Graph Theory - CIR3

Final project : Travelling Salesman Problem

Eléna COMMEREUX - Léo PINEAU
Maxime NARBAUD - Tanguy MORINIÈRE

11 May 2020

Contents

Introduction

The Travelling Salesman Problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, many heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.

TSP can be modelled as a complete undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

Therefore, given a weighted undirected complete graph, the TSP consists in finding a Hamiltonian cycle (a cycle that goes through all vertices exactly one) that minimises the sum of its weights.

Real-life situations

Several real-life situations can be modelled as the travelling Salesman problem. For problems related to the itinerary, to transport. The goal is to calculate the minimum route in the following situations: for school bus routes, the postman circuit, delivery people, garbage collectors...

It can also reduce the time needed to perform a task, when you reduce the distance to do something, you save time. This is the case in industry with the machine that performs actions on an object (drilling). This can be useful to solve the problem of minimizing the wiring distance in the field of IT or telephony during installation.

Exact algorithm

This algorithm tests all possibilities in order to have an exact, and the better path. It builds a complete path starting from an initial vertex. For this, it adds at each recursive call a new vertex in the path, which was not marked as visited in this call, or in the previous.

Moreover, the distance between the last vertex in the path and the new one is added with the previous distance, so that the function return if this distance is higher than the distance of the better path.

3.1 Pseudo-code

Input: P is an empty path

Function *backTrackRecursive(Graph : G, Path : P, Path :* $tempP$*, Int :* $previousDistance$*, Int :* $previousVertex$*) :*

- Add $previousVertex$ in $tempP$
- **if** *all vertices have been visited* **then**
  - $previousDistance \leftarrow previousDistance$ plus distance between $previousVertex$ and starting vertex
  - **if** $previousDistance$ < $bestDistance$ **then**
    - $bestDistance \leftarrow previousDistance$
    - $P \leftarrow tempP$
    - Add finishing vertex in P (same as starting)
  - **end**
- **else**
  - **foreach** *vertex* v *in* G *not visited* **do**
    - $actualDistance \leftarrow previousDistance$ plus distance between $previousVertex$ and v
    - **if** $actualDistance$ >= $bestDistance$ **then**
      - **return**
    - **else**
      - $backTrackRecursive(G, P, tempP, actualDistance, v)$
    - **end**
  - **end**
- **end**

end

Function *backtracking(Graph : G, Path : P) :*

- $backTrackRecursive(G, P, tempP, 0, 1)$

end

3.2 Time complexity

Consider n the number of vertices in the graph G. Here we have a recursive function with the following loop:

```
foreach vertex v in G not visited do
    ...
end
```

With the initial call of $backTrackRecursive()$, we add the starting vertex in the path and we mark it as visited. The loop run as many times as there are unvisited vertices, so we do the loop $(n-1)$ times.

In this loop, we call recursively the function $backTrackRecursive()$, by marking the selected vertex in the loop as visited. So, in the next call, the for-each loop will run $(n-2)$ times. At the next function call the loop will run $(n-3)$ times... Until all vertices has been visited.

Finally, we have a complexity of $(n-1)$*$(n-2)$*...*1, that is equal to : $\mathcal{O}((n-1)!)$.

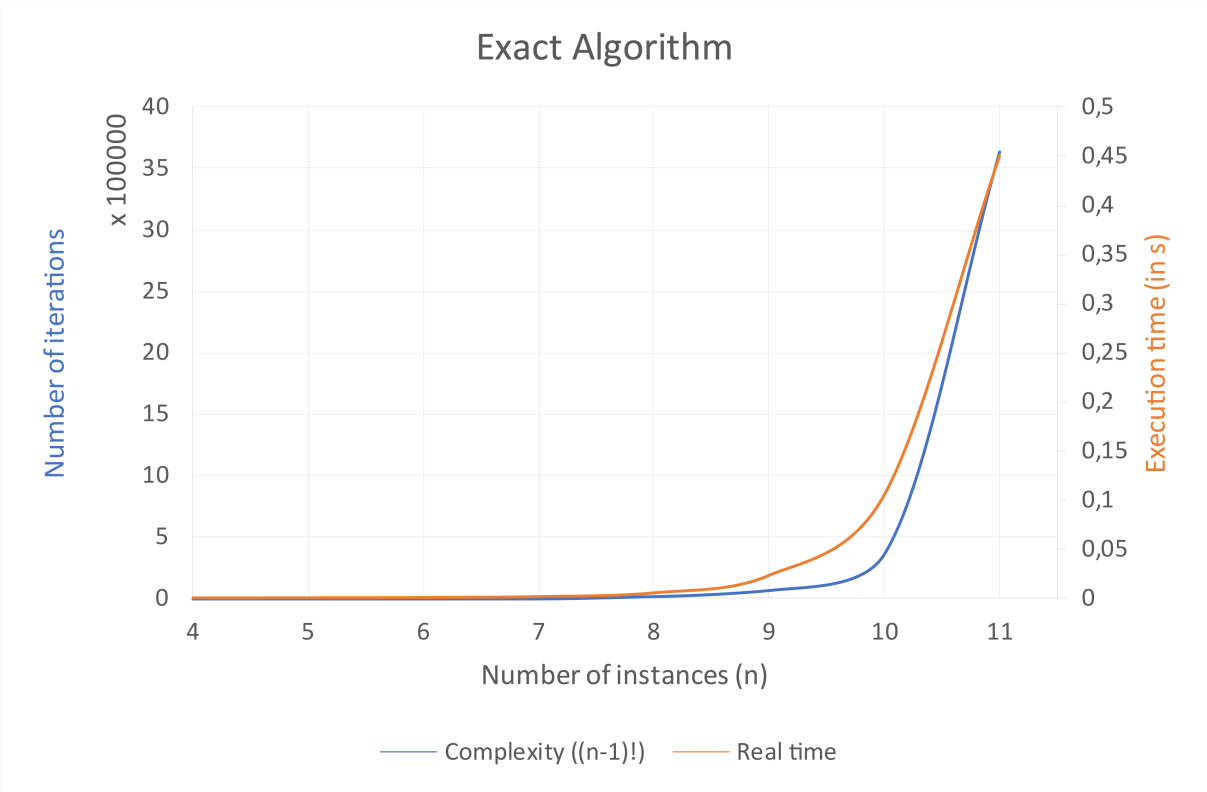
This complexity is in the worst case, because in each function call the loop can be stopped if the calculated distance is higher than the better distance.

