UCLouvain

epl

École polytechnique de Louvain

# Solving the Traveling Salesman Problem with a locality hypothesis by subdividing the problem

Author: **Alexandre HENNECART**
Supervisor: **Pierre SCHAUS**
Readers: **Vincent LEGAT, Charles PÊCHEUR, Yves DEVILLE**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

# Contents

**1 Introduction** **3**

**2 Thesis objective** **4**

**3 Deep learning approach** **5**
- 3.1 Learning the Travelling Salesperson Problem Requires Rethinking Generalization . . . 5
- 3.2 Objective . . . 5
- 3.3 How does it work? . . . 6
  - 3.3.1 NAR vs AR . . . 8
- 3.4 Results obtained . . . 8
  - 3.4.1 Impact of TSP training size . . . 9
  - 3.4.2 Impact of graph sparsification . . . 9
  - 3.4.3 Comparison between AR and NAR decoder . . . 10
  - 3.4.4 Comparison between solution search and the learning paradigm 11
- 3.5 Limitations . . . 11

**4 The idea for our solution** **12**
- 4.1 What do we want? . . . 12
- 4.2 The idea . . . 12
- 4.3 Initial assumption . . . 13

**5 Implementation** **14**
- 5.1 The pipeline . . . 14
- 5.2 Creating the TSP instance . . . 15
- 5.3 Selection and resolution of small TSPs . . . 16
  - 5.3.1 The 4 methods to select nodes . . . 16
  - 5.3.2 The solvers used to solve the small TSPs . . . 20
- 5.4 Updating the frequency matrix . . . 21
- 5.5 Solution search . . . 22

**6 Initial assumption validation** **24**
- 6.1 Distance - Frequency correlation . . . 24
- 6.2 Local selection vs random selection . . . 25

**7 Evaluation methods** **27**
7.1 What to measure? . . . 27
7.1.1 Optimality gap . . . 27
7.1.2 Time of execution . . . 28
7.2 Which elements should be evaluated? . . . 28
7.2.1 The different methods to select nodes . . . 28
7.2.2 The different parameters of these models . . . 29
7.2.3 The different methods to solve the sub-TSPs . . . 30
7.2.4 The size of the TSP . . . 30
7.3 How to measure? . . . 30

**8 Results** **31**
8.1 Parameters for each method . . . 31
8.1.1 Pareto Front . . . 31
8.1.2 Euclidean KNN . . . 32
8.1.3 Mahalanobis KNN . . . 34
8.1.4 Square . . . 36
8.1.5 Line . . . 38
8.2 Variation based on TSP size . . . 39
8.3 Method used to solve sub-TSP . . . 41
8.3.1 Time of computation . . . 42
8.3.2 Optimality Gap . . . 43
8.3.3 Proportion of same edges . . . 44

**9 Discussion** **45**
9.1 Result analysis . . . 45
9.1.1 Influence of the parameters: . . . 45
9.1.2 Influence of the TSP size . . . 47
9.1.3 Influence of the method used to solve the sub-TSP . . . 48
9.2 Comparison against the deep learning approach . . . 48
9.3 Other advantages of the solution . . . 49

**10 Conclusion** **50**

**11 GitHub** **51**

# Chapter 1

# Introduction

The traveling salesman problem is a well-known problem in theoretical computer science. It consists of finding the shortest path connecting a set of cities/points in a 2-dimensional space. This problem is very interesting because it is NP-hard, which means that its resolution time is exponential depending on the number of points to be considered. Indeed, a naive approach would consist of testing all possible paths and taking the shortest path thus found, this is a valid solution for very small instances of the problem (less than 10 cities/points), but when we are looking for slightly larger problems, the number of possible paths explodes (there are $6,082 * 10^{16}$ possible paths for a TSP with 20 points!), so the calculation time also explodes and it is very quickly necessary to wait for years before finding the optimal solution for bigger TSPs with this naive approach!
It is therefore necessary to find algorithms making it possible to obtain a solution sufficiently close to the optimal solution while maintaining an acceptable calculation time for practical use (no one wants to wait several hours before their GPS gives the route to follow by example). This search for a local solution in reasonable time is a very important subject in theoretical computer science and it is in this direction that this Master's thesis takes place.
Let's now express the traveling salesman problem in a more mathematical way. First, the cities/points to be connected together are called *nodes* and these nodes have 2 coordinate, one for X axis and one for the Y. For example the 5th node of a TSP is called $N_5$. The paths from one node to another are called *edges* and each edge has a corresponding length which represents the euclidean distance between 2 nodes. In order to represent a node, we call $E_{ij}$ the node that goes from $N_i$ to $N_j$. The edge's lengths are "stored" in a matrix called $distance_matrix$. Most of the usual methods to solve the TSP problem use the distance matrix.

# Chapter 2

# Thesis objective

It should first be remembered that there are already many different solution methods to solve this problem (including the very famous solution proposed by William J. Cook et al: the Concorde solution [1] which is to date the best solver available) and that the the objective of this Master's thesis is not to revolutionize theoretical computer science by finding a better solution, but to explore other avenues to see if there is an interesting idea.
And, as said in the introduction, the majority of existing methods for solving the traveling salesman problem are based on Euclidean distances between different nodes. In this Master's thesis, the objective is to adopt another approach: find the probability that an edge is in the optimal solution and solve the problem no longer thanks to the distance between the nodes, but thanks to a computed weights associated to each edges.
To achieve this objective, two approaches will be compared in this master thesis. The first comes from the "Learning the Travelling Salesperson Problem Requires Rethinking Generalization" article written by Chaitanya K. Joshi, Quentin Cappart, Louis-Martin Rousseau and Thomas Laurent [4] which uses an approach using deep learning (see Chapter 3), and the second is statistical approach created during this Master thesis which consists of reducing the problem into a set of sub-problems that are simpler and quicker to solve (see Chapter 5 for explanation on how it works).

# Chapter 3

# Deep learning approach

## 3.1 Learning the Travelling Salesperson Problem Requires Rethinking Generalization

At the beginning of this master's thesis, the main goal was to find a way to solve a TSP problem using a deep neural network using reinforcement learning. We therefore looked for leads in the existing literature, and the best lead found on this subject was the paper "Learning the Travelling Salesperson Problem Requires Rethinking Generalization" written by Chaitanya K. Joshi, Quentin Cappart, Louis-Martin Rousseau and Thomas Laurent [4].
In this paper, the authors seek to implement a way to solve TSP problems using a pipeline including a deep neural network.

## 3.2 Objective

As the authors of the paper explain, their goal is to create a model that can quickly provide a good quality solution to a TSP problem. But like any model, it needs to be trained and generally only works on instances similar to those provided during training. This is why the authors are looking for a way to train their model on small instances (20 to 50 nodes) and to be able to generalize to larger instances (200 nodes).
Indeed, the larger the training instances, the longer the training time will be (as indicated in the paper, training a model on instances of 50 nodes took 40 hours where training on 200 nodes took 495 hours), so having a model capable of generalizing on greater instance with a small training phase is very interesting.

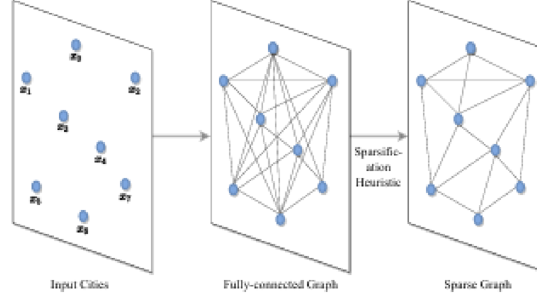
## 3.3 How does it work?

