GRAPH THEORY

Final Project

The Maximum Edge Weight Clique Problem

ANSELMET Eloi - AUJEAU Laurine - BOURREAU Hugo - GROSS Charles-Clément - LANDAIS Pierre

Summary

Introduction

Our project consists in building algorithms that solve the maximum edge weight clique problem (MEWCP). But what is a MEWCP? Given a simple undirected graph G = (V, E) and non-negative weight for each edge, the maximum edge-weight clique problem (MEWCP) is to find the clique of maximum weight. It is NP-Hard, which can be translated into one for solving any NP-problem (nondeterministic polynomial time) problem. But how are we going to solve this problem now ? First we need to study four algorithm for our problem :

1. The exact algorithm
2. The constructive heuristic algorithm
3. The local search heuristic algorithm
4. The tabu search meta-heuristic algorithm

All algorithms are built differently but the last three are connected, you will see below.

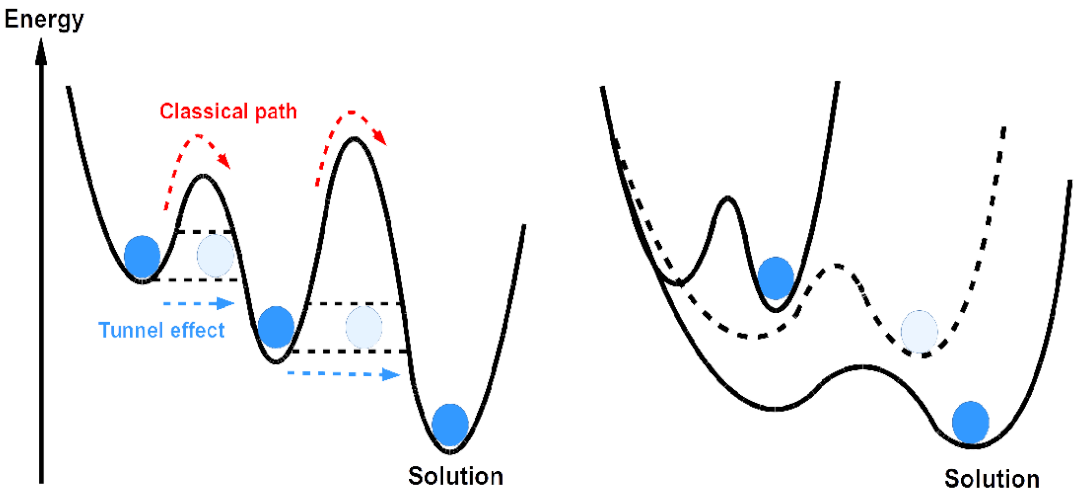
Purpose of this problem (Q1)

The clique problem has many applications such as :

- Quantum Computing
- Computational Chemistry
- Team Collaboration
- Metallurgy

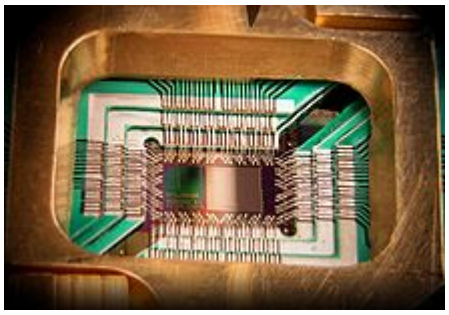
What is quantum annealer? It is a meta-procedure for finding a procedure that finds an absolute minimum size or length or cost or distance from within a possibly very large, but nonetheless finite set of possible solutions using quantum fluctuation-based computation instead of classical computation.

Recently, quantum annealer is studied to solve NP-hard problems including MCP. Quantum annealer can solve the quadratic unconstrained binary optimization (QUBO) problem. Since quantum annealer solvers are heuristic, efficient exact solvers are required to evaluate them. QUBO can be formulated as MEWCP by the vertex-and-edge-weighted complete graphs where negative weight is allowed. Hence handling negative weight is one future work. Nodes in V as qubits represent problem variables with programmable weights, and edges as couplers in E have programmable connection strengths.

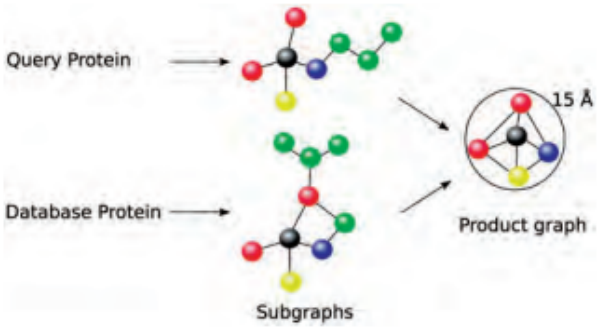


Quantum Tunnelling **Adiabatic evolution**

Thanks to the studies of quantum annealer, companies like D-Wave Systems are building some of the best chip on the quantum computing market !



The algorithms for finding a maximum clique are also frequently used in chemical information where their main application is to search for similarity between molecules. These algorithms are used for screening databases of compounds to filter out molecules that are similar to known biologically active molecules and are feasible to be active themselves. Also, these algorithms are used for comparing protein structures, to provide the information about protein function and also the information about possible interactions between proteins. Every vertex is a group of acid or amine. And every edge is equivalent to peptide bond.



Team collaboration :

Let's imagine that we want to build the biggest team with the better relationships between members. Vertices represent members and edges represent compatibility between members. The compatibility is good between 2 members when the weight of the edge is high. If the weight of an edge is equal to 0, we suppose that they don't know each other and they can't collaborate together. We want to get the biggest team where everyone knows the other members, and with the better cohesion between them.

Metallurgy :

Let’s imagine that we want to create the biggest alloy with the better yield. Vertices represent metals and edges represent compatibility between metals. If the compatibility is good between 2 metals, it means that the weight is high and that we can produce an alloy with these 2 metals with a good yield, without waste metals during chemical transformation. If the weight of an edge is equal to 0, we suppose that they don’t react together and can’t create an alloy. We want to get the biggest alloy with the biggest number of metals with the better yield.

Exact Algorithm (Q2)

Definition

The objective of this algorithm is to parse all graphs and try every combination of vertices to find the maximum edge weight clique. When this algorithm is executed we expect to have the correct answer and not an approximation like other algorithms.

Pseudo-code

The proposed algorithm is made in a recursive way.
In this algorithm we use it as a method of a class, and this class have this attributes we can use in every call they are shown from line 1 to 5 :

Algorithm 1 Exact algorithm

```
1:  graph ← matrix of vertices with the weight between edges
2:  store ← Array of n positions (to save current clique)
3:  degres ← Array of n positions (with the degree of every vertex)
4:  n ← Number of vertices
5:  maxClique ← StoretheMEWC
6:
7:  function EXACT(previousVertex, cliqueSize)
8:      foundBiggerClique ← False
9:      for v from previousVertex + 1 to n do
10:         if degre[v] >= cliqueSize and v connexe with previousVertex then
11:             store[cliqueSize] ← v
12:             if isClique(store) then
13:                 foundBiggerClique ← True
14:                 if not exact(v, cliqueSize + 1) then
15:                     if w(store) > w(maxClique) then
16:                         maxClique ← store
17:     return foundBiggerClique
```

We call this algorithm like this : exact(-1, 0)
Our first parameter means there is no previous vertex (only first call) and the clique size is 0 for now.In the for loop we will try to add every remaining vertices to the clique. The “if” line 10 means we add the vertex to the current clique only if its degree is sufficient and it is connexe with the previous vertex. If so, we add it to the current clique and we execute the function “isClique” which returns true if it is indeed a clique and false if it is not.
If we have a clique, we call this function again but with the remaining vertices (the same vertex can’t be two times in the same clique) and size of current clique plus one.

