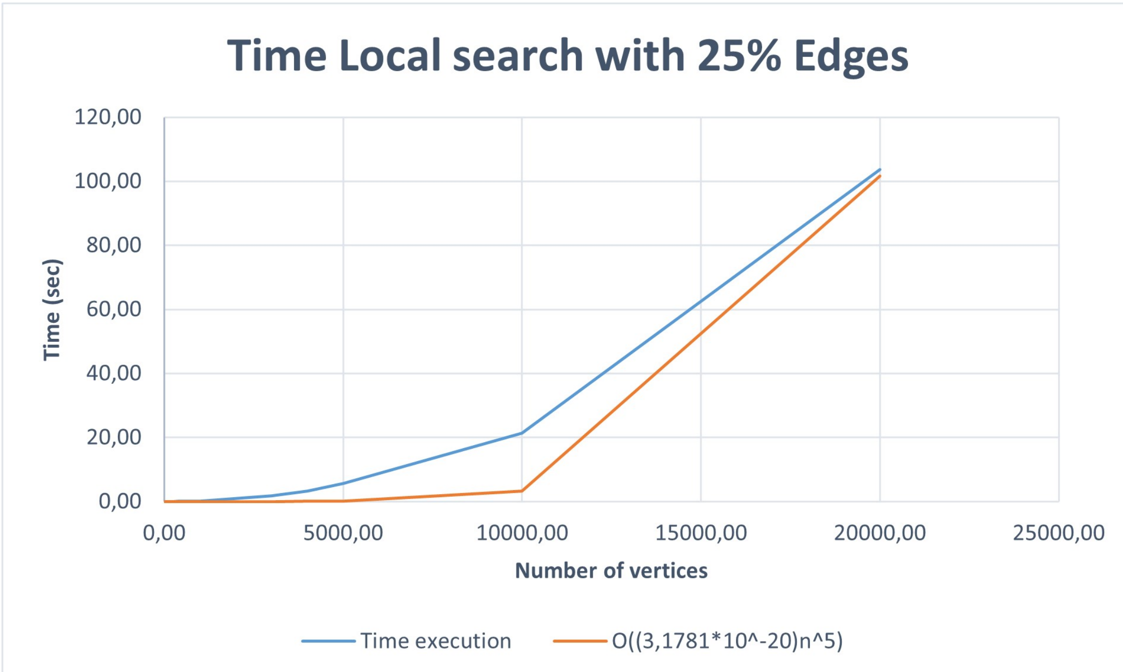
Therefore, $\boxed{T(n) \in O(n^5)}$

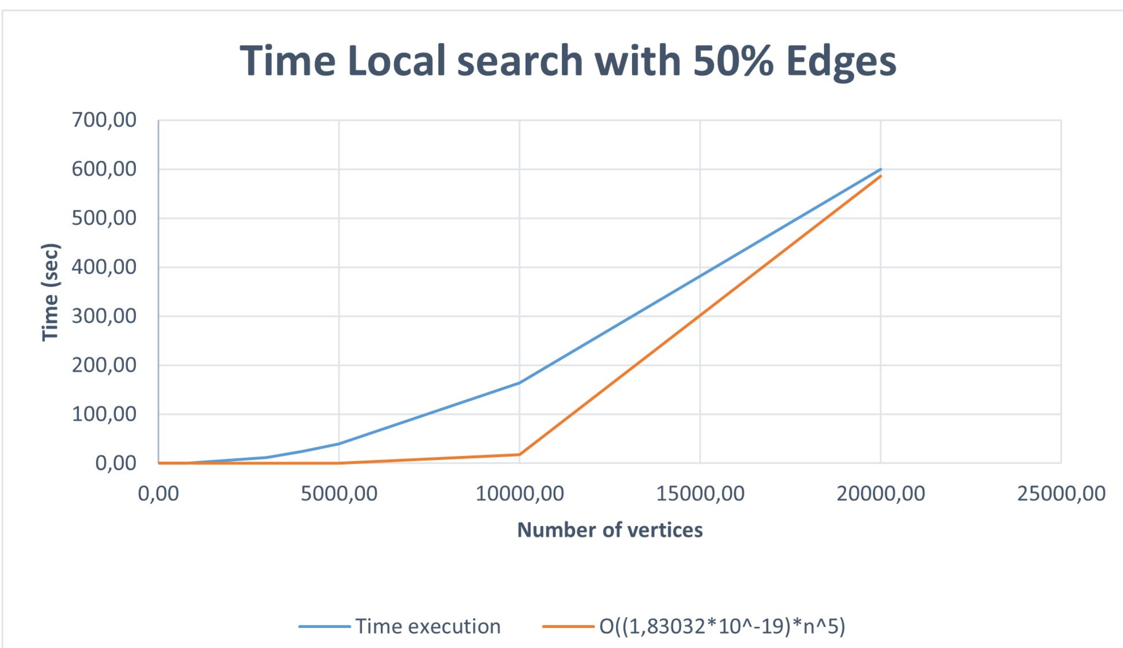
4.3 Instances

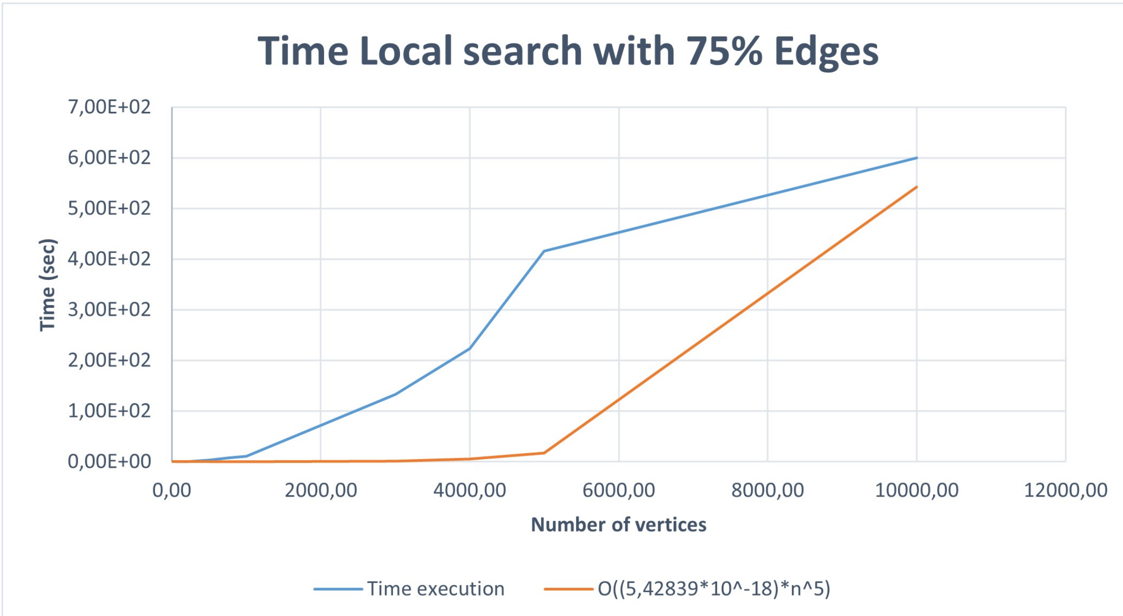
At the beginning, we were testing all combinations of two vertices in any case, even if one of the added vertex did not create a clique. Then, we added an improvement that tests if a vertex creates a clique before adding it in the actual solution (like the backtracking improvement in the exact algorithm). Before making this improvement, we added two conditions to prevent the algorithm to run indefinitely. The first one was a clock which was starting at the beginning of the algorithm, and for each new set of vertices tested, the clock value was compared to the max value. If the max value - set by the user - was exceeded, we stop the program. The second stop condition was if we start with a graph with too much edges, we only keep the first $k+1$ solution returned by the algorithm each time. We had set the maximum value of clock to ten minutes and the edge condition at 50,000, and some of the instances were stopped thanks to the conditions. Then, after making the "backtracking" improvement, we took the largest instance from the first set of instances - which is 20,000 with 25 percent of edges - and it was now running in less than two minutes, without the edge condition. That means that the clock condition is now also useless in that case. Therefore, we removed the two stop conditions from the algorithm.