The pipeline proposed by the authors (shown in the Figure 3.1 below) is divided into 5 parts :

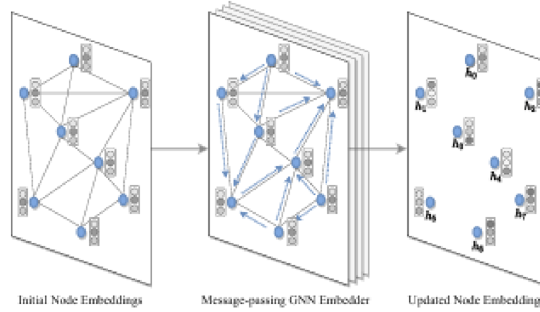
- Problem Definition: In this part, all the elements necessary to resolve the TSP are defined (position of cities, distances between them, method of calculating/evaluating the final tour, etc.). It is also in this part that the authors do graph sparsification (rather than searching on all the edges, they will only keep part of them, keeping only the edges between nodes close to each others) in order to decrease the number of possibility and so, decrease the computation time.
- Graph Embedding: It is in this part that the authors encode the cities into the neural network via a Graph Neural Network (GNN) encoder.
- Solution Decoding : In this part, the authors generate a probability matrix from the neural network using 2 different methods, non-autoregressive decoding (NAR) and autoregressive decoding (AR). The probability matrix thus obtained gives the probability for each edge of belonging to the final solution.
- Solution Search: After obtaining the probability matrix, we must find a final solution. To do this, the authors use a local search on the probabilities rather than on the initial distances. This local search starts from a first solution found on the probability thanks to a greedy search or a beam search.
- Policy Learning: To train the model, the authors use 2 methods, either supervised learning or reinforced learning.

The main point in this paper is that instead of having a deep neural network model which directly gives a solution, the model gives a probability for each edge to belong to a final solution of good quality. And then a simple local search allows you to find the optimal solution with these probabilities.

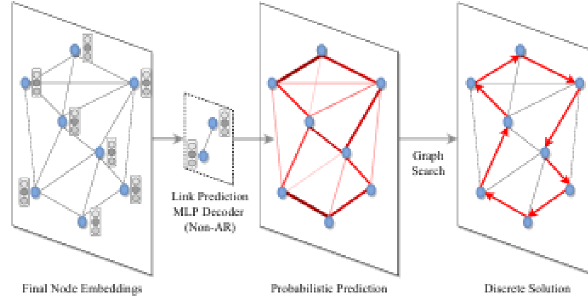
(a) Neural combinatorial optimization pipeline in stages.




(b) **Problem Definition:** TSP is formulated via a fully-connected graph of cities/nodes. The graph can be sparsified via heuristics such as $k$-nearest neighbors.




(c) **Graph Embedding:** Embeddings for each graph node are obtained using a Graph Neural Network encoder, which builds local structural features.




(d) **Solution Decoding & Search:** Probabilities are assigned to each node for belonging to the solution set, either independent of one-another (*i.e.* Non-autoregressive decoding) or conditionally through graph traversal (*i.e.* Autoregressive decoding). The predicted probabilities are converted into discrete decisions through classical graph search techniques such as greedy search or beam search.

Figure 3.1: End-to-end neural combinatorial optimization pipeline from the cited paper ("Learning the Travelling Salesperson Problem Requires Rethinking Generalization") [4]

### 3.3.1 NAR vs AR

The 2 methods (NAR and AR) are used to decode the result obtained using the neural network, but using 2 different approaches:

- NAR : The non-autoregressive decoding method decodes each edge independently of each other, in other words, the result (the weight of the edge in the probability matrix) is obtained in parallel for each edge.
- AR : The autoregressive decoding method uses the results obtained previously to find the weight of the following edges. In other words, the weight decodings of each edge are related to each other.

## 3.4 Results obtained

First, let's explain how these results were measured.
As the two main points are the execution time and the accuracy of the model, it will be on these 2 metrics that the model will be evaluated. But, as the model has several parameters, it is necessary to make measurements for each one.
The first point to measure is the generalization capacity of the model on larger instances. The authors therefore measured their model according to different training sets. To do this, they defined different TSP sizes (20, 50, 100 and 200 nodes) and measured the performance of their models after training on 1,280,000 instances of each size (so 4 times in total) in order to see which size of TSP training instance will allows a better generalization of the problem. They also evaluated a model trained on 1,280,000 instances of varying sizes ranging between 20 and 50 nodes.
The second point to measure is the impact of the model's hyperparameters and there are 3 of them:

- The graph sparsification (how much edges do they need to keep)
- The comparison between AR and NAR decoder
- The comparison between the 2 solution search (greedy search and beam search)
- The impact of the learning paradigm

In order to evaluate their model, the authors compare it against a simple, non-learnt farthest insertion heuristic baseline (this heuristic is explained here : [8])
Now, let's talk about the results obtained in the paper as well as the conclusions drawn by the authors.

### 3.4.1 Impact of TSP training size

We can see in Figure 3.2 that training on the TSP 20 and 200 generalizes very poorly regardless of the size (except for TSP20 on very small TSP instance). While these on the TSP 100 are very good for large instances. The two others training samples size are roughly better for all TSP instance size with TSP50 slightly better for bigger TSP instance and the one with variable TSP instance size slightly better for smaller TSP instance.

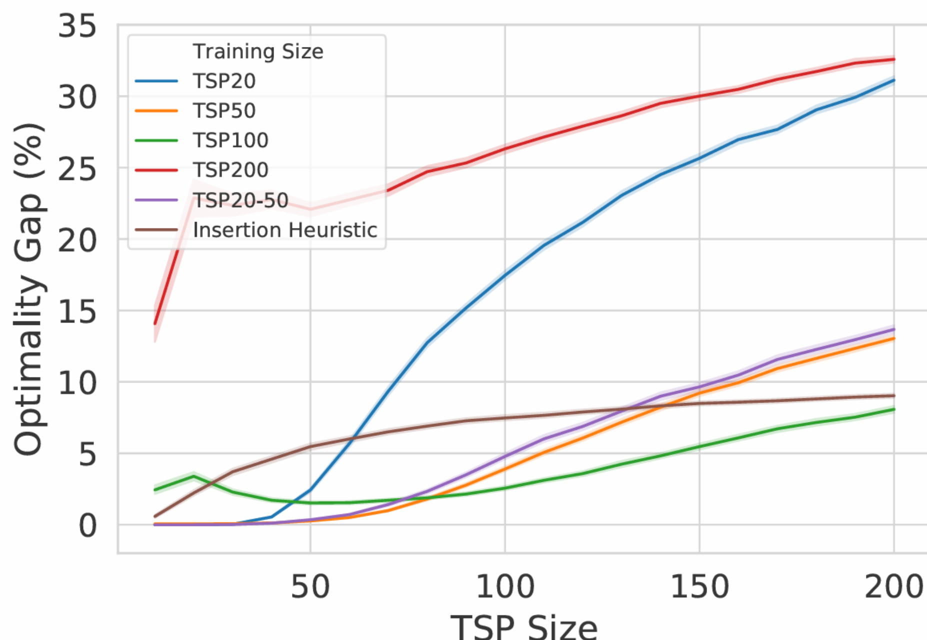



Figure 3.2: Learning from various TSP sizes. The prevalent protocol of evaluation on training sizes overshadows brittle out-of-distribution performance to larger and smaller graphs. (Figure from [4])

### 3.4.2 Impact of graph sparsification

The impact of graph sparsification is shown on Figure 3.3, with 2 different KNN (K-Nearest Neighbors) methods with 2 variants for each. The fist method is to only keep a fixed number of nearest node (the 10 or 20 nearest) and the second one keep a fraction of all edges (keeping only the edges connected to the 20% or 50% of the closest nodes). As we can see, only keeping a fixed amount of node is worst than keeping all node, and using only a fraction of node is not influenced by the chosen fraction (20% or 50%).