The objective of the variable “*foundBiggerClique*” is to go into the if line 14 only if we have the max size clique with the current vertices. For example if 1, 2 and 3 are connexe we don’t want to calculate clique [1,2], [1,3], [2,3] and [1,2,3]. We know that the weight of [1,2,3] is the best one because it has the same edges and some other

so the weight will be bigger. So if we can't add a vertex, the function return false and we know we have a clique of max size for the current vertex and now we can calculate its weight and see if it's bigger than the best we have

Complexity

To find every clique of size k we have to try every combinaisons of k size so we have this :

$$\binom{n}{k} = C_n^k$$

Now we know how to find every clique of k size, but the clique we want could be of any size so we will make k vary from 1 to n (for real k will start from 2 because to find a maximal clique we need at least 2 vertices). To find the theoretical complexity we need to solve this :

$$\sum_{k=1}^{n} k.C_n^k = \sum_{k=1}^{n} k\binom{n}{k}$$

So we have this :

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

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

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

$$\sum_{k=1}^{n} k\binom{n}{k} = n\sum_{k=1}^{n-1} \binom{n-1}{k-1}$$

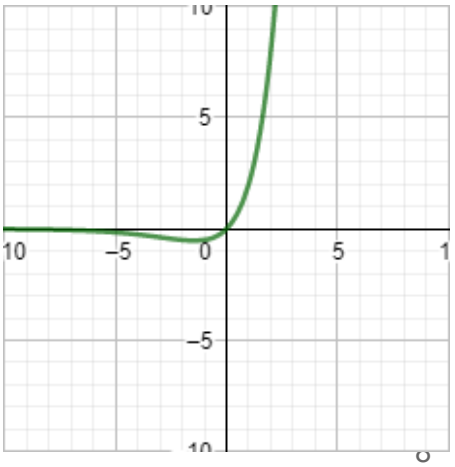
$$\sum_{k=1}^{n} k\binom{n}{k} = n\sum_{k=1}^{n-1} \binom{n-1}{k-1}.1^{k-1}.1^{n-1-k-1}$$

$$\sum_{k=1}^{n} k\binom{n}{k} = n\sum_{k=0}^{n-1} \binom{n-1}{k}.1^{k}.1^{n-1-k}$$

$$\sum_{k=1}^{n} k\binom{n}{k} = n.(1+1)^{n-1}$$

$$\sum_{k=1}^{n} k\binom{n}{k} = n.2^{n-1}$$

We finally find a complexity of **O(n.2^n)**

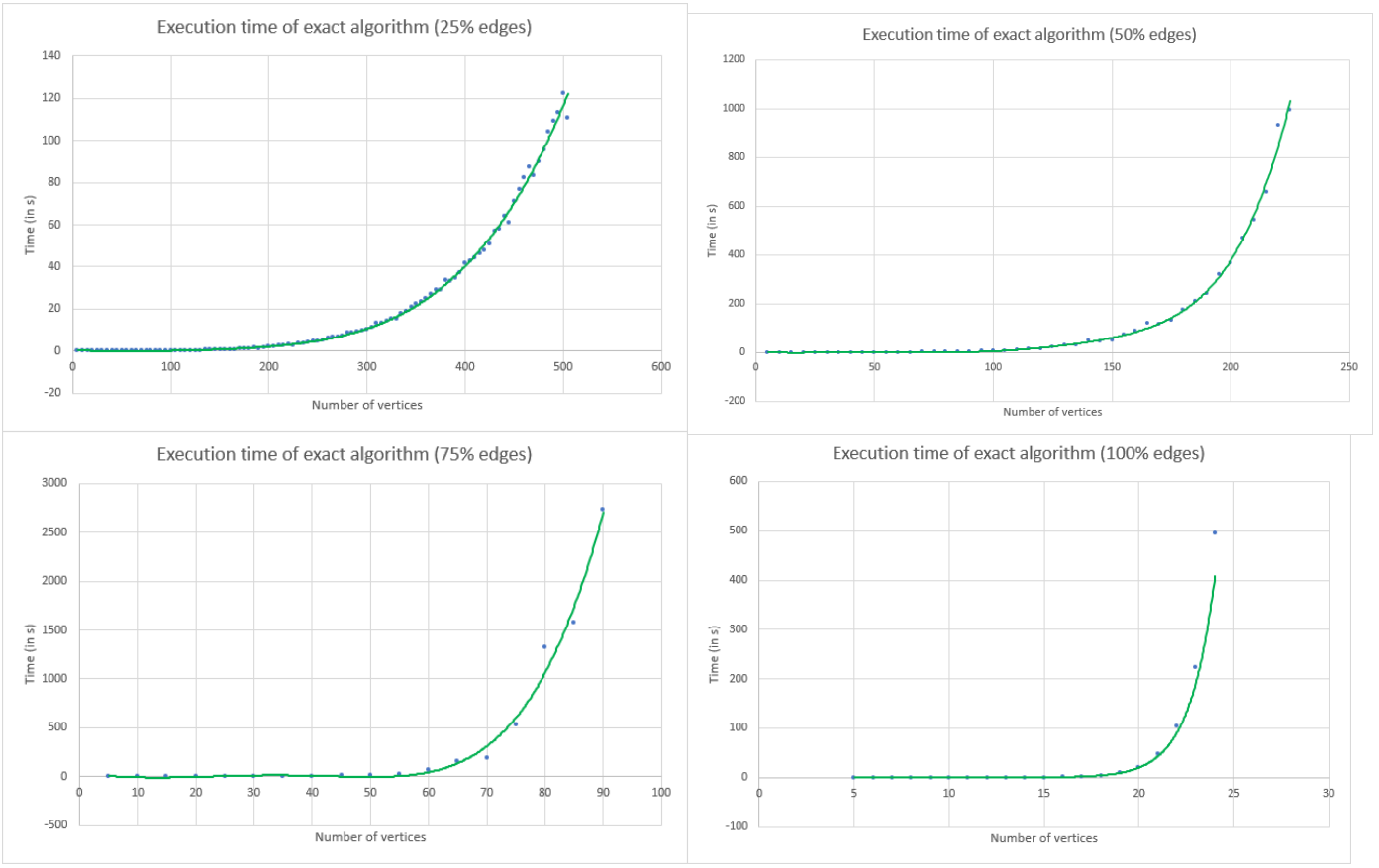


Best and Worst cases

In this algorithm, the more there are edges, the more there are potential cliques. There are some improvements in the code to avoid losing time in the loop. We know we will not have better results. But in general there are more cases to try.

In a graph with n vertices and m edges. The best case for execution time is if the cliques are the smallest possible, edges are separated all over the graph. We will not have big cliques and not a lot of recursives calls. However, bigger the cliques will be, more recursives calls we will have and longer it will take.

