



In this short paper, I'm dealing with the inverse of the Travelling Salesman Problem because I yearn to get the longest path to reach each city only one time. This is a project that we had been facing during the 4<sup>th</sup> year of the engineering cycle and I decided to share my solution. The first basic method uses a brute force algorithm while the second is a local search algorithm. As they have different complexities, I don't use the same number of cities for the both. I'm rapidly tackling the 2-opt algorithm, I only wrote out the pseudo-code. Then I'm quickly reviewing the Minimum Spanning Tree with the Prim's Algorithm.

## Introduction

### 1°) Reading the .csv file

I created the void « fileReader » that takes in argument the filename, it reads the file line by line (except the first which is a description line) and create an object city at each line and instantiates it. Each city created is inserted in a vector of city. A city has 4 attributes: name, latitude, longitude and its index in the vector (first city inserted in the vector has an index value of 0).

### 2°) Data structure for the graph

Although the matrix isn't an optimal choice for memory space optimisation, it sounded easier to me for the manipulation. So I created a pointer of pointer that contains the distance between all cities whose

distance is greater than 100km. As an object city has a position attribute, we can access the distance between two cities easily thanks to the matrix.

## **Maximum length path algorithms**

**The aim is here to get the longest path starting from Paris and finishing at Paris.**

### **1°) Brute force algorithm**

This very greedy algorithm tests all the possible paths and gives back the longest. The complexity is very easy to compute, from the starting city you have  $n-1$  possibilities, then from the second you have  $n-2$  possibilities etc... Therefore, you have  $(n-1)!$  paths possible.  $((n-1)!)/2$  to be more accurate, because the graph is undirected. It takes  $O(n)$  to get the distance (we go through the matrix  $n$  times, but we exactly know where we have to get each distance). So the complexity of this algorithm is  $O(n!)$ .

The function `BruteForceAlgorithm` is the function that tests all the path. For practical reasons, we created a second .csv with less cities (if we test this algorithm on the 72 cities, it would take  $71!$  path to generate, it would take years and years). Inside we create a first path “randomly” in order to have a reference to compare with. The solution path is a vector of integers; all these integers are the index of the cities.

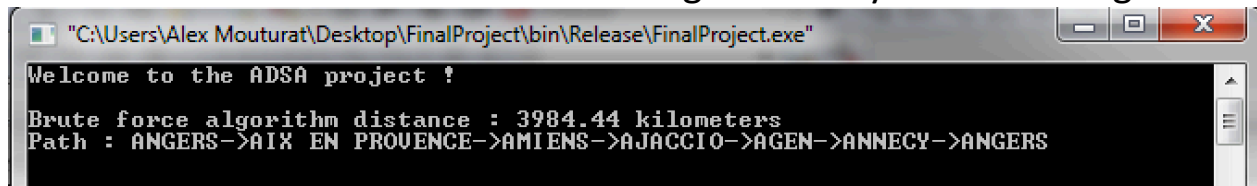
Then a function `Next_permutation` is a function from the library `<algorithm>` that takes the starting index and the final index of the vector that you want to permute and gives back all the possible permutations. We order this function to permute city from the second city (because we want the starting city to be always Paris) to the penultimate (because we want the finishing city to be always Paris).

Each path created is compared with the reference and saved if the distance is longer. Once all paths are tested, it returns the longest.

As we were sceptic for the Next\_permutation function, we checked if all the permutations were done:

We have 120 permutations as planned (our path contains 6 cities and the first cities is the same as the arrival city).

This is the final result given by the algorithm:

A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Users\Alex Mouturat\Desktop\FinalProject\bin\Release\FinalProject.exe". The window contains the following text:

```
Welcome to the ADSA project !  
Brute force algorithm distance : 3984.44 kilometers  
Path : ANGERS->AIX EN PROUVENCE->AMIENS->AJACCIO->AGEN->ANNECY->ANGERS
```

## 2) Local search Algorithm

This heuristic algorithm gives an efficient solution, not the best. The idea is to find the furthest neighbour of the current city and add it to the longest path and repeat for each city.