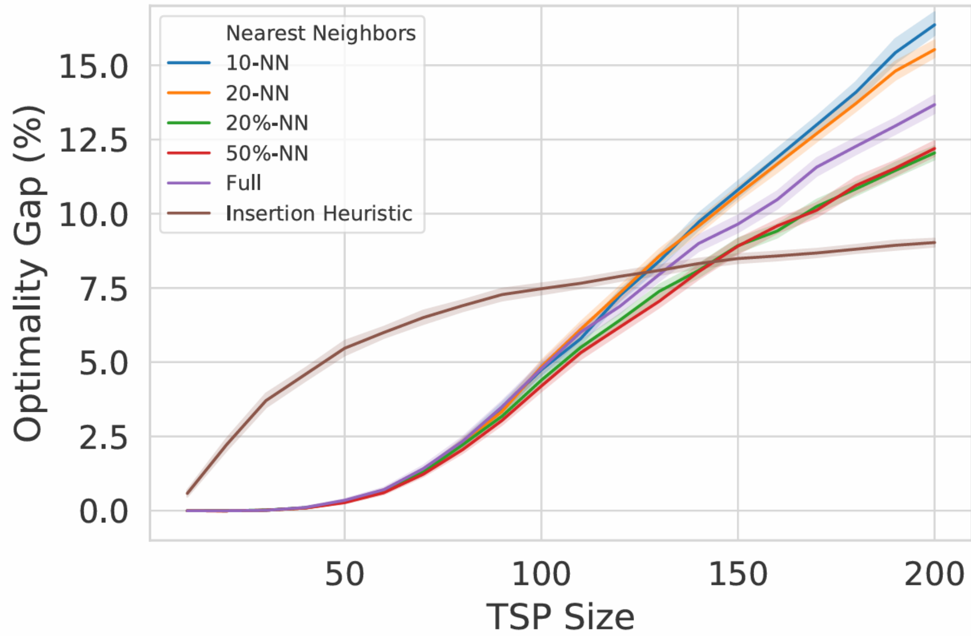



Figure 3.3: Impact of graph sparsification. Maintaining a constant graph diameter across TSP sizes leads to better generalization on larger problems than using full graphs. (Figure from [4])

### 3.4.3 Comparison between AR and NAR decoder

As We can see in Figure 3.4, The NAR decoder is way better than the AR decoder, but it takes much computation time.

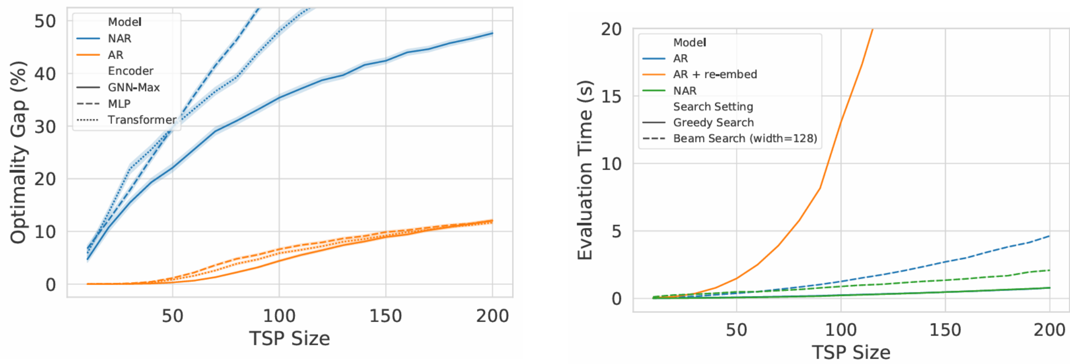



Figure 3.4: Comparing AR and NAR decoders. Sequential AR decoding is a powerful inductive bias for TSP as it enables significantly better generalization, even in the absence of graph structure (MLP encoders). (Figure from [4])

### 3.4.4 Comparison between solution search and the learning paradigm

The Figure 3.5 show us that RL models outperform SL model, except when the SL model use a beam search.

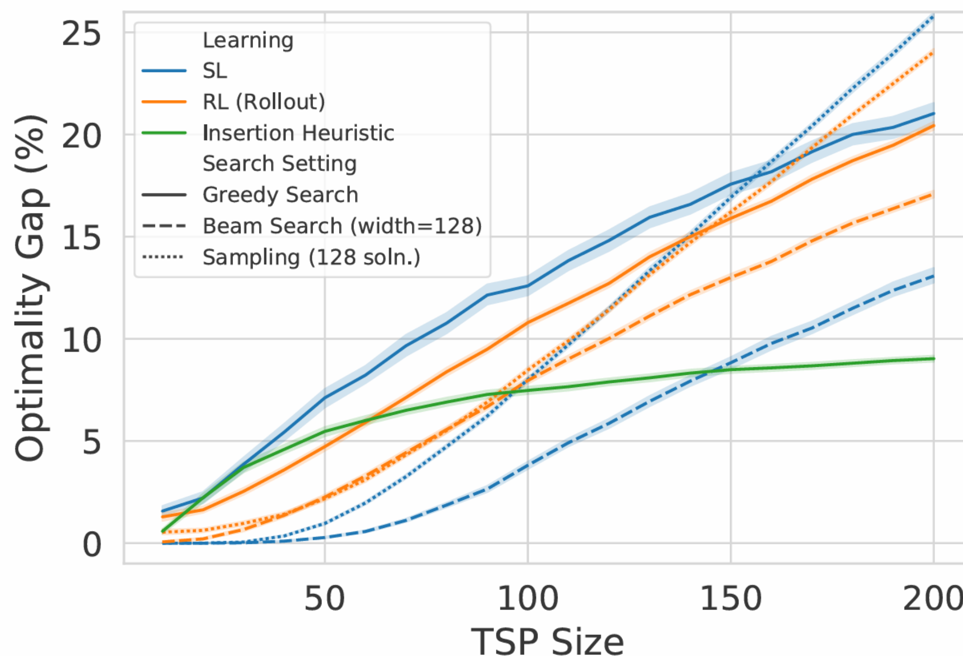



Figure 3.5: Comparing solution search settings. Under greedy decoding, RL demonstrates better performance and generalization. Conversely, SL models improve over their RL counterparts when performing beam search or sampling. (Figure from [4])

## 3.5 Limitations

But, as the authors point out, this approach has many limitations. The most obvious of these is the time taken for learning the deep neural model which already takes 40 hours for their fastest model to train (the one on instances of variable sizes between 20 and 50 nodes) and can take a lot of time for larger models (495 hours for training on 200 node instances).
Another major limitation is the difficulty in generalizing on instances much larger than 200 nodes. Indeed, the results clearly indicate a degradation in accuracy the larger the test TSP, we can safely assume that this approach is not compatible with instances of up to a thousand nodes.

# Chapter 4

# The idea for our solution

As said previously, a deep-learning approach has many limitations. Would it be possible to adopt a similar approach, but via a fast and easily adaptable algorithm to larger TSP instances?
This question is the heart of this thesis, let's see together how to answer it.

## 4.1 What do we want?

We seek to keep an approach similar to the one proposed in the paper detailed in the previous chapter [4]. This means that we want to find a way to get the probability for each edge to belong to the final solution, then do a local search on these probabilities to obtain a solution.
The important thing to keep in mind is that the method used must be fast while maintaining good accuracy, without going through a long training phase and can be used regardless of the size of the TSP. The most difficult part will obviously be finding the probability that an edge belongs to the final solution (because a local search can always be fast and many very efficient implementations already exist like 2-opt for example).

## 4.2 The idea

It all starts from a simple idea: if we take a small number of nodes from our TSP to make a smaller TSP, and we solve it, it's a safe bet that the edges taken from the small TSP will have a good chance of being taken in the final solution of the large TSP than edges not taken. Thus, by repeating this operation a large number of times, each time taking a small group of different nodes close to each other, it should be possible to find which edges are most often taken and which therefore have the most chance to be in complete TSP solution. In other words, we use the

frequency of presence of an edge in a local sub-solution to say that this edge has a greater probability of belonging to the final solution.

## 4.3 Initial assumption

Given that we are only going to solve sub-TSPs composed of nodes close to each other, we induce a bias in our solution: the solution links the close nodes together without ever linking the nodes far from each other, which means that not all edges have an equal chance of being taken into account in the search for the final solution! Indeed, we make a locality hypothesis by assuming that nodes far from each other will never be connected in the optimal solution while those close to each other have a greater chance of being linked together. This assumption is therefore at the center of this Master's thesis and we will seek to prove its (in)accuracy subsequently.

# Chapter 5

# Implementation

Now let's talk about the implementation put in place to achieve this goal.