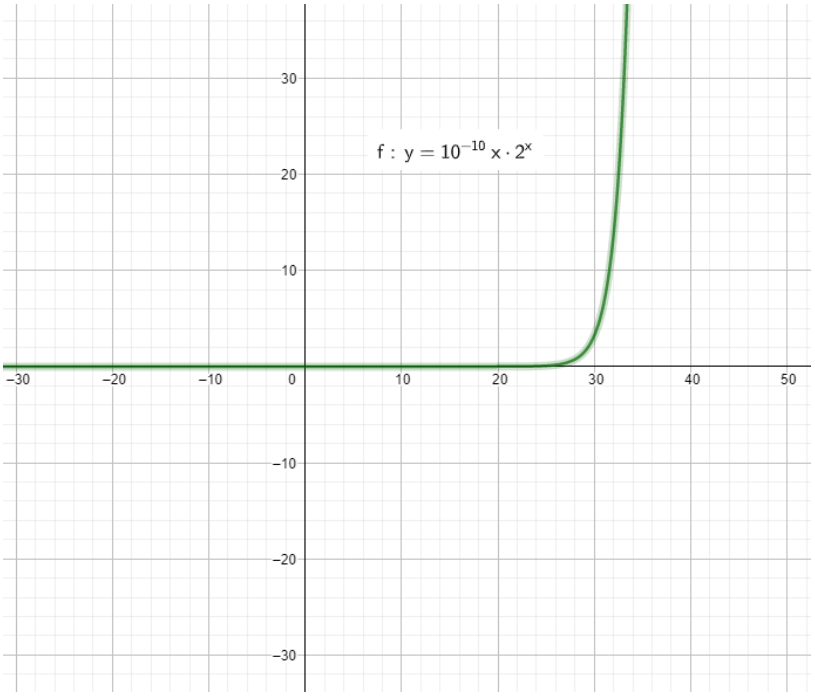
Experimental part



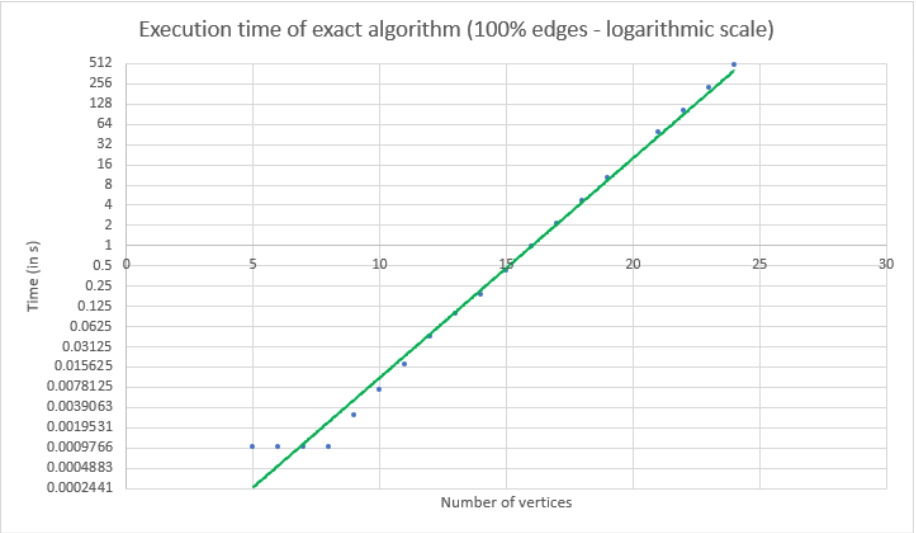
To obtain these graphes we ran four sets of instances with 25, 50, 75 and 100% of edges. We can see more edges there are for the same amount of vertices more it takes time. Less further we can go in the number of vertices because of execution time.

The shapes of the curves we obtain look the same.

With the theoretical complexity we calculated before we add a constant and the curve looks like the 4 others. By adjusting the constant we can find curves that have the same shape between theoretical and experimental.



This graphic is one of the previous curves on a logarithmic scale. We can see all points are pretty lined up except the first one. Firsts one are approximative because execution time was very fast and the mesure can't go under a certain time.



Let's find the complexity on logarithmic scale :

$$\log_2(n.2^n) = \log_2(n) + \log_2(2^n)$$
$$\log_2(n.2^n) = n + \log_2(n)$$
$$\log_2(n.2^n) \approx n$$

It’s coherent with the curve we obtain because our curve is a linear function type y = an + b
Where b is a coefficient in front of $n.2^n$ on a logarithm scale and n is the leading coefficient. So we have y = n + b. The curve match the theorical complexity.

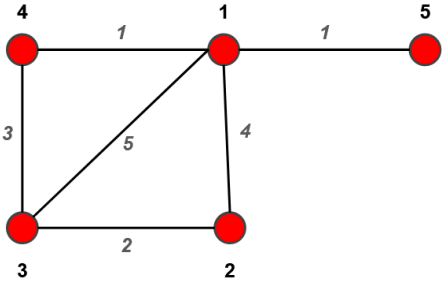
Constructive Heuristic Algorithm (Q3)

Definition

An heuristic algorithm is not the best one : it is not perfect and it can be nonoptimal. It's just a good way to get an approximation. [link]

A constructive heuristic algorithm is a heuristic algorithm which starts with an empty solution and in each iteration, the algorithm builds step by step his solution. It is not an algorithm which starts with a solution and tries to improve it at each iteration. [link]

A quick example



Let's start with this graph. The best result that we can have is the clique {1,2,3} with a weight of 11.

Step 1 : First, we search the starting vertex which is the vertex with the most neighbors. It is obviously the vertex 1. We will add it to our final clique. Now, we "delete" all vertices which are not connected to 1 to avoid checking them later.

Step 2 : Now, we will search the most weighted edge starting from 1 : it is the edge {1/3}. Then we add 3 to the clique. Now we add to the total weight of the clique each edge between the picked vertex (here 3) and vertices in the clique (here 1). Like in step 1, we delete all vertices which are not connected to 3 : in this case we delete the vertex 5.

Step 3 : Repeat the step 2 until there are no more vertices which are not "deleted" or in the clique. In our case we choose the vertex 4 instead of the vertex 2 and we delete the vertex 2. We add to the total weight the weight of edges {4/3} and {4/1}. It is the end of the graph.

Result : Then we get a final clique {1,3,4} of weight 9. It is not the better one but it is not the purpose of this algorithm. With this one, we know the maximum weight can be more than 9 or is equal to 9.

Pseudo Code

NB : In our case, I only put G in parameter because it is an object and it contains :

- nbEdges // Number of edges
- nbVerticies // Number of vertices
- My matrix/list which represent vertices and edges
- availableVertices <- list() // Vertices which are eligible to be in the clique

- chosenVertices <- list() // Vertices which are in the clique
- totalWeight <- 0 // Total weight of my clique

Algorithm 2 Constructive Algorithm

```
1: procedure CONSTRUCTIVE ALGORITHM(G)
2:     startingVertex                                  ▷ Vertex with the most neighbors
3:     chosenVerticies ← chosenVertices U startingVertex
4:     availableVerticies ← availableVertices − {startingVertex}
5:     Throw from availableVerticies verticies which don't share an edge with startingVertex
6:     currentVertex = startingVertex
7:     while availableVertices ≠ null do
8:         currentVertex   ▷ Get the most weighted edge which links currentVertex and a vertex in availableVerticies and take the
                                                                                  associated other vertex
9:         Update the weight of the clique by adding edge's weight between currentVertex and each vertex in chosenVertices
10:        chosenVerticies ← chosenVertices U currentVertex
11:        availableVerticies ← availableVertices − {currentVertex}
12:        Throw from availableVerticies verticies which don't share an edge with currentVertex
13:    return G
```

Complexity of this pseudo-algorithm : $O(n^3)$

Best and Worst Cases

A constructive algorithm checks only the weight between a vertex and his neighbors, not further. Therefore, the result of the algorithm will only be affected by the direct neighbors (and not by branching, cycles, pendant vertices…). The number of edges and vertices doesn't change the quality of the output file.

Experimental Part

To get our execution times, we did the average between 3 measures for each instance. For lists, we are capped by a long time of graph creation. For matrices, we can go further but we had a lack of time.

We can see that the number of edges (25/50/75 % of n) has an impact on the execution time and the biggest instance that we can load. The list version is more impacted than the matrix version (cf figures 1 & 2).