3.3 Optimal Solution

This method cannot provide an optimal solution with instances higher than 17 for our machine. Indeed, we managed to calculate the exact path in 30 minutes for the instance 17 by optimizing the compilation. However, it was already too long and higher instances require better machines, although there is still a physical limit.

More, with these instances, we find the same results by using constructive heuristic followed by a local search, in a much better time.

3.4 Execution time and performance



We can see that the execution time's curve has nearly the same progression that the theoretical. The progression is slow at the beginning, and increases very quickly from a certain value, typically from the factorial function.

We have built the execution time's curve with the following values that we measured with different instances (*Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz*) :

Nb of instances	4	5	6	7	8
Exec. Time (in s)	0,00004	0,0002	0,00045	0,0014	0,0052
Nb of instances	9	10	11	12	13
Exec. Time (in s)	0,023	0,105	0,45	1,9	10
Nb of instances	14	15	16		
Exec. Time (in s)	60	342	1262		

Constructive heuristic

The global functioning of this algorithm is quite simple: we add the initial vertex in the path. Then, at each iteration we search the vertex which is closest to the previous one and not visited, we add it to the path, and we mark it as visited. Repeat until all vertices are visited.

4.1 Pseudo-code

```
Input: P is an empty path
Function constructiveHeuristic(Graph : G, Path : P) :
    Add starting vertex in P
    while we have not visited all vertices do
        closestDistance ← ∞
        foreach vertex v in G do
            if vertex v not visited then
                actualDistance ← distance between the last vertex in P and v
                if actualDistance < closestDistance then
                    closestDistance ← actualDistance
                    closestVertex ← v
                end
            end
        end
        Add closestVertex in P
        Mark closestVertex as visited
    end
    Add finishing vertex in P (same as starting)
end
```

4.2 Time complexity

Consider n the number of vertices in the graph G, we have this first loop:

```
while we have not visited all vertices do
    ...
end
```

We do this first loop n times because we want to visit all vertices one time.
Next, we have a second loop in this first one:

```
foreach vertex v in G do
    ...
end
```

In this second, we want to run all vertices, and we have n vertex in the graph G. So, we do this loop also n times.

All other operations, like adding a vertex in the path P, getting the distance between two vertices (due to adjacency matrix) or marking a vertex as visited cost $\mathcal{O}(1)$.

So, finally we do a first loop n times with a second loop in this that we do n times. We have a total complexity of n*n that is equal to: $\mathcal{O}(n^2)$

4.3 Optimal Solution

This algorithm is fast, however it does not find the optimal solution, or at least does not guarantee an optimal solution. In some cases, the optimal solution can be found, depending on the graph configuration.

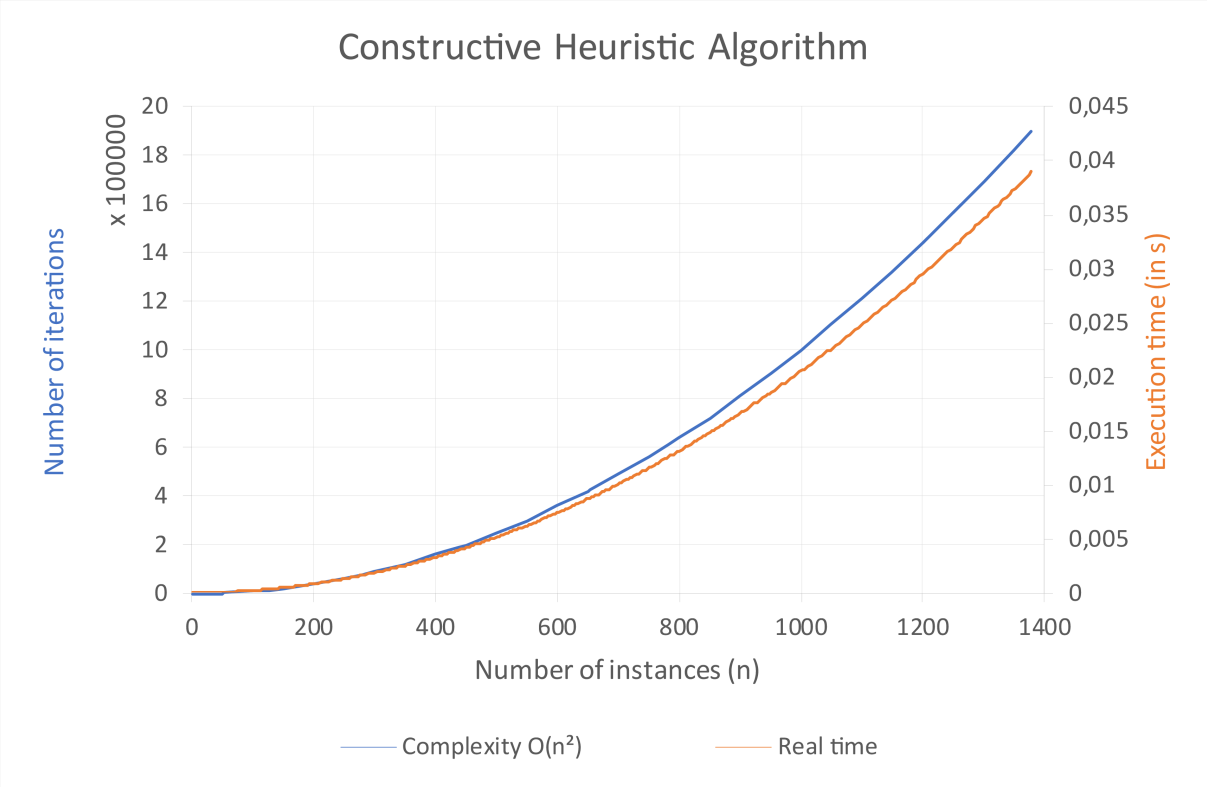
We measured the following path lengths, giving an error percentage of **6,26** between the values of the constructive algorithm and the exact. However, this value is only representative for small instances:

Instances	**7**	**10**	**11**	**12**
Exact	1346	1637	1639	1799
Constructive	1403	1730	1732	1892
Instances	**13**	**14**	**15**	**16**
Exact	1805	1872	1908	2059
Constructive	1979	2046	2082	2161

Moreover, we observed that if the distance between each vertex is in a small interval, then the constructive algorithm can give a good result. On the other hand, if this interval becomes larger, the error in the solution also becomes greater. To illustrate that, we generated 4 instances containing 14 vertices with different range (minimum and maximum distance between each vertex):

Range	Exact Path	Constructive Path	Constru / Exact (%)
990 -> 999	13875	13883	0,1
500 -> 999	7712	8006	3,7
250 -> 999	5207	5635	7,6
10 -> 999	2223	2763	19,5

4.4 Execution time and performance



We can see that the execution time's curve has nearly the same progression that the theoretical.

We have built the execution time's curve with the following values that we measured with different instances (*Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz*) :

Nb of instances	5	10	11	17	51
Exec. Time (in s)	0,00001	0,00002	0,000031	0,000036	0,0001
Nb of instances	52	100	101	127	280
Exec. Time (in s)	0,00015	0,0004	0,00041	0,0005	0,0023
Nb of instances	439	654	783	1379	
Exec. Time (in s)	0,0041	0,0081	0,013	0,039	

Local search heuristic

We start with a path that can be generated with the constructive algorithm, or filled in the order like this $\{1, 2, 3, 4, 5, 6...\}$. At each iteration, we search to invert two vertices in order to reduce the total path length. For instance, we take the edge $(1, 2)$ + $(5, 6)$, then we check by inverting $(1, 5)$ + $(2, 6)$ if the distance is lower. If so, we invert it in the path, giving: $\{1, 5, 4, 3, 2, 6...\}$. Furthermore, we invert the path between these two vertices, so as not to extend the path length.

5.1 Pseudo-code

Input: P is a path that can be generated with a constructive heuristic algorithm

Function *localSearch(Graph : G, Path : P, Int : $maxIteration$) :*

- $permutation \leftarrow$ true
- $iteration \leftarrow 0$
- **while** $permutation$ *is true* ***OR*** $iteration < maxIteration$ **do**
  - $permutation \leftarrow permuteEdges(G, P)$
  - Increment $iteration$
- **end**

end

Function *permuteEdges(Graph : G, Path : P) :*

- **foreach** *vertex* $v1$ *in* P *until the penultimate* **do**
  - **foreach** *vertex* $v2$ *starting at* $v1 + 2$ *in* P **do**
    - **if** *distance between (*$v1$*,* $v1 + 1$*) plus distance between (*$v2$*,* $v2 + 1$*) > distance between (*$v1$*,* $v2$*) plus distance between (*$v1 + 1$*,* $v2 + 1$*)* **then**
      - Swap $v1 + 1$ with $v2$ in P
      - Reverse the order of all vertices between $v1 + 2$ and $v2$ in P
      - **return** true
    - **end**
  - **end**
- **end**
- **return** false

end

5.2 Time complexity

Consider n the number of vertices in the graph G, we have this first loop in the first function $localSearch()$:

while $permutation$ *is true* ***OR*** $iteration < maxIteration$ **do**
  ...
end

Consider that the number $maxIteration$ is equal to the number of vertices in the graph G. So, we do this first loop n times in the worst case. Indeed, this loop can be stopped if there is no longer any possible permutation.

In this loop we call the second function $permuteEdges()$ that contain two other loop :

foreach *vertex* $v1$ *in* P *until the penultimate* **do**
  ...
end

In this loop, we take all vertices from the path P until the penultimate (not included), that contain n vertices (because we have a complete path). So we do this loop $(n-2)$ times in the worst case.

Next, we have a last loop in the previous one:

foreach *vertex* $v2$ *starting at* $v1+2$ *in* P **do**
  ...
end

Here, we take all vertices until the last (not included) in the path P, and we start two vertices after the previous loop. So we also do the loop $(n-2)$ times at first time, after $(n-3)$ times, then $(n-4)$... until $T(1)$.

Finally, in the function $permuteEdges()$ we have a time complexity of $(n-2)$ + $(n-3)$ + ... + $T(1)$ giving a function in $\mathcal{O}(n^2)$. And we call this function n times in $localSearch()$, therefore we have a total complexity equal to $\mathcal{O}(n^3)$ in the worst case.