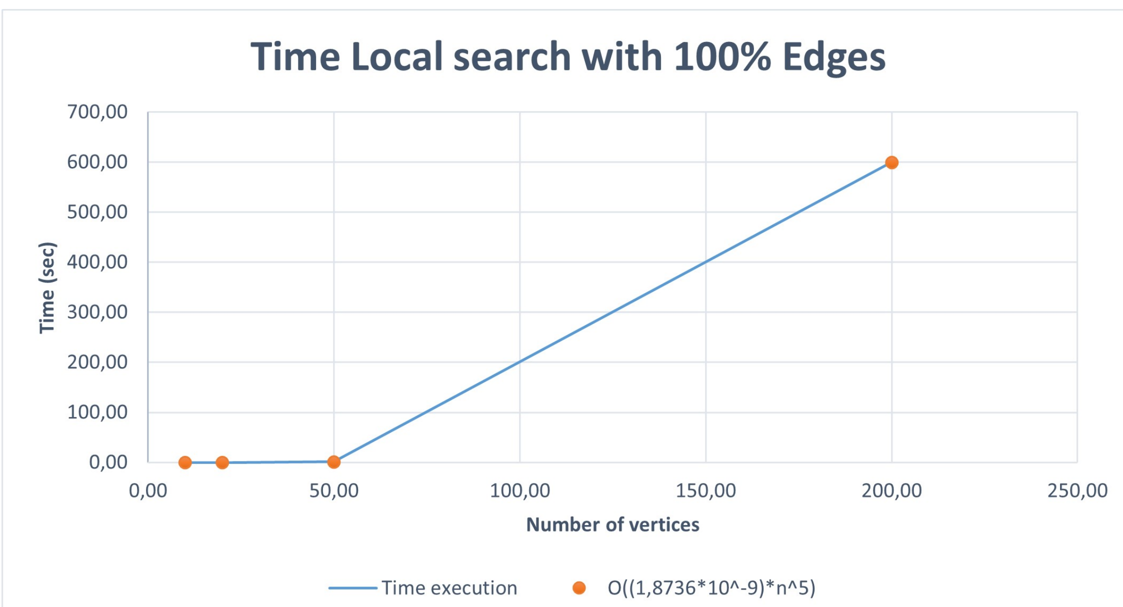
4.4 Experiments

Our input solution is the edge that has the most of incident edges.


Time Local search with 25% Edges
120,00
100,00
80,00
60,00
40,00
20,00
0,00
Time (sec)
0,00
5000,00
10000,00
15000,00
20000,00
25000,00
Number of vertices
Time execution
O((3,1781*10^-20)n^5)


Time Local search with 50% Edges
700,00
600,00
500,00
400,00
300,00
200,00
100,00
0,00
Time (sec)
0,00
5000,00
10000,00
15000,00
20000,00
25000,00
Number of vertices
Time execution
O((1,83032*10^-19)*n^5)


Time Local search with 75% Edges
7,00E+02
6,00E+02
5,00E+02
4,00E+02
3,00E+02
2,00E+02
1,00E+02
0,00E+00
Time (sec)
0,00
2000,00
4000,00
6000,00
8000,00
10000,00
12000,00
Number of vertices
Time execution
O((5,42839*10^-18)*n^5)


Time Local search with 100% Edges
700,00
600,00
500,00
400,00
300,00
200,00
100,00
0,00
Time (sec)
0,00
50,00
100,00
150,00
200,00
250,00
Number of vertices
Time execution
O((1,8736*10^-9)*n^5)

5 Tabu search meta-heuristic algorithm

5.1 Pseudo-code

Algorithm 4 Find a better maximum weight than the local search algorithm, by tabu-ing the local maximums found before

Note: localMainModified() is almost the same function than localMain() seen before, but it **does not visit** vertices that are in the tabu list given in argument.

procedure TABUSEARCH(*adjacencyList*, *maxClique*)
  tabuList ← empty array of vertices
  testClique ← empty structure with weight, array of vertices and size of array
  maxClique ← localMainModified(*adjacencyList*, *maxClique*, *tabuList*)
  put all vertices of maxClique in tabuList
  while *tabuList* does not contain all vertices that create edges **do**
    clear *testClique*
    testClique ← edge not present in tabuList
    testClique ← localMainModified(*adjacencyList*, *testClique*, *tabuList*)
    put all vertices of testClique in tabuList
    if *testClique.weight* > *maxClique.weight* **then**
      update *maxClique*
    end if
  end while
  return *maxClique*
end procedure

The main goal of this algorithm is to browse all vertices with local search to find a global maximum, which is not the exact solution, but a same or better one than the local search.

5.2 Time complexity

For the Tabu search, we use the local search algorithm. We will calculate the worst case in two situations, we know the local search algorithm worst case :

$$T(n) \approx n^2 \cdot \sum_{k=1}^{n} k^2 = n^2 \cdot \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Therefore, $\boxed{T(n) \in O(n^5)}$

We are looking at if we can have a worst case with the Tabu search algorithm. In this case we imagine having a graph with subgraphs of degree 1 for each vertex. The time complexity is :

$$T(n) \approx \frac{n}{2} \cdot \binom{n-2}{2} = \frac{n}{2} \cdot \frac{(n-2)!}{2!(n-4)!} = \frac{n}{2} \cdot \frac{(n-2)(n-3)(n-4)!}{2!(n-4)!} = \frac{n(n-2)(n-3)}{4}$$