For the list one, we can see that for each n > 1500, we can see differences between 25, 50 and 75%. These differences grow while n increases. For n > 5000, the 75%'s curve grows faster. For n > 6000, the 50%'s curve grows faster. The 25%'s curve increases in the same way whatever the value of n.

For the matrix, we can see that for each n > 4000, we start to see some differences between 25, 50 and 75%. These differences grow while n increases. For n > 13 000, the 75%'s curve grows faster. For n > 15 000, the 50%'s curve grows faster. The 25%'s curve increases in the same way whatever the value of n.

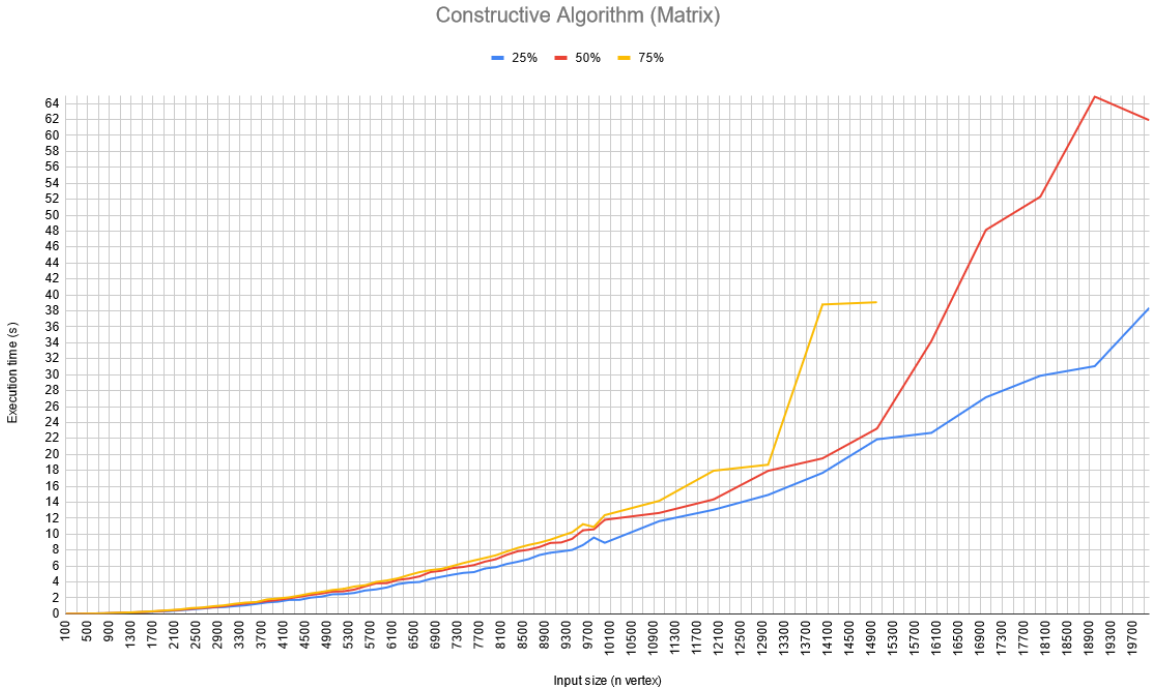


Figure 1

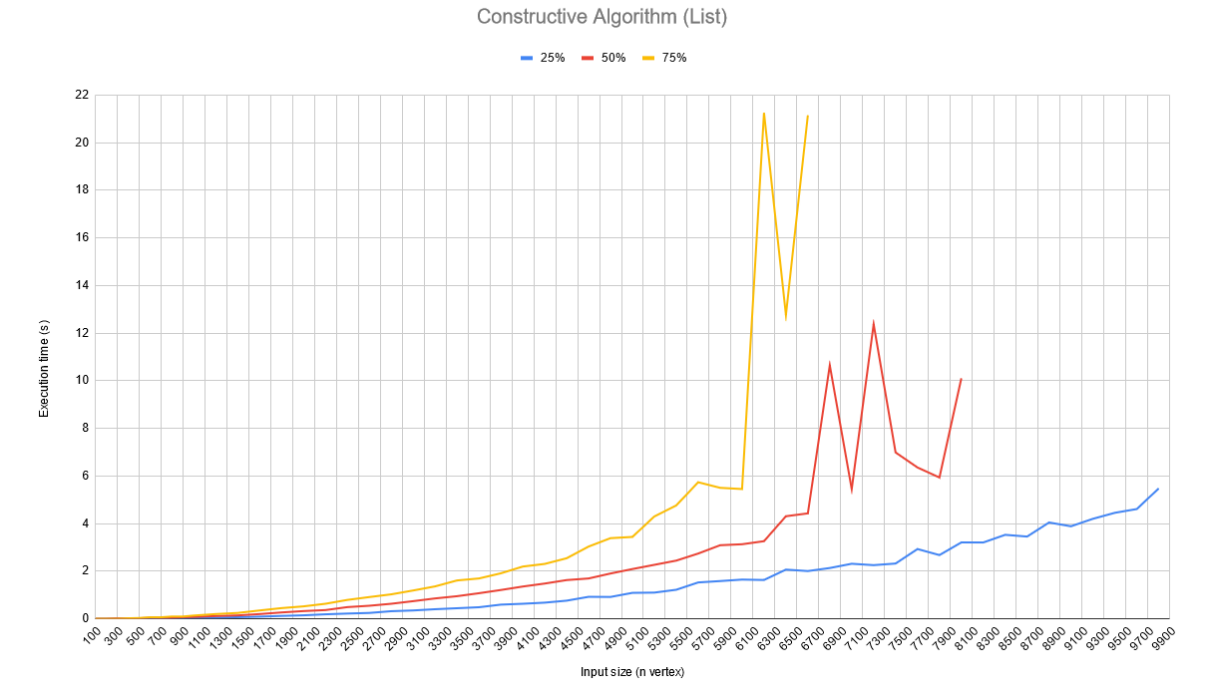


Figure 2

Implementation of the matrix version has a complexity of $9n^2 + 11n + 6$. The list version has $2n^3 + 6n^2 + 12n + 7$. With these complexities, we can suppose that the matrix version can go easily further than the list version. This hypothesis is confirmed by the following figures.

For m = 25% of n, list seems better than matrix, at least for n < 10 000 (cf figure 3).

Figure 3

For m = 50% of n, list seems better than matrix for n < 6600 (cf figure 4). After, we have peaks on the list curve then we can't really know if the matrix becomes better or if they tend to be equal. Therefore, while n is increasing, matrix's version tend to be better than list's version.

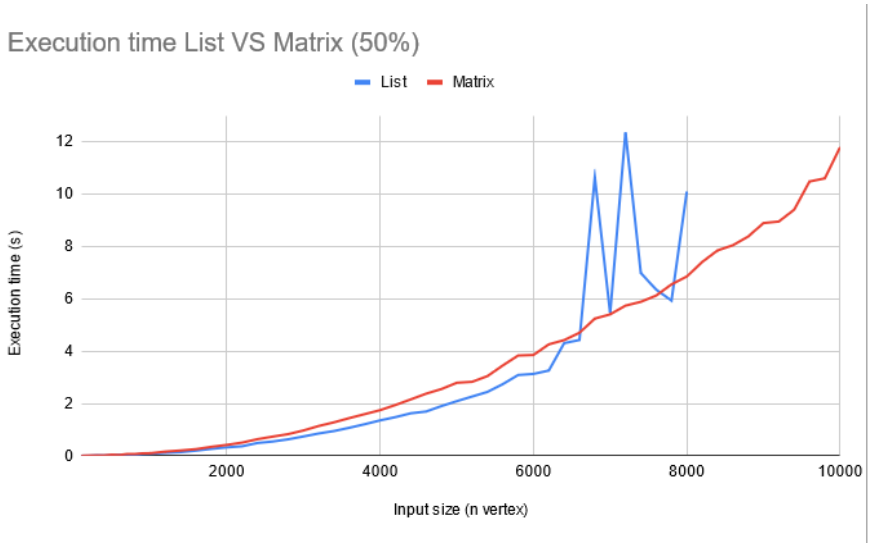


Figure 4

For m = 75% of n, list and matrix are equals for n < 4500 (cf figure 4). After this value, the matrix's version seems better.