Moreover, we can consider that swapping two vertices or reverse the path cost $\mathcal{O}(1)$ with some representation in memory.

5.3 Optimal Solution

In order to have an optimum solution, but not necessary the optimal solution, we need to execute this algorithm until there is no more possible permutations. We cannot have the optimal solution for this algorithm if we stop the first while before all possible permutations has been done.

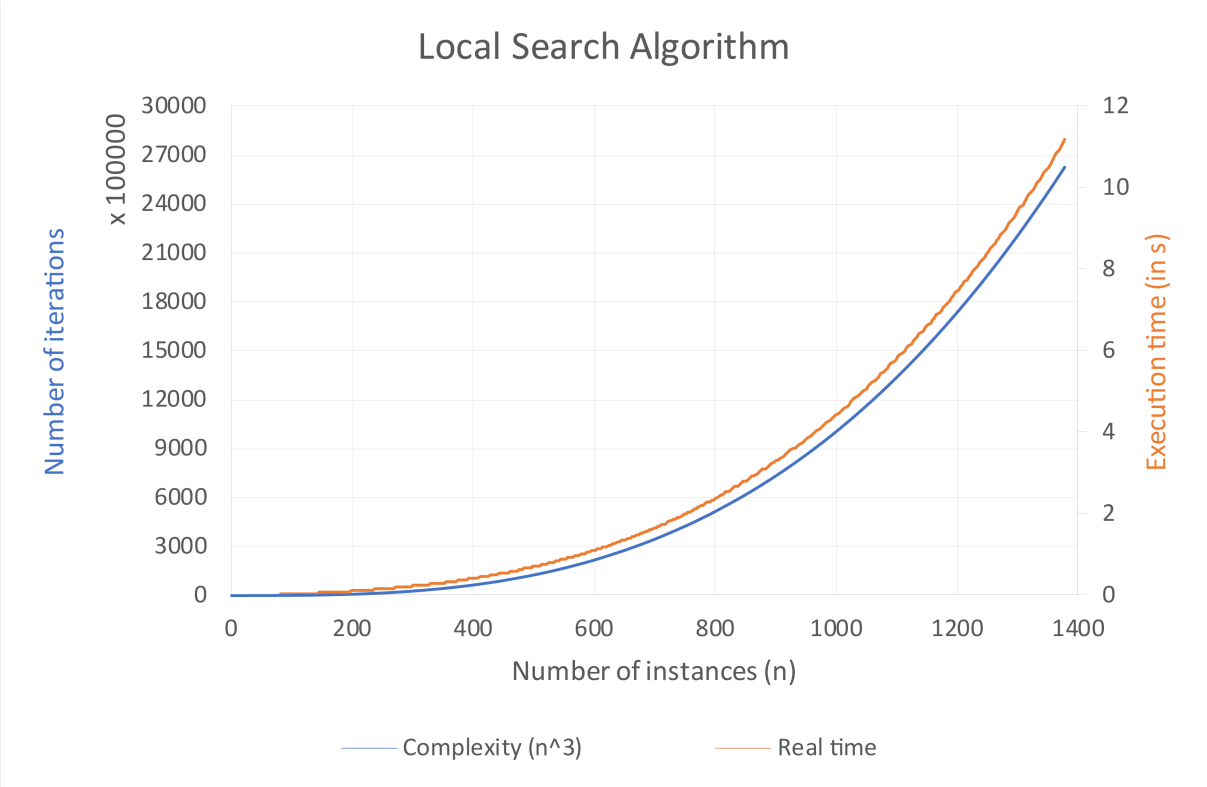
We observed that we must make on average "0,3 * the number of vertices" iterations, in order to have not any more permutation reducing the path length (with an initial path generated with the constructive heuristic algorithm). According to the following values that we measured:

Instances	**10**	**17**	**51**	**52**	**100**	**101**
Needed iteration	3	4	14	16	43	36
Instances	**127**	**280**	**439**	**654**	**783**	**1379**
Needed iteration	33	57	100	237	203	428

Here, we compare values obtained with the constructive heuristic algorithm and the local search algorithm. Knowing that this algorithm uses the constructive one, this gives us an idea of the improvement that it can bring to the result, of the order of 13.6% on average:

Instance	Constructive	Local Search	Constru / Local (%)
17	2187	2085	4,7
51	511	438	14,3
52	8980	7967	11,3
100	27807	22437	19,3
101	803	658	18,1
127	135737	122097	10,0
280	3157	2767	12,4
439	131281	113210	13,8
654	43457	35422	18,5
783	11054	9336	15,5
1379	68964	60382	12,4
5915	695602	603007	13,3
		Mean	13,6

5.4 Execution time and performance



We can see that the execution time's curve has nearly the same progression that the theoretical. Moreover, the growth is slightly faster than the constructive heuristic algorithm which was in $\mathcal{O}(n^2)$, while here we are in $\mathcal{O}(n^3)$.

We have built the execution time's curve with the following values that we measured with different instances (*Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz*) :

Nb of instances	5	10	11	17	51
Exec. Time (in s)	0,000029	0,000064	0,000069	0,000094	0,0015
Nb of instances	52	100	101	127	280
Exec. Time (in s)	0,006	0,01	0,009	0,009	0,05
Nb of instances	439	654	783	1379	
Exec. Time (in s)	0,33	2,06	1,74	11,2	

GRASP meta-heuristic

This algorithm has in a first time a construction phase. In this, we built iteratively a solution, one element at a time. We choose randomly one of the best candidates in a restricted candidate list (RCL), but not necessarily the better candidate.