## 5.1 The pipeline

First of all, let's talk about the general pipeline of our solution which remains very similar to the one used in the paper 'Learning the Traveling Salesperson Problem Requires Rethinking Generalization' [4]. Here are the different steps included in this pipeline:

1. Creating the TSP instance: all the primary steps necessary to allow our solution to operate on the TSP.

2. Selection and resolution of small TSPs: we select a small number of nodes for which we solve the new sub-TSP and we repeat this step many times with different sets of nodes each time.

3. Updating the frequency matrix: each time the previous step is carried out, we keep the result in memory by updating a frequency matrix which counts the number of times an edge is selected.

4. Solution search: a solution search is subsequently carried out to find a final solution using the frequency matrix obtained in the previous step. The chosen method to perform the solution search is the 2-opt [2].

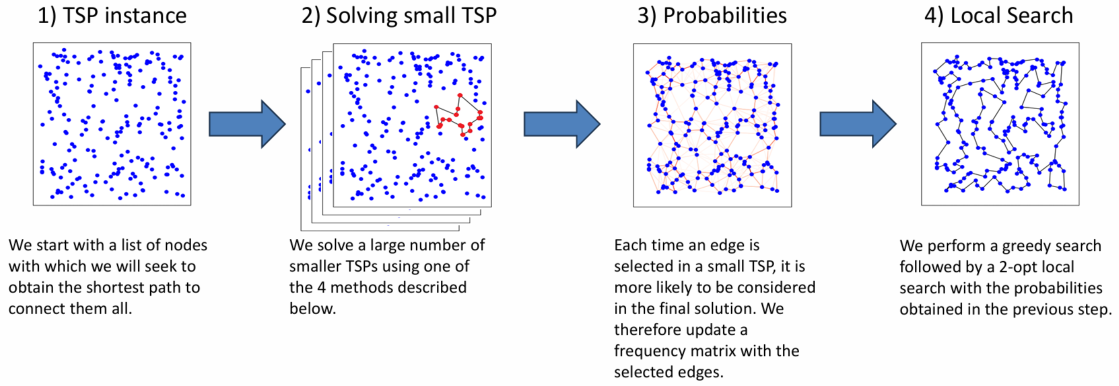



Figure 5.1: Solution pipeline

And the corresponding pseudo code is the following :

**Algorithm 1** Pipeline

$M \leftarrow Size\ of\ the\ full\ TSP$
$N \leftarrow Number\ of\ sub\ TSPs$
$K \leftarrow Size\ of\ sub\ TSPs$
$frequency_matrix \leftarrow M * M\ matrix\ filled\ with\ 0$
$real_distance_matrix \leftarrow M * M\ matrix\ with\ real\ distance\ between\ nodes$
**for** $iteration = 1, 2, \ldots, N$ **do**
  # Get all nearest node from a random node
  $random_node \leftarrow Select\ a\ random\ node$
  $reduce_distance_matrix \leftarrow select_nodes(random_node, K)$
  $tour_nodes \leftarrow solve(reduce_distance_matrix)$
  **for** $edge\ in\ tour_nodes$ **do**
    $i,\ j \leftarrow edge.start,\ edge.end$
    $frequency_matrix[i][j] \leftarrow frequency_matrix[i][j] + 1$
    $frequency_matrix[j][i] \leftarrow frequency_matrix[j][i] + 1$
  **end for**
**end for**
$greedy_solution \leftarrow greedy_search(frequency_matrix)$
$solution \leftarrow 2opt_local_search(greedy_solution, real_distance_matrix)$

## 5.2 Creating the TSP instance

A TSP can be represented by a list of points with $x$ and $y$ coordinates specific to each, but this representation is not very practical for solving the problem. The

solution therefore consists of calculating a distance matrix D between each point in which the entry $D_{ij}$ corresponds to the distance between node $i$ and node $j$. Since the length of the edge going from node $i$ to node $j$ has the same length as the edge going from node $j$ to node $i$, this matrix is therefore a symmetric matrix.

## 5.3 Selection and resolution of small TSPs

The second step is to select a small number of nodes to obtain a simplified version of the problem, because it is much faster to solve a small TSP than a large one since this problem is NP-complete, which means that the time to execution increases exponentially with the number of nodes.

### 5.3.1 The 4 methods to select nodes

Four methods were implemented in order to select these nodes, here is a description of how each of them works. These methods return a reduced distance matrix with only the values of the selected nodes.

#### Euclidean KNN

The first idea is to use a KNN (K-Nearest Neighbors) method to find a subset of nodes. In order to achieve this, we select a node in the TSP problem and take all the $K$ closest nodes to form a sub-problem of size $K + 1$ and do this for all nodes or a part of all nodes. An example is given by the Figure 5.2.

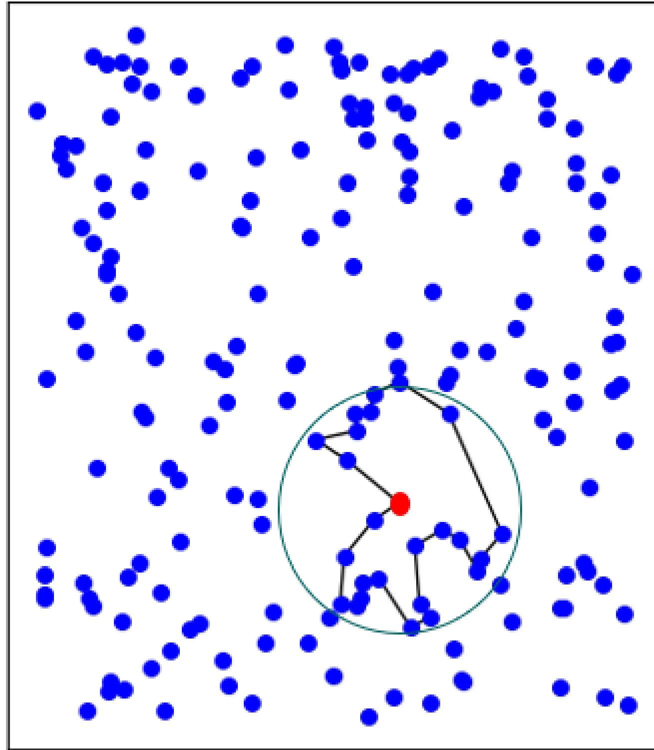


Figure 5.2: Example of node selection (and TSP solving) using the euclidean KNN.

### Mahalanobis KNN

This second method works in a very similar way to the previous one, except that instead of taking the nearest nodes according to a Euclidean distance, we take them according a randomly generated Mahalanobis distance (this is a non-Euclidean distance generated from a randomly generated covariance matrix in our case (see [7] for more information)). The idea here is to take a random nodes as starting node (the same node can be selected multiple time as the starting node), then take the $K$ closest nodes according to the distance generated to obtain a sub-problem of size $K + 1$. The advantage here is that we can take the same node several times as starting nodes since the calculation of the distances is different each time and so, the selected nodes list. This therefore allows us to have more different sub-problems than with a Euclidean distance and, we hope so, improving the solution obtained by doing more sub-TSP solving. An example is given by the Figure 5.3.

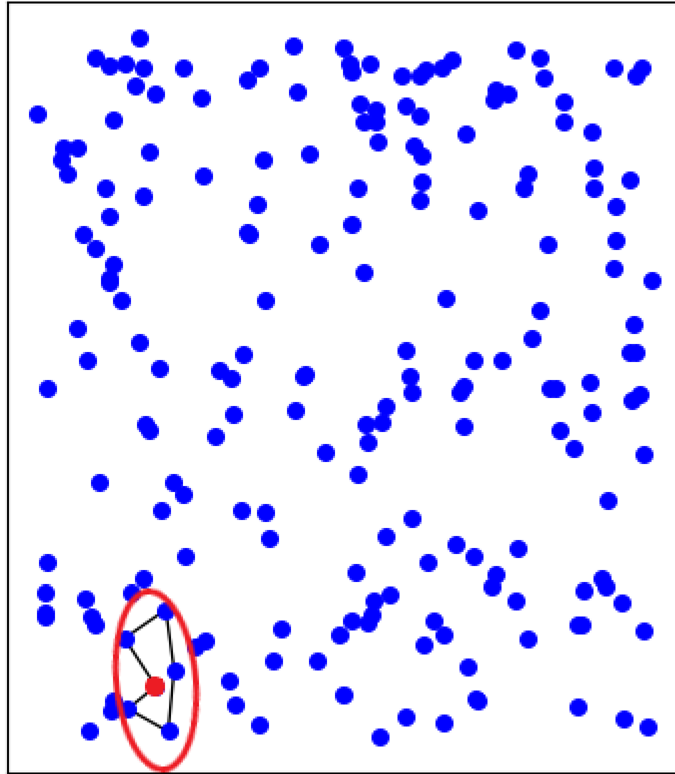


Figure 5.3: Example of node selection (and TSP solving) using the Mahalanobis KNN.