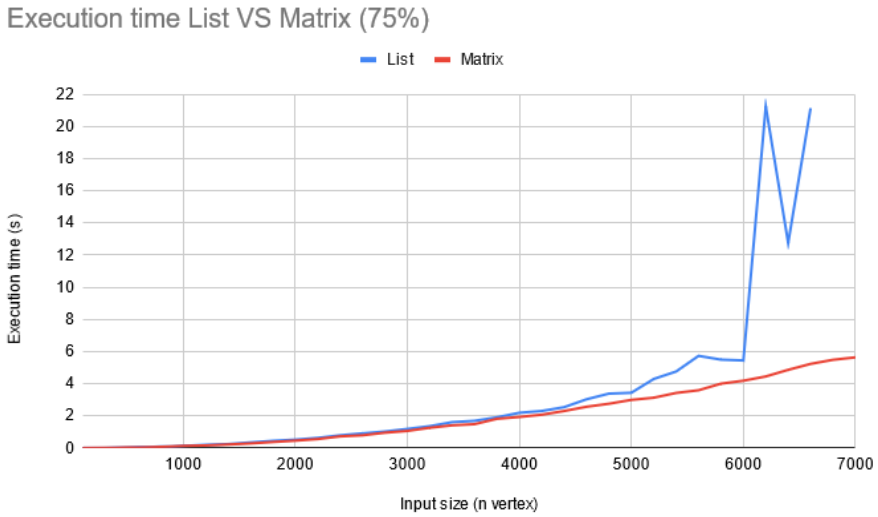


Figure 5

According to these figures and to our theoretical complexity, the list's version is the better one for little graphs (n < 10 000 and m < 50% of n). The matrix's one is not bad either ! Differences are lower than 2sec. We can have a big lack of time only if we should resolve a lot of little graphs. For bigger graphs (n > 10 000 and m > 50% of n), the matrix's version seems better. If n tends to infinite, the matrix's version will economise a lot of seconds/minutes.

The lack of time during tests is due to the instance creation and the graph creation (fill arrays / lists while browsing input file).

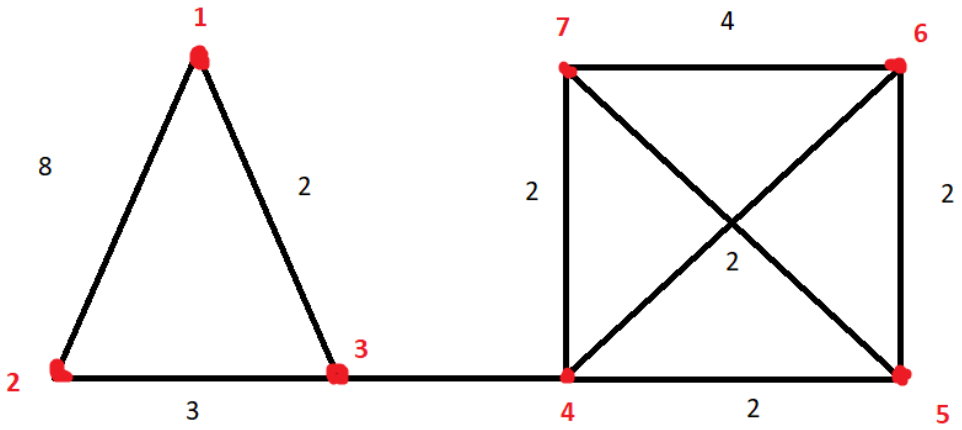
If we should choose only one version, the matrix's one will be the better option : not the better for little graphs but it stills pretty good, and the better option for bigger graphs.

Local Search Heuristic Algorithm (Q4)

Definition

Local search is a method to solve combinatorial optimization problems. As many relevant combinatorial optimization problems are NP-hard, we often may not expect to find an algorithm that is guaranteed to return an optimal solution in a reasonable amount of time.Therefore, one often resorts to heuristic methods that return good, suboptimal solutions in reasonable running times. Local search is a heuristic method that is popular for its ability to trade solution quality against computation time. By spending more time, we will generally get better solutions.
The performance of local search, in terms of quality or running time, may be investigated empirically, probabilistically, and from a worst-case perspective.

A quick example



Let's start with this graph. The best result that we can have is the clique {4,5,6,7} with a weight of 14.
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.

Step 1 : First, we start with a solution then we remove a vertex of this solution, to finally add two more that are not in the solution.

Step 2 : 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.

Step 3 : At the end of "VisitNeighbors", 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.

Result : It either updates in a better solution or stops.

Pseudo Code

Algorithm 3 Local Search

```
1: function LocalSearch(beginVertex, cutoff_time, iteration_max)
2:     clique ← Array of n positions
3:     neighbors_of_vertex ← Array with all neighobors of beginVertex
4:
5:     while neighbors_of_vertex ≠ ∅ and ellapsed time < cutoff_time and iteration < iteration_max do
6:         clique ← Add two vertices from neighbors_of_vertex and we remove them
7:         if isClique(clique) then
8:             clique ← Delete one vertex (random)
9:         else
10:            We remove two vertices we added to clique
11:    return clique
```

First we initialise our variables, in order to create our clique and our neighbors of our clique.Then in our while loop, we are going to test if we need to update our clique or not. Indeed, our while loop is conducted by the numbers of neighbors left and the time elapsed.With our neighbor array, we are going to add 2 vertices like it is said and remove them.Basically, if we have a clique better we update our clique. However if not, we end by removing two vertices that we added !