In a second time, we apply a local search algorithm (here a 2-opt algorithm), in order to make the previous built solution better. Finally, we do these 2 phases a certain number of iterations.

6.1 Pseudo-code

Input: P is an empty path

Function *grasp(Graph: G, Path: P, Int: $alpha$, Int: $maxIteration$):*
    $iteration \leftarrow 0$
    while $iteration$ < $maxIteration$ **do**
        $constructGreedyRandomizedSolution(G, tempP, alpha)$
        $localSearch(G, tempP)$
        if $tempP$ *is better than* P **then**
            $P \leftarrow tempP$;
        end
        Increment $iteration$
    end
end

Function *constructGreedyRandomizedSolution(Graph: G, Path: $tempP$, Int: $alpha$):*
    $weight \leftarrow sumOfWeightsForEachVertex(G)$
    while $tempP$ *is not complete* **do**
        $RCL \leftarrow makeRCL(G, alpha, weight)$
        $chosenVertex \leftarrow selectRandomElement(RCL)$
        Add $chosenVertex$ in $tempP$
        $adaptGreedyFunction(G, weight, chosenVertex)$
    end
end

```
Function makeRCL(Graph: G, Int: alpha, Pair: weight):
    foreach vertex v in weight do
        if value of v (weight) < maxWeight then
            maxWeight ← value of v
        end
    end
    foreach vertex v in weight do
        if value of v (weight) >= alpha*maxWeight then
            Add v in RCL
        end
    end
    return RCL
end

Function adaptGreedyFunction(Graph: G, Pair: weight, Int: chosenVertex):
    foreach vertex v in weight do
        if v is chosenVertex then
            Erase v in weight
        end
        else
            Remove distance chosenVertex from v in weight
        end
    end
end
```

6.2 Time complexity

Consider n the number of vertices in the graph G, we have this first loop in the first function grasp():

```
while iteration < maxIteration do
    ...
end
```

We suppose that $maxIteration$ is equal to the number of vertices in the graph G. So, the worst case would be that we do this first loop n times. However, this loop can be reduced by choosing a lower number of iterations.

In this first loop we will call several functions. The first one is $constructGreedyRandomize Solution()$. For begin, we are going to initialize the weight associate at every vertex. This function cost $\mathcal{O}(n^2)$:

```
foreach vertex v1 in G do
    foreach vertex v2 in G do
        ...
    end
end
```

Then, we are going to create a randomize solution:

```
while tempP is not complete do
    ...
end
```

In this loop, we make a restricted candidate list (RCL) for each vertex. Every time we choose a vertex, we need to adapt the solution and the associate weight of every other vertex. So, we always do this loop n times, but we need to calculate functions $makeRCL()$ et $adaptSolution()$.

In order to create a restricted candidate list, we need to find the vertex with the biggest weight.

In the worst case, we do this loop n times, when no vertex is already added to the solution. However, each time a vertex is added to the solution, the $weightList$ decreases and this loop also decreases.

Once we found our biggest distance, we go through our list again to create our RCL. This loop has the same complexity as the previous one:

```
foreach vertex v in weight do
    ...
end
```

After choosing our solution, we must adapt the $weigtList$ with the function $adaptSolution()$:

```
foreach vertex v in weight do
    ...
end
```

We go through the $weightList$ once in order to erase the chosen solution, then we do this loop again to adapt the weight of vertices. This loop has the same complexity as the previous one.

Finally, in the function of $constructGreedyRandomizeSolution()$ we have a complexity of n^2 $4n^2$. The complexity of $localSearch()$ has been calculated before : $\mathcal{O}(n^3)$.

To conclude:
In the function of $grasp()$ we repeat these functions n times, so we have a complexity of n * (n^2 + $4n^2$ + n^3), giving $\mathcal{O}(n^4)$ in the worst case.

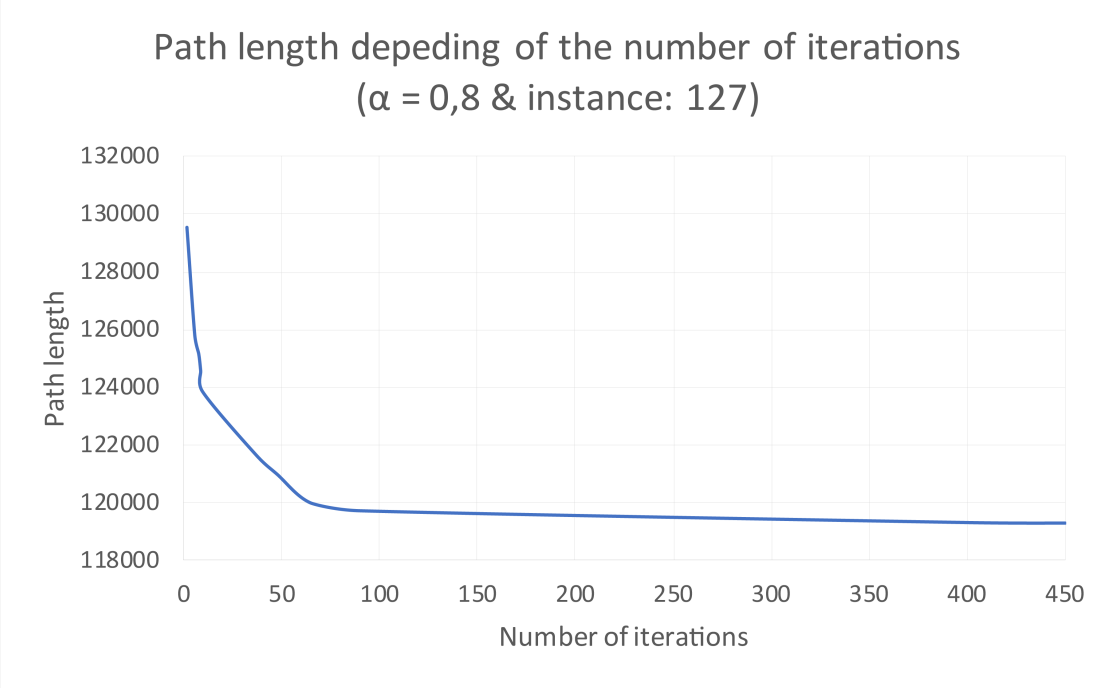
Moreover, the complexity of the Grasp algorithm may be different depending on input parameters. For example, the number of maximum iterations for Grasp can change the complexity.