### Square division

Another method of selecting nodes to create the sub-problem is to geometrically divide the problem into areas of equal size as shown in Figure 5.4 (where the problem has been divided into 100 areas containing on average 2 nodes each).
Then we select several of these small areas to create the sub-problem (in Figure 5.4, a 3 by 3 square area has been selected). This step is repeated for all possible combinations to go through all the nodes of the initial problem.
But this method has a drawback, the nodes located on the edges of the problem are less often taken, which leads to an unequal distribution between the nodes. To resolve this problem, a line division (see next point) is carried out for each zone in the edges of the problem.

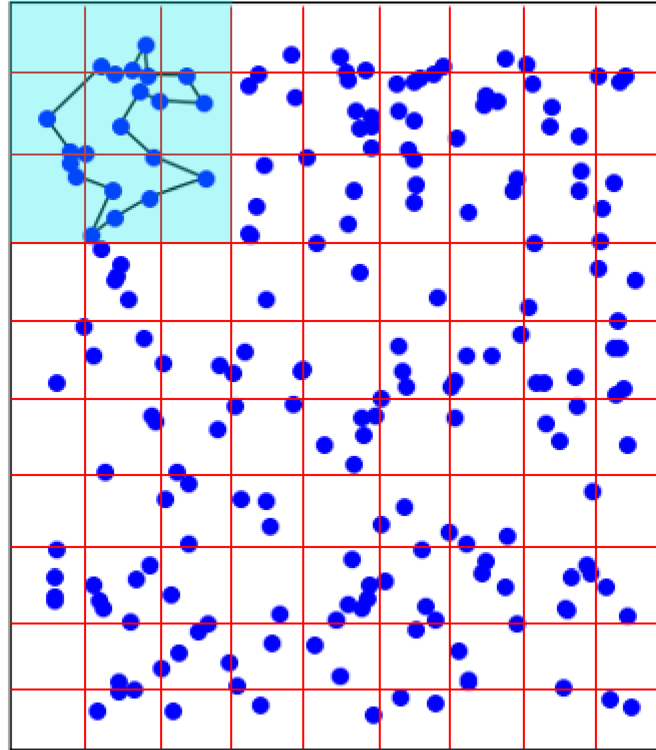


Figure 5.4: Example of node selection (and TSP solving) using the square division.

### Line division

This method uses the basic principles of the previous one: the problem is geometrically divided into areas of equal size, except that this time, instead of selecting a square area, it is an area in the shape of a line going from a edge to edge of the problem with a width of 2 or 3 areas as shown in Figure 5.5. This operation is carried out for all possible combinations, both vertically and horizontally.

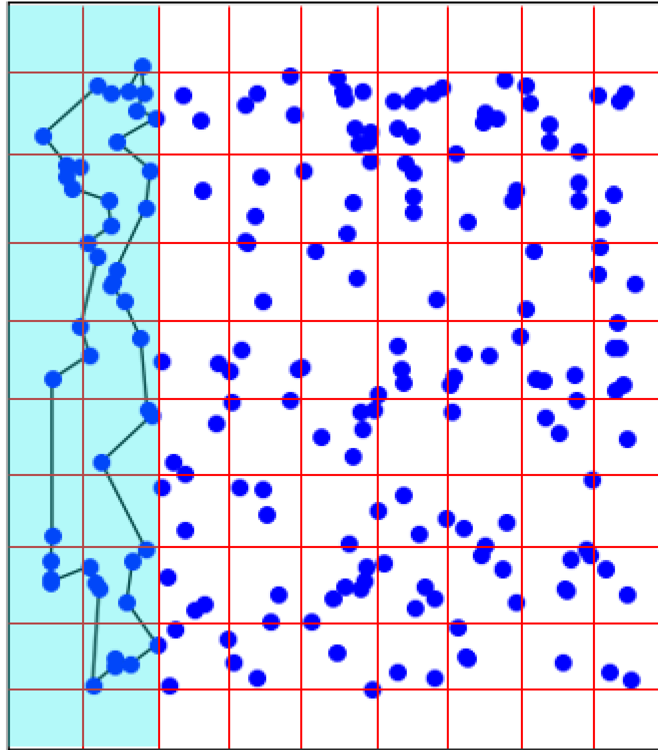


Figure 5.5: Example of node selection (and TSP solving) using the line division.

### 5.3.2 The solvers used to solve the small TSPs

With the reduced distance matrix obtained in the previous point, we can solve the associated sub-TSP via a "classic" method (i.e. an already existing algorithm). For this master's thesis, 3 different TSP resolution algorithms were used. Each one of these resolution methods returns a path obtained with the reduced distance matrix, this path will be used in section 5.4 (*Updating the frequency matrix*).

#### 2-opt local search

The first solution is to use the 2-opt algorithm proposed by Georges A Croes in 1958 [2] to solve the sub-TSP. To be able to use the algorithm in this master's thesis, I implemented it in python.

#### Python-TSP: Simulated Annealing

To have more evaluation methods, the python library python-TSP [3] was used. This library entirely coded in Python is very complete and implements many TSP problem solving methods.

For this method, we used the Simulated Annealing algorithm [10] provided with this library.

**Concorde**

The last method used consists of using the best current solver, the Concorde developer by William J. Cook et al. [1].

## 5.4 Updating the frequency matrix

Once the nodes have been selected and the sub-TSP has been solved, we must keep track of this resolution to be able to obtain the final solution. To do this, a frequency matrix $F$ is created. In this matrix each entry $F_{ij}$ corresponds to the number of times that the edge going from node $i$ to node $j$ (and vice versa) has been selected during all the resolutions of the previous sub-TSPs. The path going from node $i$ to node $j$ being the same as the path going from node $j$ to node $i$, this matrix is also a symmetric matrix.
In Figure 5.6, you can see a visual representation of this frequency matrix $F$ associated with the overall TSP graph after all sub-TSPs have been resolved. Each entry of the matrix $F$ corresponds to an edge between 2 nodes on the graph, if the value of $F_{ij}$ (i.e. the edge between node $i$ and node $j$) is equal to 0 (i.e. it has never been taken in the resolutions of the sub-TSPs), it is not displayed in the graph. On the other hand, the higher the value of an edge in P, the darker it appears and therefore the more likely it is to belong to the final solution.

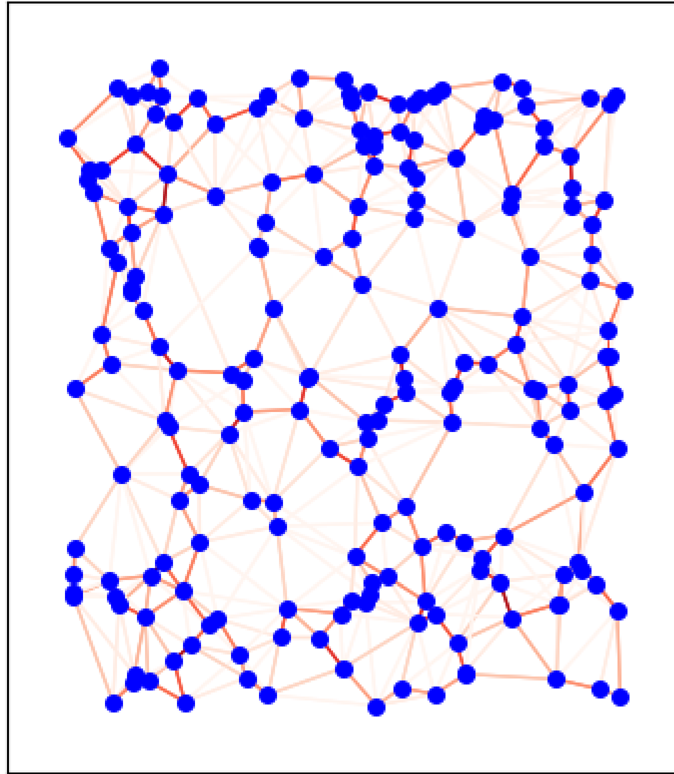


Figure 5.6: Example of edge frequency for a TSP instance.