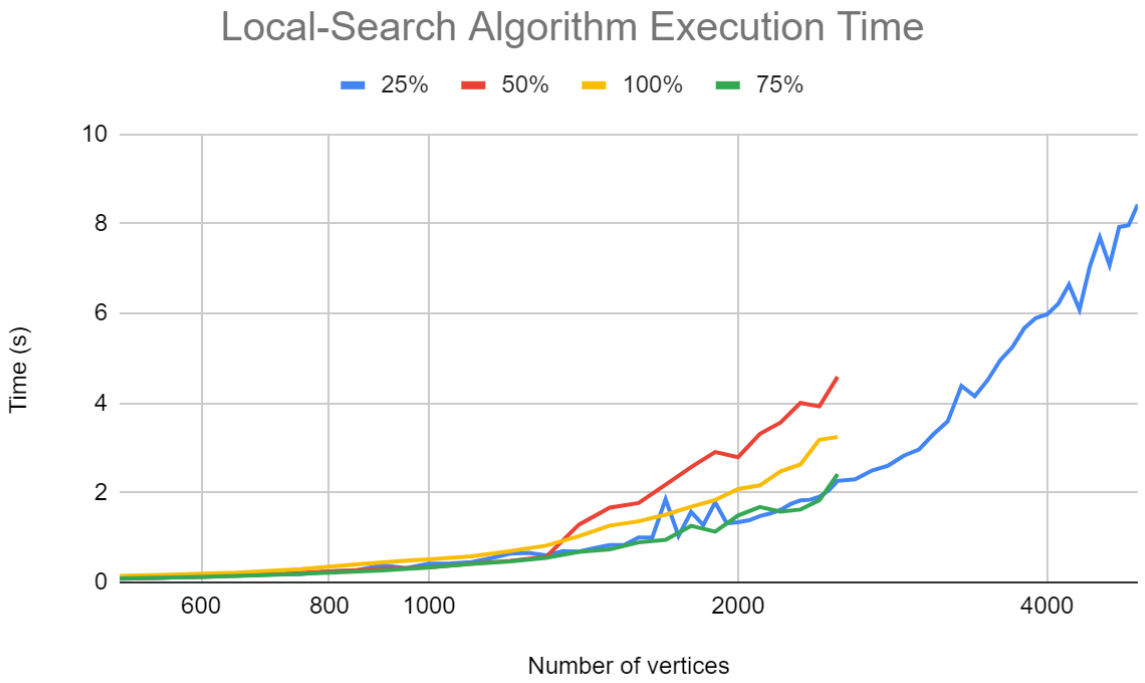
Best and Worst Cases

The worst case is having too many neighbors next to our clique. In the opposite way, the best case is having not a lot of neighbors next to the clique. Basically, the more neighbors the clique has, the more research in the algorithm they will be.

Complexity

We only travel in our “for” loop. Which means that our complexity here is down to n plus our 3 “if”. So basically, n+3.
In conclusion our complexity for Local Search is **O(n)**.

Experimental Part



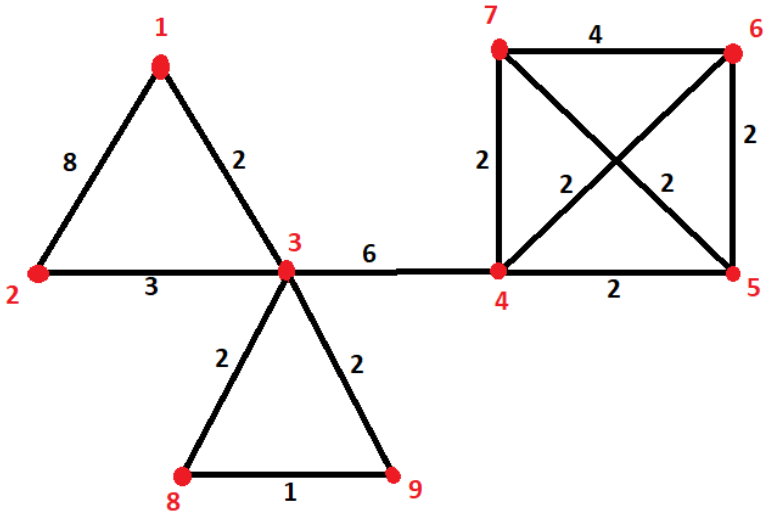
The more neighbors the clique has, the more iteration the algorithm will run in order to get all neighbors. Thus, our complexity is really high to reach out to all neighbors.

Tabu Search Mera-Heuristic Algorithm (Q5)

Definition

The Tabu Search algorithm is a metaheuristic technique. It is created to resolve problems with an approximate solution because the classic methode do not succeed in finding a correct solution in a reasonable period of time. It is a NP-Hard problem.
Tabu Search tries to find the best solution among the neighbors of the current solution for each iteration. The precedent and recent best solutions are “Tabu” to avoid going in circles.
Here, we use the Tabu Search algorithm to resolve the maximum edge weight clique problem.

A quick example



Let’s start with this graph. The best result that we can have is the clique {4,5,6,7} with a weight of 14.

Step 1 : First, we create an initial solution with a first call of local search, using the vertex of maximum weight. In this case we start with the vertex 3 , the algorithm always has a possible solution. For example the clique {1,2,3} with a weight of 13.

Step 2 : Then, we create a list which contains the neighbours vertices of the clique we just found. (Warning: don’t add the vertex if they are in the current clique)
So we have here, a list of {8, 9, 4}

Step 3 : After that, we use the local_search on every vertex of the list . If the cliques we just found aren't in the tabu-list, we compare them with the current Best clique, and if one of the cliques is better, we replace it.

Step 4 : We check the stopping conditions, if the stopping conditions are reached, we return the best solution and the algorithm is over. Otherwise, we go to step 5.

Step 5 : We update the tabu-list (basically we add the new best clique found previously) and we come back to step 1.

Result : Finally, we get a final clique {4,5,6,7} with a weight of 14. The stopping conditions are the number of iterations and the execution time. So here, we suppose that the algorithm is executed in a short time. If it is false, the algorithm returns the best current solution. For example the clique {1, 2, 3} with a weight of 13.

Pseudo Code

Algorithm 4 Tabu-Search algorithm

```
1:  procedure TabuSearch(iteration, maxTabusize, maxTimer)
2:      maxNode ← getMaxWeightedNode()
3:      firstClique ← Local-Search(maxNode)
4:      sBest ← firstClique
5:      bestCandidate ← firstClique
6:      tabuList ← [ ]
7:      tabuList.push(firstClique)
8:      while stoppingCondition do
9:          sNeighborhood ← getNeighbors(bestCandidate)
10:         bestCandidate ← sNeighborhood[0]
11:         for each s ∈ sNeighborhood do
12:             if (not tabuList.contains(s)) and (fitness(s) > fitness(bestCandidate)) then
13:                 bestCandidate ← s
14:         if (fitness(bestCandidate) > fitness(sBest)) then
15:             sBest ← bestCandidate
16:         tabuList.push(bestCandidate)
17:         if tabuList.size > maxTabuSize then
18:             tabuList.pop(0)
19:     return sBest
```

Our initial solution will start at the vertex with the maximum weight. This way we optimize our chances of starting from a vertex that is part of the best clique or around the best clique.
In this algorithm, the “tabuList” which contains the best recent solutions are considered tabu and the “fitness” function returns the weight of the specified clique.

The “while” will continue searching for an optimal solution (by calling getNeighbors) until the stopping conditions are met (exemple : time limit, the number of iterations without modifying the best solution). In getNeighbors we call the Local-Search Algorithm which will find the best local Algorithm around the current one. Additionally, the algorithm keeps track of the best solution in the neighbourhood, that is not tabu. sBest is the best clique in all the graph and bestCandidate is the best clique around our last bestCandidate. That’s why at first bestCandidate is our firstClique.

We judged that 30 minutes was sufficient for our tests because 30 mins would have been enough for this kind of instance, for other instance tests with more vertex we could increase the time limit in order to find a better solution. We put an iteration of 5 because after several tests at 1 iteration and at 10 iteration we found that the most suitable was 5 for our instances.
The time limit and the number of iterations allow us to define if we want a solution closer to the best clique or not.