6.3 Optimal Solution

The Grasp algorithm can provide an optimal solution for each instance. The particularity of this algorithm is that it randomly create a solution. The user can set parameters in order to adjust the solution, but it will always be random. So, we can imagine that the random solution gives us the exact solution. In this case the Grasp algorithm could provide us an optimal solution for each instance.
However, in most cases we need to use the local search algorithm to improve our solution.

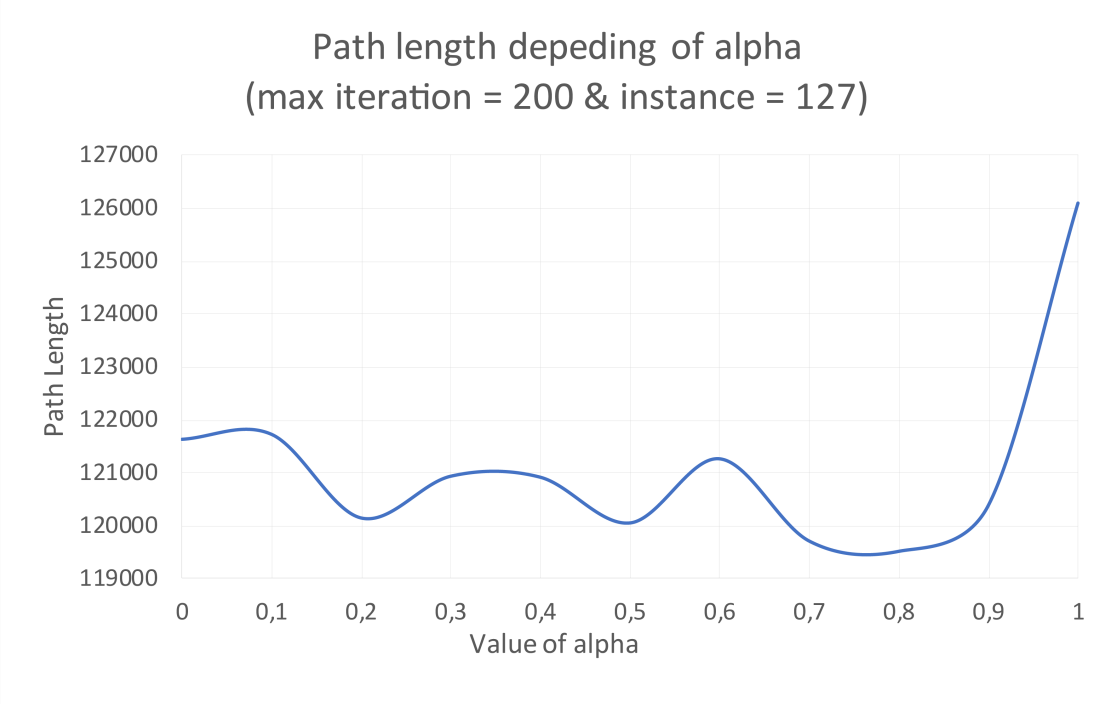
In order to have an optimal solution, we must determine the better parameter for this algorithm (max number of iteration and alpha number). For this, we used the instance 127 and we ran the algorithm 5 times and do the mean of the obtained results.

With this instance, we determined that from around 80 iterations, the total path length reduces much less quickly. However, execution time remains low even with 500 iterations (about 5 seconds). The following graphic present the best path length found at a given iteration:



In some cases, we find the best path in the first iterations but this remains rare. In addition, we generally find better and better results by increasing the number of iterations, but in this case, the difference of the path length between iteration 100 and 500 is minimal.

Next, we determined that the best value for alpha is about 0,8. For this, we ran the algorithm 5 times with 200 iterations for each value of alpha between 0 to 1, with a 0,1 interval:

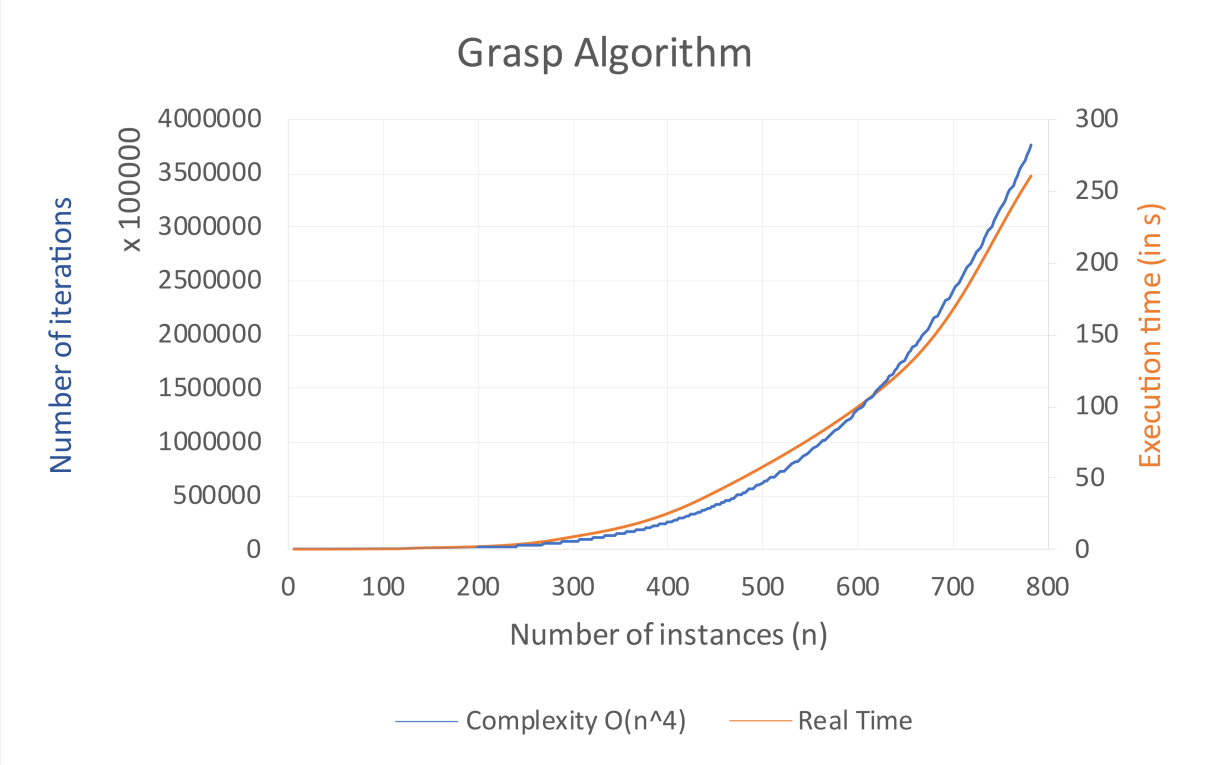


In our case, Grasp algorithm provide good results (short distance) and good execution time (<1min) until the 439 instances. The instance 783 and 1379 can also provide good results but with a slightly longer execution time (1min < x < 3min). For other instances, the execution time would be much longer if the same level of requirement were maintained.

With this algorithm, we need to make a choice, whether decrease our parameters in order to reduce the execution time or keep our parameters and have an execution time much longer. With larger number of vertices, it can be more complicated to have a result close to an optimal solution.

Even if we set the right parameters, the Grasp algorithm can produce bad results. Indeed, a part of the program is random. However, the number of iterations for grasp is here to reduce this problem. So, unless you are unlucky the Grasp algorithm cannot provide a bad result with the right parameters.

6.4 Execution time and performance



We can see that the execution time's curve has nearly the same progression that the theoretical. Moreover, the growth is slightly faster than the local search algorithm which was in $\mathcal{O}(n^3)$, while here we are in $\mathcal{O}(n^4)$.

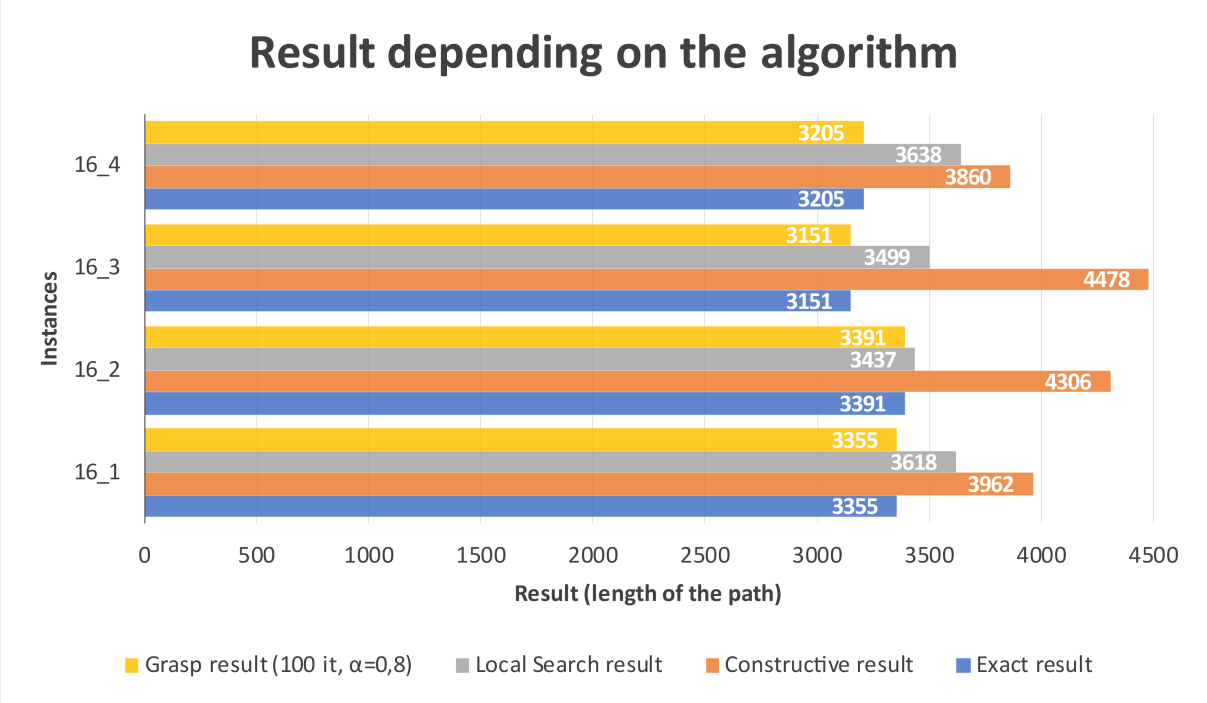
We have built the execution time's curve with the following values that we measured with different instances (*Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz*) :