This algorithm is pretty easy to implement. The algorithm starts by inserting Paris in the longest path vector. Then it iteratively finds the maximum in the matrix for the corresponding row to Paris. Once it's found it adds the index of the column (with correspond to one city position) in the longest path vector. And it repeats until all cities are inserted.

In term of complexity, find the maximum for a row is  $O(n)$  and we repeat  $n$  time this operation (for each city that we insert), we are at  $O(n^2)$ . When a maximum distance between two cities is found, the algorithm checks if the city isn't already inserted in the longest path vector that takes  $O(n)$  in the worst case (when almost all the cities are inserted in the vector). Therefore, the complexity of the algorithm is  $O(n^3)$ . This complexity could be improved by another way to verify if the city is already inserted in the vector solution. A way to do that is to implement a boolean array with all the values initialized with false. Each time a city is inserted in the vector solution, we switched to a true value at the index of the city. This step only takes  $O(1)$  and would improve the effectiveness of the algorithm.

This is the approximate solution of the TSP using a local algorithm :

```
Heuristic local algorithm distance : 194363 kilometers
Path : PARIS->SAINT GEORGES->LE HAURE->SAINT LAURENT DU MARONI->CAEN->ROURA->CALAIS->IRACOUBO->DIEPPE->CAYENNE->ROUEN->KOUROU->LE MANS->SINNAMARY->CHERBOURG->FORT DE FRANCE->ANGERS->POINTE A PITRE->POITIER->BASSE TERRE->TOURS->SAINT BENOIT->TARBES->SAINT DENIS->BORDEAUX->DZAUDZI->AMIENS->AJACCIO->LILLE->BASTIA->ARRAS->PERPIGNAN->METZ->BIARRITZ->STRASBOURG->TOULOUSE->NANCY->AGEN->MULHOUSE->BEZIERS->REIMS->TOULON->VERSAILLES->NICE->RENNES->MARSEILLE->MELUN->AIX EN PROVENCE->SAINT BRIEUC->MONTPELLIER->BREST->NIMES->TROYES->BRIVE->BESANCON->LIMOGES->ANNECY->ORLEANS->GRENOBLE->NANTES->LYON->LA ROCHELLE->DIJON->CLERMONT FERRAND->AUXERRE->SAINT ETIENNE->LORIENT->ROANNE->BOURGES->VICHY->AGEN->PARIS
```

### 3) Lin-Kernighan algorithm

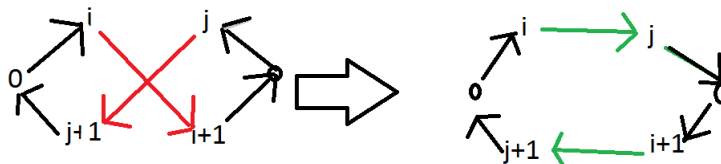
Lin-Kernighan is a general heuristic algorithm for the travelling salesman problem known as k-opt heuristic algorithm. We use a particular case from this algorithm, the simplest one, 2-opt algorithm.

This explains briefly how it works:

- Generate k random initial tours (city permutations)
- Iteratively improve tours until local maximum reached

For each iteration, apply best possible 2-opt change:

- Find best pair of edges  $(i, i+1)$  and  $(j, j+1)$  such that replacing them with  $(i, j)$  and  $(i+1, j+1)$  maximizes our length.



*\*\*\*Here is the pseudo code:*

Step 1: Generate a random path

Step 2: Calculate the total distance of that path. This distance will be set as the best distance for the moment

Step 3: Split the path in two other path

Step 4: Make the 2-opt switch between these path and then calculate the distance of this new path

Step 5: If the new path is longer than the old path then go back to step 3. Else if the improvement has been made, we stop.