Complexity

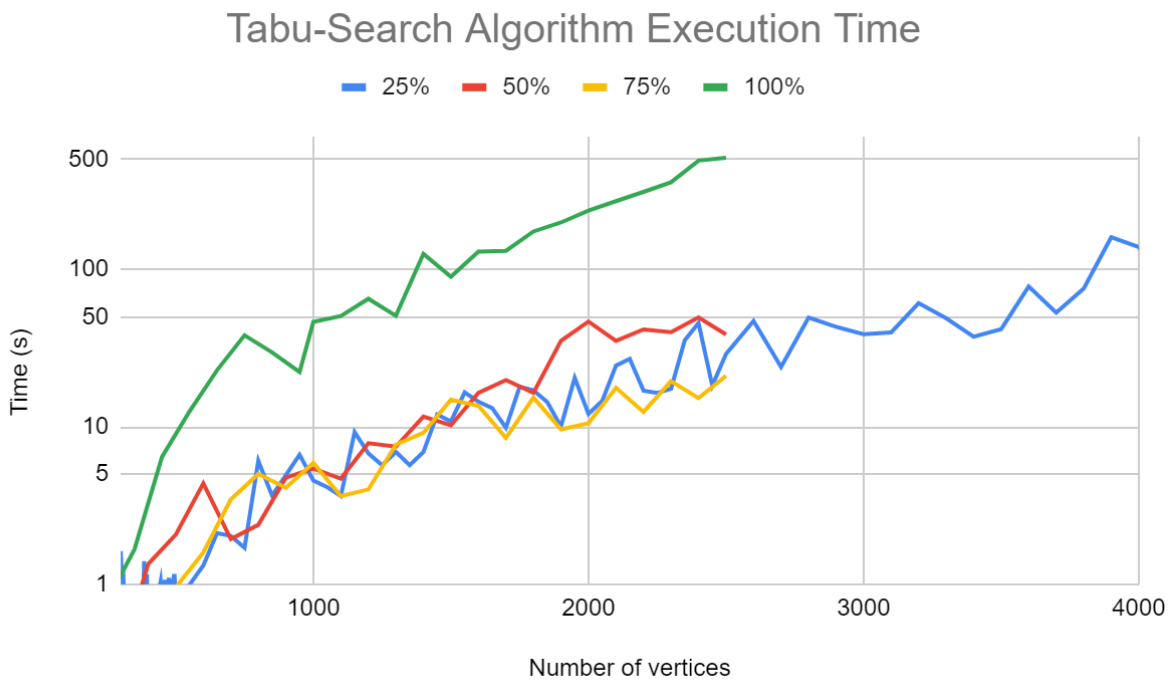
Summary	Temporal Complexity			Spatial Complexity
	Best	Average	Worst	
Tabu-Search	O(n^3)	O(n^3)	O(n^3)	O(1)

All loops of this algorithm are executed in n iterations so we have this equation :

= X +n^3+n^2 *X +n^2 X is the complexity of the localSearch : O(n)
= 2n^3 + n^2 +n

- Best case : The algorithm doesn’t have an ideal or best case so there is not a complexity in this case because it has to go over all the loops of the algorithm. So this complexity is the same as the medium case : O(n^3).
- Average case : We go over n iterations multiplicated by n elements so the complexity is O(n^3).
- Worst case : The algorithm has a lot of vertices and a big percentage of edges . The complexity is still O(n²).

Experimental Part



We can see that the tabu-search is not influenced by the percentage of vertices connected. Except for the 100% because the algorithm tries to find a better solution which doesn't exist even if he got the best of all the graphs.

Comparison of algorithms (Q7)

Percentage of error

In this part we want to check the quality of our output files for little graphs (n < 505). This quality is calculated by the percentage of error with the following formula :

$$\left|\frac{vExperimental - vTheoretical}{vTheoretical}\right| * 100$$

vTheoretical is the max clique weight collected by the Exact Algorithm.
vExperimental is the max clique weight collected by our algorithm (Local Search, Tabu Search, Constructive).

If the value is close to 0, it means that the output is close to the exact algorithm's output and we have a good quality. If not, the output is close to 100, the output is really far from the exact algorithm's output and we have a bad output quality.

Constructive Algorithm seems better than Tabu Search and Local Search for all these input files. It is much better when we have m equals to a huge percentage of n. In the 100% case, it is perfect with 0% of error.

We notice that some value of the tabuSearch has a worse solution than localSearch, it is normal. It is because we execute the localSearch and we retrieve the datas (execution time and weight of the final solution). Then we ran the tabuSearch again. The local search algorithm takes randomly some neighbors of the clique. Thus, thanks to the randomiser, we could find a clique right away !

We can conclude that the constructive algorithm is the most reliable. Normally, the Tabu Search should be the best one but we probably made a mistake in our Local Search which impacts our Tabu Search.