## 5.5 Solution search

Once the frequency matrix F has been obtained thanks to the previous steps, it is finally time to find the final solution to our problem thanks to a slightly modified greedy search so that it can be carried out with the frequency matrix F followed by a local search using a 2-opt algorithm [2].
In this step, the greedy search is particular, because instead of executing on the distance matrix D, it is carried out with priority on the frequency matrix F. Indeed, after selecting a node A, the algorithm will follow the edge starting from it having the greatest value in the frequency matrix F. And if none of the remaining edges has a frequency greater than 0 (i.e. if all the nodes having been connected to node A during the resolution of the sub -TSP have already been selected in the final tour), the algorithm will choose the shortest edge starting from node A according to the distance matrix D.
You can see below the pseudo code of the greedy-search.

**Algorithm 2** Greedy search with the frequency matrix

```
tour ← [0]
actual_node ← 0
AddItem(tour, actual_node)
for i = 1, 2, ..., N do
    maximum ← −∞
    for j = 1, 2, ..., N do
        if j not in tour then
            if P_{actual_node,j} != 0 && F_{actual_node,j} > maximum then
                maximum ← P_{actual_node,j}
                best_next_node ← j
            else
                if P_{actual_node,j} == 0 && −D_{actual_node,j} > maximum then
                    maximum ← −D_{actual_node,j}
                    best_next_node ← j
                end if
            end if
        end if
    end for
    AddItem(tour, best_next_node)
    actual_node ← best_next_node
end for
```

# Chapter 6

# Initial assumption validation

Now that all algorithms are implemented, we need to do a sanity check in order to know if the initial locality assumption was correct or not. We will do it in 2 steps:

- **Check the correlation between length and frequency of edges**: This check aims to show if there is a correlation between the length of an edge and the frequency with which this edge is found in the sub-TSP solution. This check is required to prove that the model uses the initial assumption.
- **Check against a random selection**: This check aims to show whether, to choose the nodes of the sub-TSPs, it is preferable to base ourselves on the initial assumption or if choosing nodes randomly is better. If choosing node randomly is better, this means that the initial assumption was false.

## 6.1 Distance - Frequency correlation

We need to check if the KNN methods uses the initial assumption, we need to find the correlation between the length of an edge and the frequency with which it is taken in the solutions of the sub-TSPs. Indeed, if the edges connecting 2 distant points are taken as often as those connecting 2 close edges, this means that the model does not follow the initial assumption.
Here is a scatter plot relating the length of an edge to the number of times it is found in the solutions of the sub-TSPs:

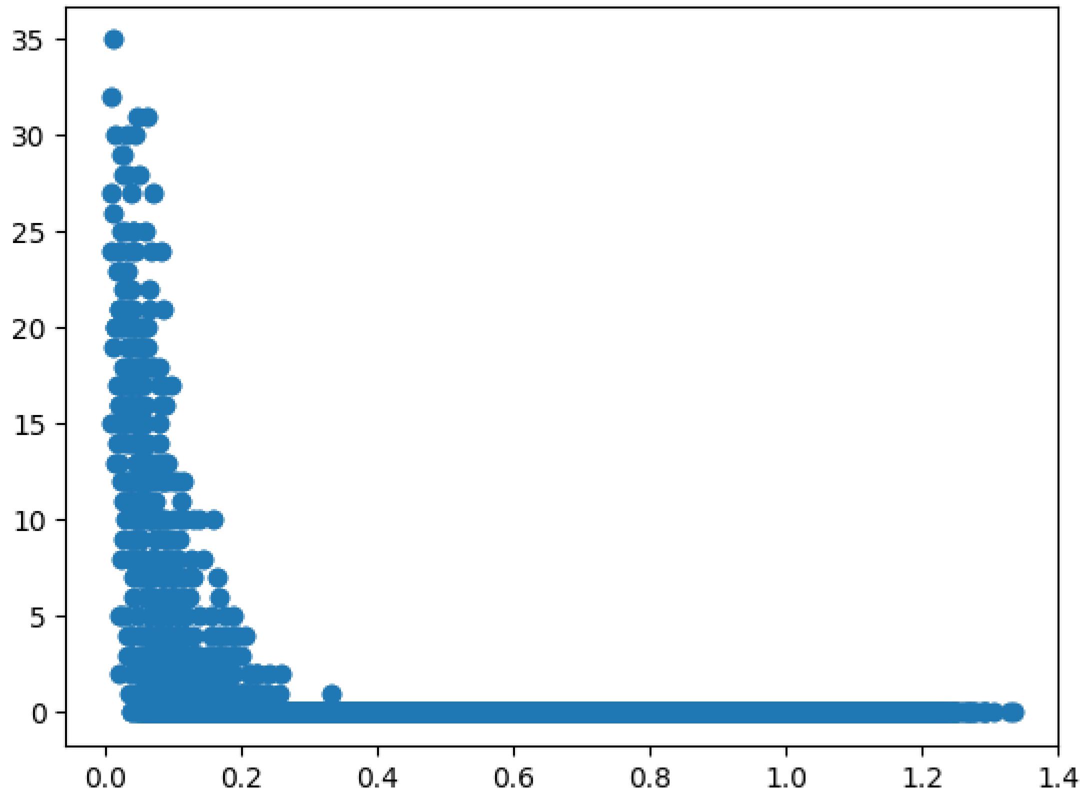



Figure 6.1: Correlation between the length (X axis) and the taken frequency (y axis) of an edge

In Figure 6.1, a very clear correlation can be observed but only in "one direction". Long edges are never taken but that does not means that a short edge is always taken. Indeed, a lot of short edges are not taken in the sub-TSP solution. But only small edges can be taken a lot of times.
This correlation prove that the algorithm respects the hypothesis.

## 6.2 Local selection vs random selection

To verify this point, 100 tests were carried out on 100 different TSP instances having 200 nodes. The values show below are therefore the average of these 100 experiments.
The algorithm chosen was the Euclidean KNN algorithm with a selection of 200 sub-TSPs comprising 25 nodes chosen according to the KNN Euclidian method explained here above (in Section 5.3.1) which was opposed to a similar algorithm having a selection of 200 sub-TSPs comprising 25 nodes randomly chosen from the

200 nodes of the TSP original. The evaluation criterion used to evaluate these two solutions is the optimality gap discussed in the next chapter 7.1.1.
The optimality gap obtained for the Euclidean KNN is 0.084 and this value is 0.119 for the random method, which is therefore a difference of 29% between the Euclidean KNN and the random method, this proves that the local assumption is relevant for the node selection.

To conclude this chapter, the initial hypothesis has been proven to be valid and we can therefore safely move on to testing all our models and their different parameters.

# Chapter 7

# Evaluation methods

Once our algorithms have been implemented, they must be evaluated precisely. It is therefore this evaluation which will be described in this chapter. We will therefore talk about which parameters to vary as well as the size of the TSP instances on which to evaluate our models (all the examples given above were carried out on TSPs with 200 nodes).

## 7.1 What to measure?

Let's first start by giving what criteria to classify the models. In solving TSP type problems, 2 factors are important: the execution time as well as the precision of the solution (does it come close to an optimal solution or not?).

### 7.1.1 Optimality gap

The first important factor in evaluating a model is obviously its accuracy. In order to evaluate this accuracy, we use the optimality gap formula:

$$|obtained_solution - optimal_solution| / optimal_solution$$

Where *obtained_solution* is the length of the tour obtained by the model created for this Master thesis and *optimal_solution* is the length of the best possible tour that can be obtained. This formula gives the difference between our solution and the optimal solution, the closer this value is to 0, the closer the solution obtained is to the optimal solution.
In order to get this optimal solution, we therefore used the Concorde algorithm.