*\*\*\*Complexity of the algorithm:*

-  $O(n-1)$  is the complexity to get the first part of the path because at the maximum we can take all the cities in the whole path without the penultimate city otherwise the path wouldn't have changed.

-  $O(n-1)$  is also the complexity to get the rest of the path.

- So  $O((n-1)^2)$  is the total complexity to get our new longest path.

-  $O(1)$  is the complexity to compute the improvement. Actually we only compare the difference of length between our starting path and the new path which include the switch.

-  $O(n)$  for the number of iterations we can execute before reaching the local maximum.

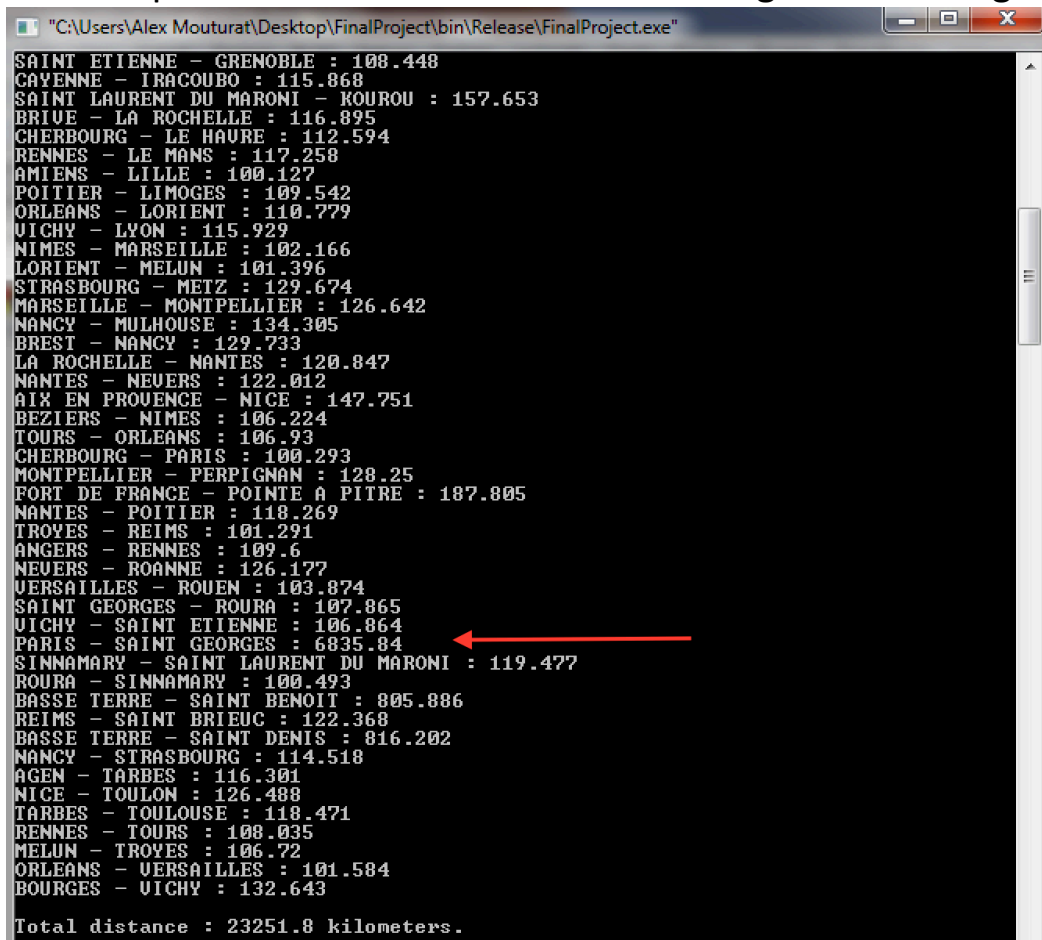
So the total complexity is  $O(n^3)$  with this approach.

## Minimum Spanning Tree

The aim of this part is to find the MST of this graph with the edge Paris – Saint Georges.