Instance	**5**	**11**	**17**	**51**
Real Time	0,0014728	0,0020692	0,0034211	0,0314014
Instance	**52**	**100**	**101**	**127**
Real Time	0,0335294	0,222942	0,228171	0,525834
Instance	**280**	**439**	**654**	**783**
Real Time	6,4265	36,2629	128,94	260,24

New set of instances and conclusion

In order to compare performances and the quality of the solutions obtained, we generated 4 random instances containing 16 vertices with a hand-made function (function $createRandom\ Graph()$ can be found in $generalFuncs.h$). We limited ourselves to 16 vertices in order to be able to calculate the exact path.

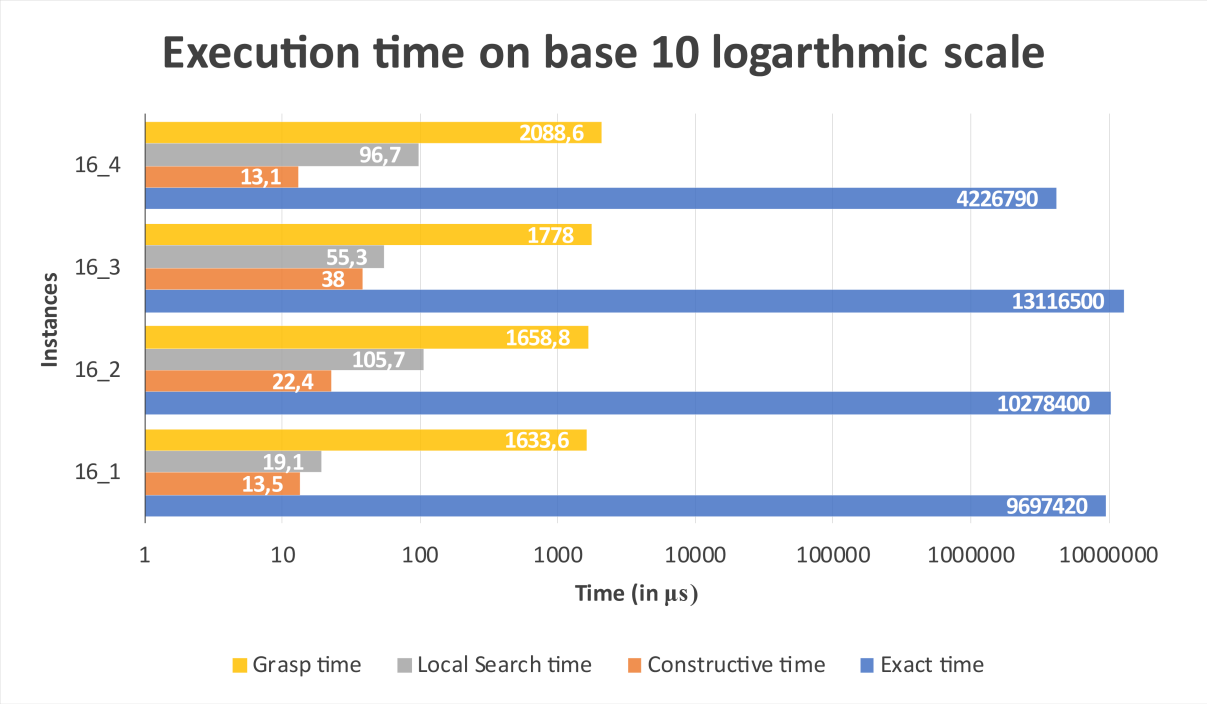
In this first graphic, we compare the quality of the solution (lower is better):



As you can see, with these instances, grasp can give the optimal path. Below, we detail the error rate between the exact path and the constructive algorithm, and the local search algorithm. (Grasp is not include because it has the same result as the exact algorithm, so an error rate of 0%):

Random Instances	16_1	16_2	16_3	16_4		Mean (%)
Constuct / Exact (%)	15,3	21,2	29,6	17,0		20,8
Local / Exact (%)	7,3	1,3	9,9	11,9		7,6

Then, we compare the execution time between each algorithm. To present the values, we used a base 10 logarithmic scale. Indeed, we have some very small results compared to others requiring this type of scale. In other words, the execution time of the constructive, local search and grasp algorithms is 99% faster than the exact algorithm (lower is better):



To conclude, constructive algorithm gives the worst results but in the shortest time. Local search algorithm is a good compromise, because it gives good results in a short time, close to the constructive algorithm.
Exact algorithm can be useful in order to know the best possible path for a graph, but takes too many time from a number of vertices. In order to have good result in a respectable time, we recommend to use grasp. The found result is generally better than local search algorithm.

Here, a small summary of the total path lengths found with the constructive, local search and grasp algorithms for different instances. We also compare the percentage difference between constructive, local search and grasp:

Instance	Constructive	Local Search	Grasp	Constru / Grasp (%)	Local / Grasp (%)
17	2187	2085	2085	4,7	0,0
51	511	438	427	16,4	2,5
52	8980	7967	7542	16,0	5,3
100	27807	22437	21282	23,5	5,1
101	803	658	638	20,5	3,0
127	135737	122097	118937	12,4	2,6
280	3157	2767	2681	15,1	3,1
439	131281	113210	109631	16,5	3,2
654	43457	35422	35150	19,1	0,8
783	11054	9336	9280	16,0	0,6