#### Concorde

The Concorde program written by William J. Cook et al. [1] is actually the best existing algorithm for solving TSP problems. It is often used as a baseline to obtain the optimal value of a TSP problem and therefore to compare models (as done in the paper presented in the beginning of this master thesis [4]).
As the program was written in ANSI C, a Python library, PyConcorde [5], was used to act as a wrapper to be able to use this solver in the models.

### 7.1.2 Time of execution

The second important point to evaluate is the execution time taken by the models, because a model that finds a very good solution in an extremely long time is not very useful.
To measure the time taken by a model to obtain a solution, the *time* library supplied with Python was used. The time measured includes the entire pipeline, from data preprocessing to the end of local search. This time is expressed in seconds.

## 7.2 Which elements should be evaluated?

Now, let's talk about the parameters on which we can evaluate the models. Each model has a certain number of parameters which can vary and it is therefore necessary to find under which parameters each model is the most efficient, but they also each have several ways of solving the subTSPs and can be used on TSPs of different sizes.

### 7.2.1 The different methods to select nodes

As said previously, there are 4 methods implemented to separate the overall problem into sub-problems. These four methods are (as a reminder):

- Euclidean KNN: The problem is divided using a KNN according to an Euclidean distance to obtain sub-TSP of size K+1.
- Mahalanobis KNN: The problem is divided using a KNN according to an mahalanobis distance to obtain sub-TSP of size K+1.
- Square division: The problem is geometrically divided in small square, and we take a bigger square made of multiple of these small square.

- Line division: The problem is geometrically divided in small square, and we take a zone from one edge to the other, with a certain width.

These four models need to be compared between each other in order to find which one is the best depending the situation (generally, the size of the TSP problem).

### 7.2.2 The different parameters of these models

But first, it's important to know what the most effective settings are for each model. To do this, several instances of the same models must be evaluated by varying the parameters specific to each.
The parameters to vary for each model are listed below:

#### Euclidean KNN

for the Euclidean KNN, we have 2 parameters to evaluate:

- The size of the sub-TSP: We must vary the K parameters in order to see if a bigger or a smaller K is more efficient.
- The number of sub-TSP: It is interesting to know if a good solution can be obtained via a small number of sub-TSPs, or if we must perform a KNN one times for all nodes as starting nodes.

#### Mahalanobis KNN

for the Mahalanobis KNN, we have 2 parameters to evaluate :

- The size of the sub-TSP: We must vary the K parameters in order to see if a bigger or a smaller K is more efficient.
- The number of sub-TSP: It is interesting to know if a good solution can be obtained via a small number of sub-TSPs, or if we must perform a lot of KNN.

#### Square division

for the Square division, we have 2 parameters to evaluate:

- The number of small squares: We must vary the size (and so on the number) of the small division in order to see if this parameter has an impact on the final solution.
- The size of the selection: We must vary the number of small division taken in our square selection in order to see if a lot of computation with small selection is better or worst of a fewer computation with bigger selection.

**Line division**

for the X=Line division, we have 2 parameters to evaluate:

- The number of small squares: We must vary the size (and so on the number) of the small division in order to see if this parameter has an impact on the final solution.
- The width of the selection: We must vary the number of small division taken as the width of our line selection in order to see if a lot of computation with small width is better or worst of a fewer computation with bigger width.

### 7.2.3 The different methods to solve the sub-TSPs

In addition to all the parameters described previously, it must be kept in mind that each model has several sub-TSP resolution algorithms. This aspect must obviously be evaluated in parallel with the parameters described previously.
As a reminder, here are the 3 methods for resolving sub-TSPs:

- 2-opt local search
- Simulated Annealing
- Concorde

### 7.2.4 The size of the TSP

The last important factor to vary is obviously the size of the TSP problems. Because a model effective on a small TSP will not necessarily be effective on a larger one. This is why the tests will be carried out on instances comprising 100, 200, 400 and 800 nodes.

## 7.3 How to measure?

Performing simple measurements on a single instance is not an optimal way to evaluate a model. This is why each measurement will be carried out 100 times on 100 different TSP instances (but these instances will have the same sizes) and the value that we will keep at the end will be the average of all these measurements. All TSP have a uniform distribution of their nodes and these have their X and Y coordinates between and 1. To measure result for the parameters (section 7.2.2) and the different methods to solve the sub-TSPs (section 7.2.3), we use TSPs with a size of 200 nodes, to measure the impact of the size of the TSP, the size is obviously variable.

# Chapter 8

# Results

In this chapter, we will present the results obtained as well as the associated graphs. We will discuss these results in the next chapter: 9 Discussion.

## 8.1 Parameters for each method

As see in section 7.2.2, the first measurement to be carried out is that relating to the model parameters (for each model) in order to be able to select the best ones for further measurements (it is indeed pointless to continue measurements with non-optimal parameters).
For each model, the results will be given in the form of graph which will be analyzed in the Chapter 9: Discussion and in the form of a Pareto Front.

### 8.1.1 Pareto Front

Let's start by explaining what the Pareto front is and why it is an element that will be crucial for the choice of parameters. As explained in [6], the Pareto front (or Pareto frontier) is a easy visual way to show all relevant solution in a multi-objective optimization problem like the one that currently interests us.
Indeed, we seek to find which parameters to choose so that the models work best according to 2 objectives/metrics: the calculation time and its accuracy (so in this case, we have a "2-objective optimization problem").
When we represent results in a scatter plot according to 2 metrics (in our case), and we seek to find the point which minimizes these 2 metrics, it is very rare to have only one solution, very often a model is good in one metric, but is beaten by another in the other metric.
The Pareto frontier makes it possible to find all the models being optimal, that is

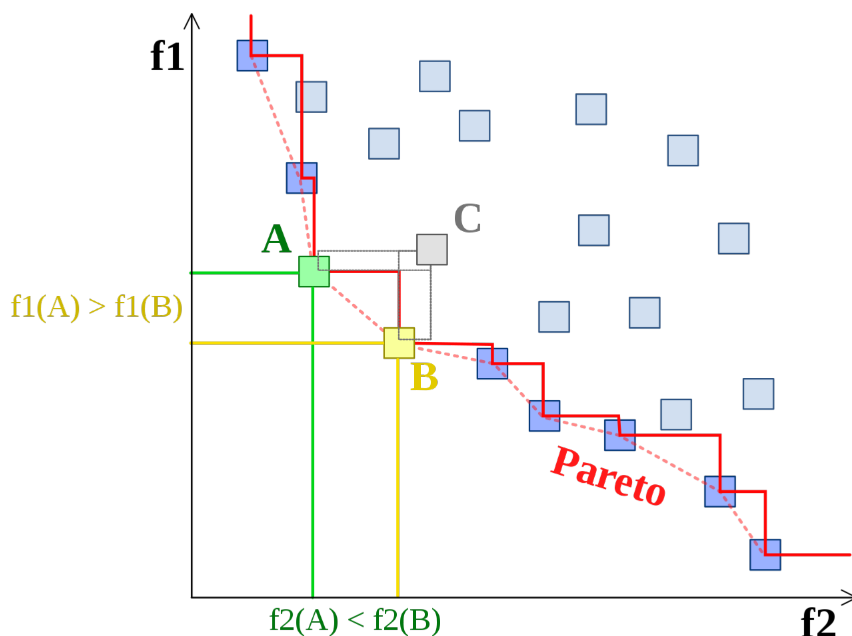



Figure 8.1: Pareto front explanation (comes from [9])

to say which are never beaten in the 2 metrics.
As you can see in Figure 8.1 below, all point in blue are in the Pareto frontier while point in grey are not. This means that only the points in blue are optimal, that for all results represented in gray, there is a result in blue being better in both metrics. Thanks to the Pareto front, the choice of optimal parameters is therefore limited to a small number of optimal possibilities and no longer to all possible combinations. The choice is often based on which metric is most important based on the problem. If we are looking for an acceptable solution for the 2 metrics, the classic method is to take the solution being closest to the origin (point A or B in the Figure 8.1

### 8.1.2 Euclidean KNN

The first model to have been analyzed is that using the K-Nearest Neighbors method with Euclidean geometry. As said in section 7.2.2, this method takes 2 parameters: the size of the sub-TSP and the number of sub-TSP.
Figures 8.2 and 8.3 show respectively the time taken on average and the average difference from the optimal solution.