To finish this little project, I choose to solve the Minimum Spanning Tree with the renowned Prim's Algorithm. The complexity of this algorithm is  $O(n^2)$ . However, a constrain is added. The minimum spanning tree has to contains the edge: Paris -> Saint Georges. My strategy is simple, we compute the MST through the algorithm as usual but at the end, we just change the parent of Saint Georges by Paris.

This is a part of the whole MST with the edge Saint-Georges – Paris:



```
"C:\Users\Alex Mouturat\Desktop\FinalProject\bin\Release\FinalProject.exe"
SAINT ETIENNE - GRENOBLE : 108.448
CAYENNE - IRACOUBO : 115.868
SAINT LAURENT DU MARONI - KOUROU : 157.653
BRIUE - LA ROCHELLE : 116.895
CHERBOURG - LE HAURE : 112.594
RENNES - LE MANS : 117.258
AMIENS - LILLE : 100.127
POITIER - LIMOGES : 109.542
ORLEANS - LORIENT : 110.779
VICHY - LYON : 115.929
NIMES - MARSEILLE : 102.166
LORIENT - MELUN : 101.396
STRASBOURG - METZ : 129.674
MARSEILLE - MONTPELLIER : 126.642
NANCY - MULHOUSE : 134.305
BREST - NANCY : 129.733
LA ROCHELLE - NANTES : 120.847
NANTES - NEUVERS : 122.012
AIX EN PROVENCE - NICE : 147.751
BEZIERS - NIMES : 106.224
TOURS - ORLEANS : 106.93
CHERBOURG - PARIS : 100.293
MONTPELLIER - PERPIGNAN : 128.25
FORT DE FRANCE - POINTE A PITRE : 187.805
NANTES - POITIER : 118.269
TROYES - REIMS : 101.291
ANGERS - RENNES : 109.6
NEUVERS - ROANNE : 126.177
VERSAILLES - ROUEN : 103.874
SAINT GEORGES - ROURA : 107.865
VICHY - SAINT ETIENNE : 106.864
PARIS - SAINT GEORGES : 6835.84
SINNAMARY - SAINT LAURENT DU MARONI : 119.477
ROURA - SINNAMARY : 100.493
BASSE TERRE - SAINT BENOIT : 805.886
REIMS - SAINT BRIEUC : 122.368
BASSE TERRE - SAINT DENIS : 816.202
NANCY - STRASBOURG : 114.518
AGEN - TARBES : 116.301
NICE - TOULON : 126.488
TARBES - TOULOUSE : 118.471
RENNES - TOURS : 108.035
MELUN - TROYES : 106.72
ORLEANS - VERSAILLES : 101.584
BOURGES - VICHY : 132.643
Total distance : 23251.8 kilometers.
```

## **Additional functions created for the project:**

void Split(vector<string> &vectorr, string line, char separator): Function that split a string and put all the blocks into the given vector of string

Double distance(city A, city B) : Function that returns the distance between two cities. The value is computed with their latitudes and longitudes

void AddDistanceMatrix(double\*\* distance, vector<city> cities) : This void implement all the distance between each cities in the matrix "distance".

void DeleteMatrix(double\*\* distance, int taille) : this void cleans up the memory space dedicated to the pointer of pointer

Double GetDistancePath(vector<int> path, double\*\* distance) : This function compute the distance total of the path.

vector<int> PathGenerator(string cityDeparture, vector<city> cities) : This function creates a path starting from the citydeparture and going through all the cities.

void DisplayPathName(vector<int> path, vector<city> cities) : This void display on the console the given path. It needs the vector of city because the path is a vector of int, these int are the position of the city. Then we need the vector of city to get back their names.

bool IsAlreadyInPath(vector<int> path,int n) : Function that returns true is a city represented by its position is in the path.

int GetPositionCity(string cityName,vector<city> cities) : Function that gives the position attribute of a city given the name of the city.