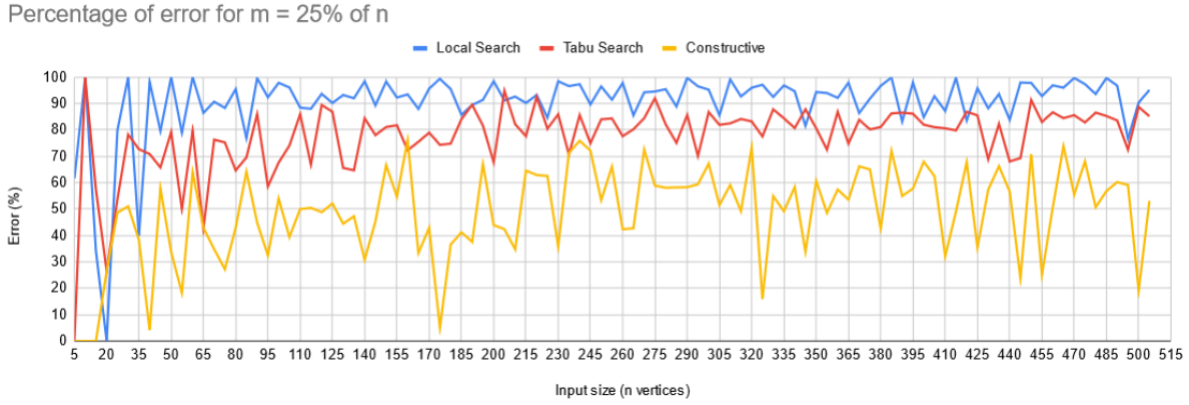


Figure 1

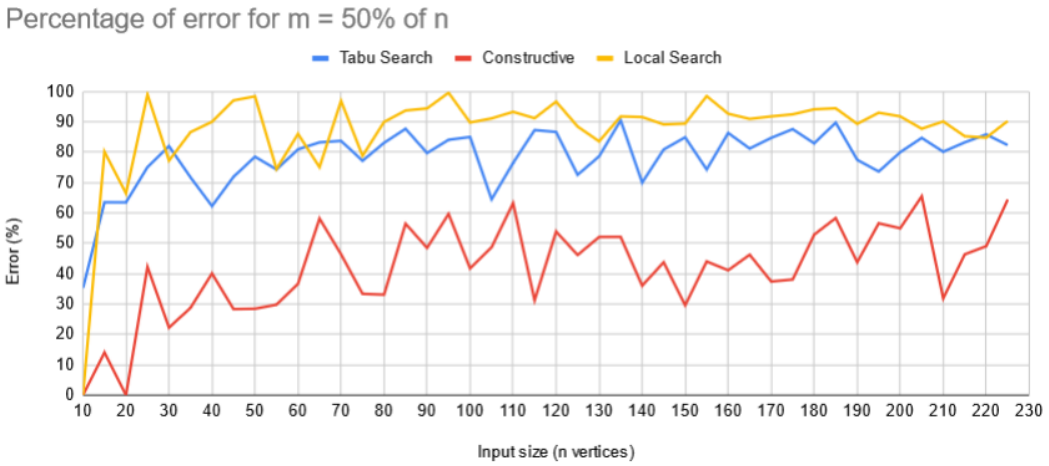


Figure 2

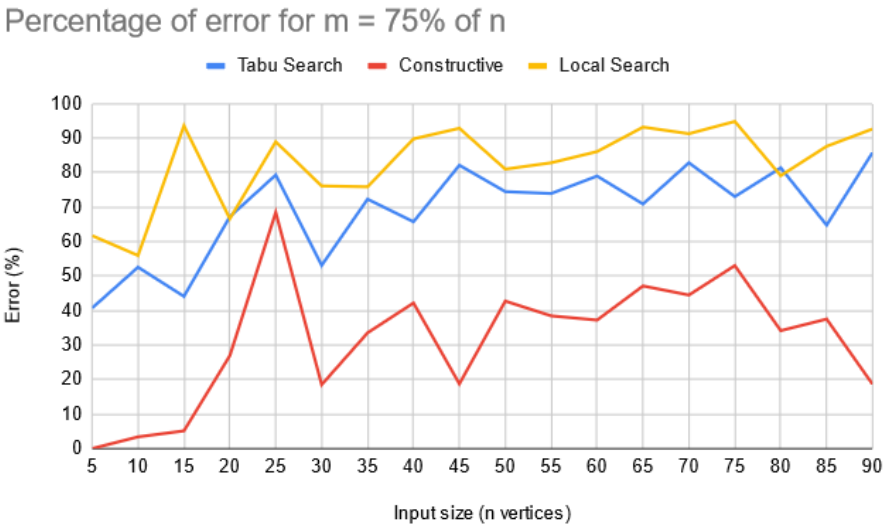


Figure 3

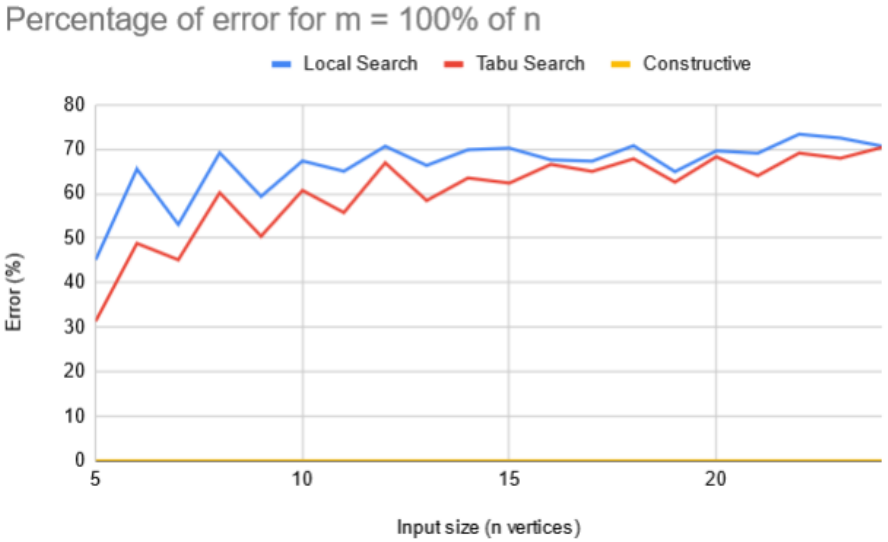
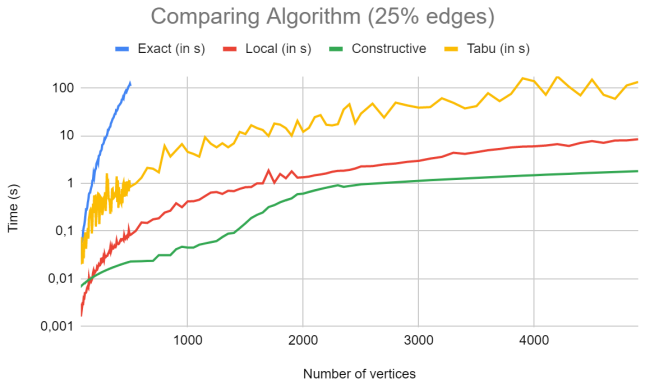
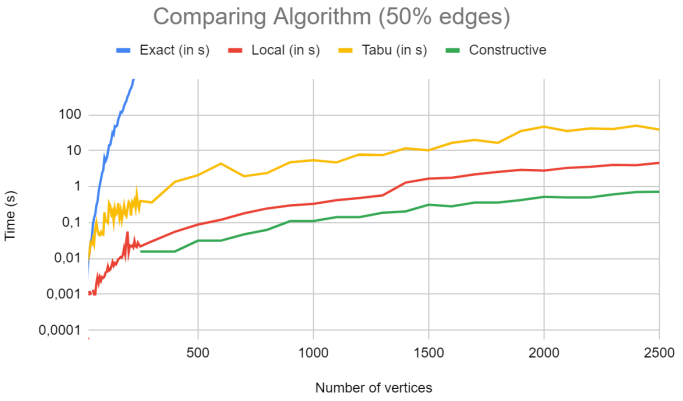
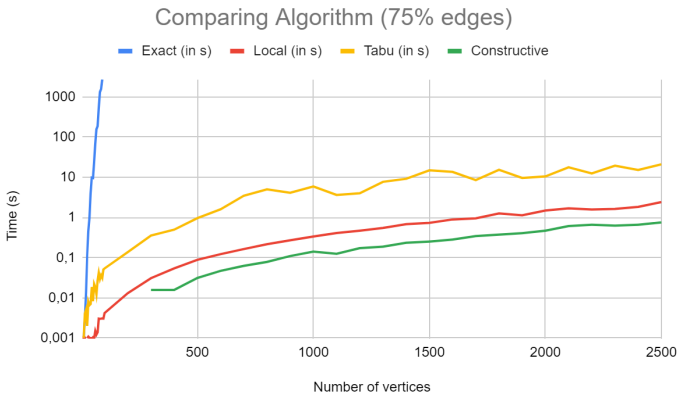


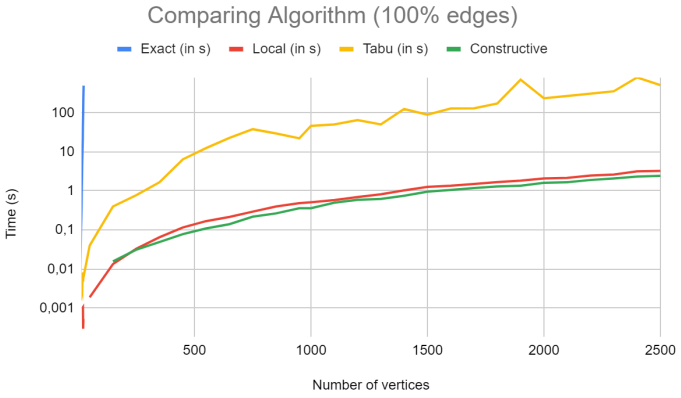
Figure 4

Execution time









In order to have a better graph we used the logarithm scale on the Y axe.
We can see on the graph above that the exact algorithm is taking a lot of time to execute. This algorithm is indeed slow. Thus, the most gainful algorithm (time option) is the local search. The tabu search is in the middle range.
The complexity of the local search, the tabu and the constructive are proportional. They have the same pattern. However, the exact has a higher complexity, thus a different pattern.

Conclusion

To conclude, we build four algorithms that solve the maximum edge weight clique problem (MEWCP).The exact algorithm is the one that gives the best solution but can't solve big graphes because of his exponential complexity so it can be used only for small graphs. Then, there is the constructive heuristic algorithm which builds a solution step by step. This one is very fast and it can be used with big graphs but it rarely returns a good solution. The local search heuristic algorithm returns a solution in only one part of the graph. Therefore, it is fast enough. Finally, the tabu search meta-heuristic algorithm uses in many loops the local search algorithm to find a best solution but it contains stopping conditions in order that the algorithm can solve in a reasonable time and avoid looping in circles.Theoretically, the Tabu-search using the Local-search should be more optimal in terms of time and reliability than the Exact-Algorithm for files with larger numbers of vertices. Unfortunately we encountered some difficulty during the design of the Local-search which led to the poor reliability of the best clique found and therefore affected our Tabu-search.