Therefore, $\boxed{T(n) \in O(n^3)}$

We can conclude that the complexity is:

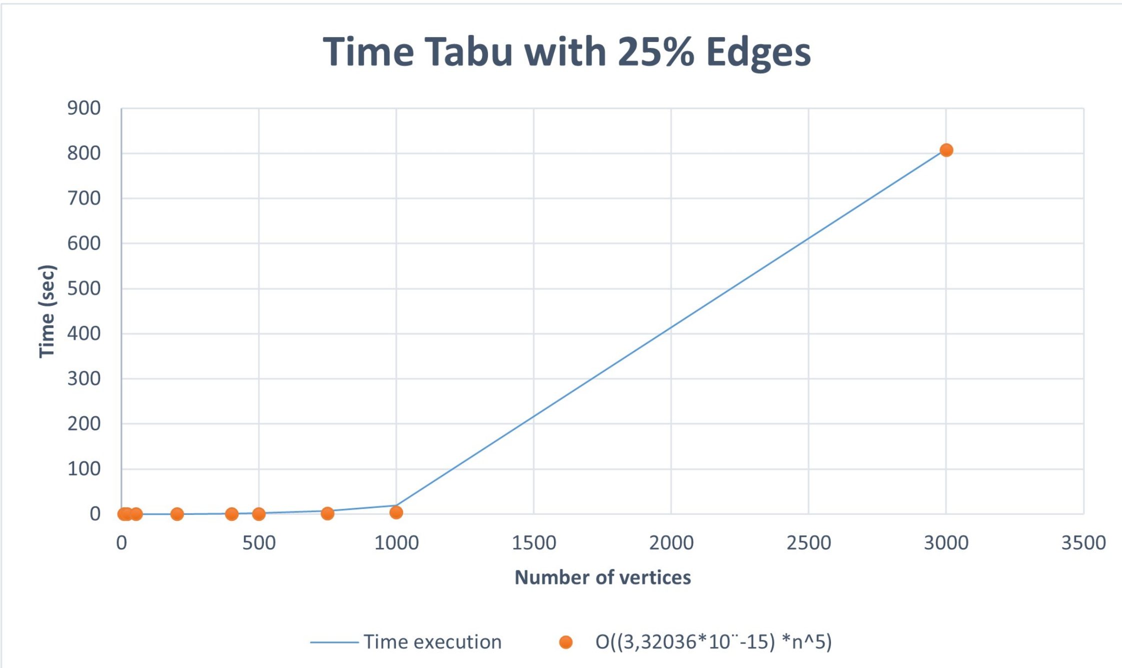
$$\boxed{T(n) \in O(n^5)}$$

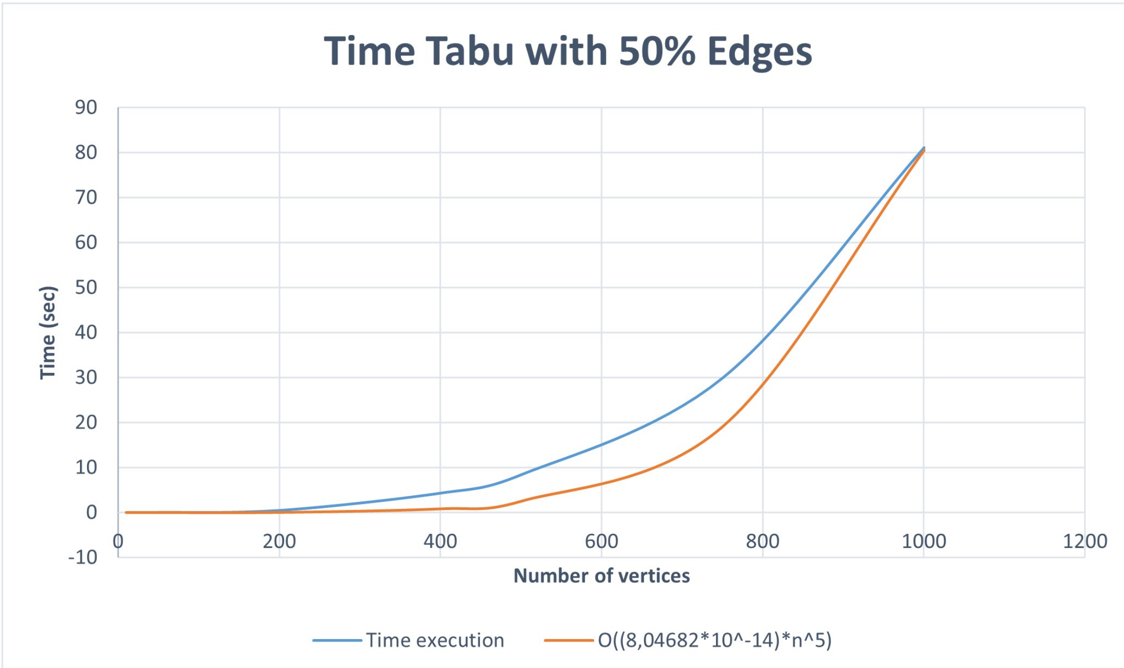
5.3 Instances

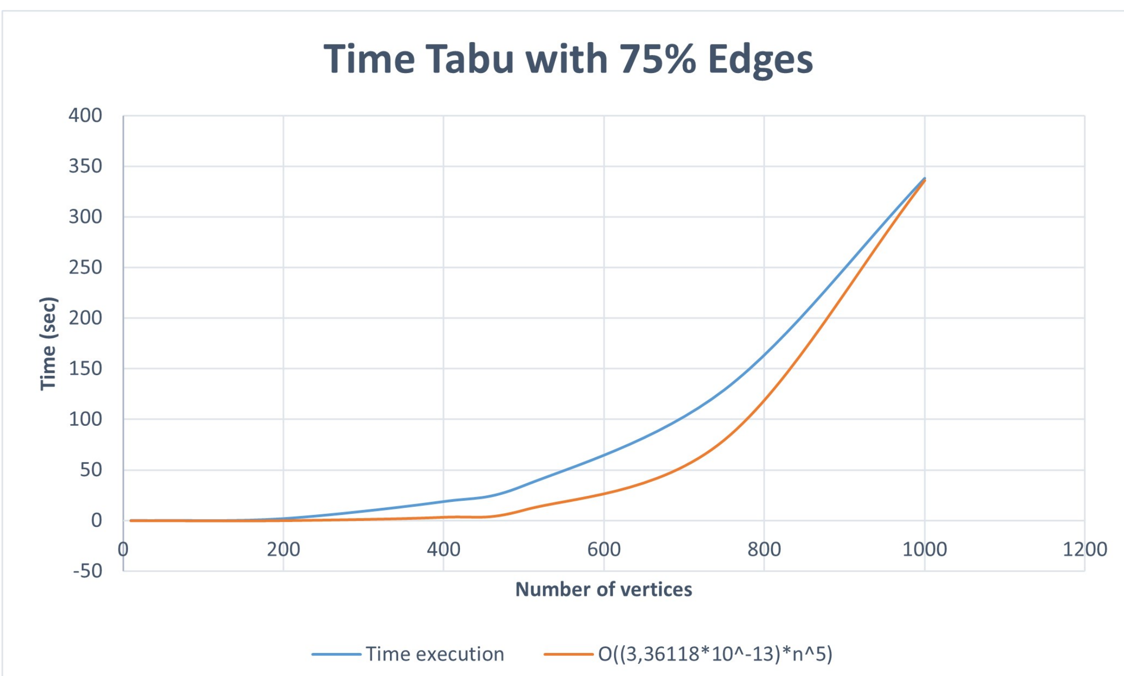
This algorithm will be fast if the input graph does not contain large cliques (with not a lot of vertices for each clique). Then, less the input graph has edges, more the Tabu solution will be near than the local one, because the graph has less chances to have multiple cliques of high length.

5.4 Experiments

Like the local search algorithm, we take in input the edge that has the most of incident edges.




Time Tabu with 50% Edges
Time (sec)
90
80
70
60
50
40
30
20
10
0
-10
0
200
400
600
800
1000
1200
Number of vertices
Time execution
O((8,04682*10^-14)*n^5)


Time Tabu with 75% Edges
Time (sec)
400
350
300
250
200
150
100
50
0
-50
0
200
400
600
800
1000
1200
Number of vertices
Time execution
O((3,36118*10^-13)*n^5)

6 Conclusion

At first, for time complexity, construct heuristic is the fastest one, it means that - comparing to other algorithms - it is the most efficient one to compute large graphs. However, it is not the most accurate one. Then, local search algorithm is better to find the maximum local from same input solution, contrary to constructive. Then, Tabu search browses all vertices of the graph to return the global maximum, which is more accurate than the precedent ones. And finally, exact algorithm browses all combinations of vertices of the graph, so it is the less efficient one, but the most exact one - hence its name.

We can see if we compare theoretical time complexity and practice complexity on graph than the curves are really close. We can think than our algorithms are pretty fine for the different graph.