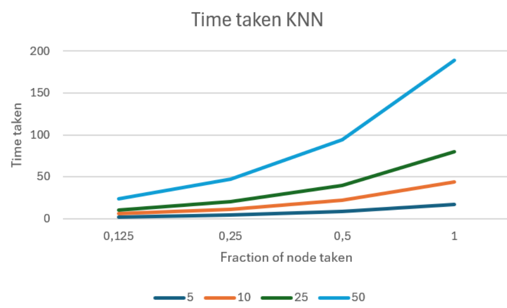



Figure 8.2: Time taken for the Euclidean KNN method

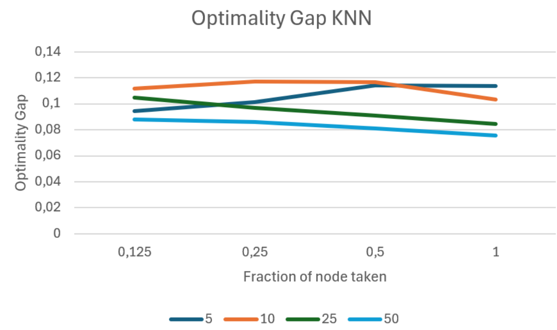



Figure 8.3: Accuracy for the Euclidean KNN method

Figure 8.4 below shows the proportion of edges (on average) that are found in both the solution given by the model and in the final solution. It would seem logical that the more edges of the optimal solution are found in the solution given by the model, the closer the solution of the model will be to the optimal solution and therefore the better the accuracy will be.

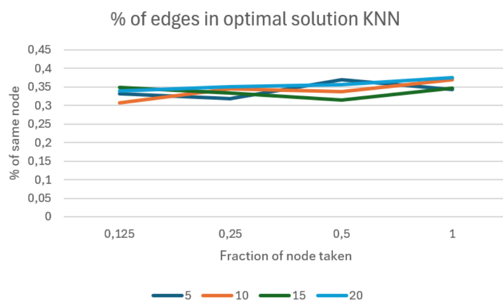



Figure 8.4: Percentage of edges present in the optimal solution

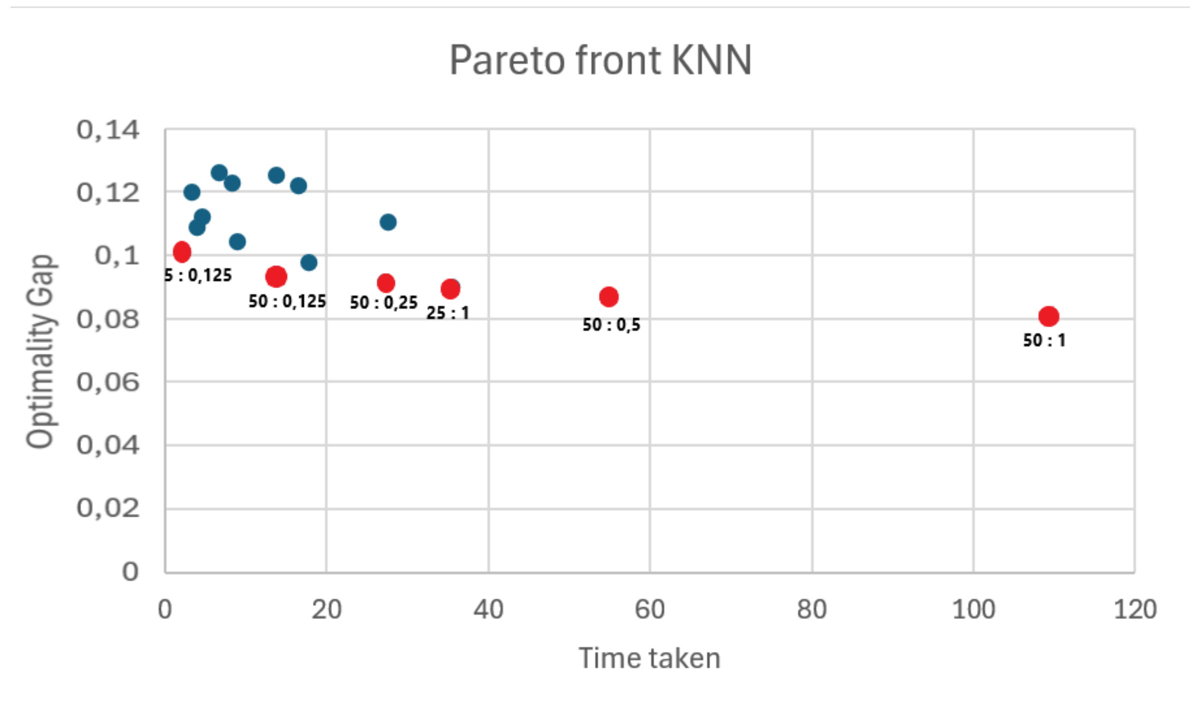



Figure 8.5: Pareto Front for the Euclidean KNN

The Figure 8.5 above show the Pareto Front for the Euclidean KNN method to solve the sub-TSPs, we notice that the best choice to maintain acceptable precision while keeping the calculation time sufficiently short is the solution taking a size of 50 nodes for the sub- TSP and retaining only 12.5% of all nodes.
So we will keep these parameters for the rest of the evaluation for the Euclidean KNN.

### 8.1.3 Mahalanobis KNN

As for the method with Euclidean KNN, here are the results for the KNN using a Mahalanobis distance. Figure 8.6 here below shows the time taken on average, Figure 8.7 shows the average difference with the optimal solution while Figure 8.8 shows the proportion of edges also found in the optimal solution.

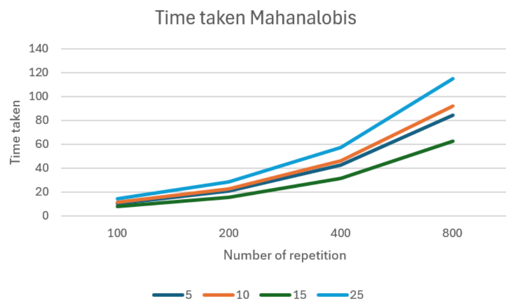



Figure 8.6: Time taken for the Mahalanobis KNN method

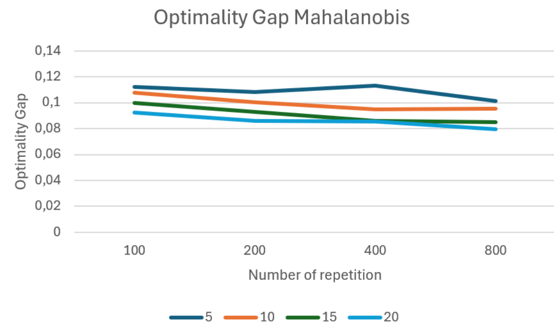



Figure 8.7: Accuracy for the Mahalanobis KNN method

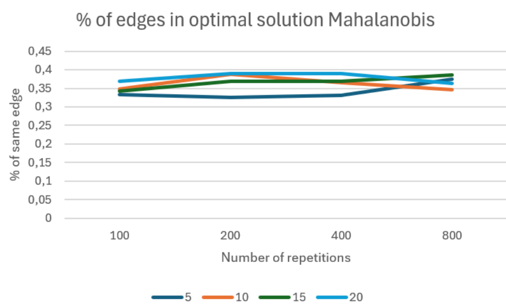



Figure 8.8: Percentage of edges present in the optimal solution

The Figure 8.9 below show the Pareto Front for the Mahalanobis KNN method. For the rest of the evaluations for this method, the size of the sub-TSP will be 20 nodes and the number of repetitions will be 200, because these parameters allow us to maintain a good accuracy with a reasonable computation time.

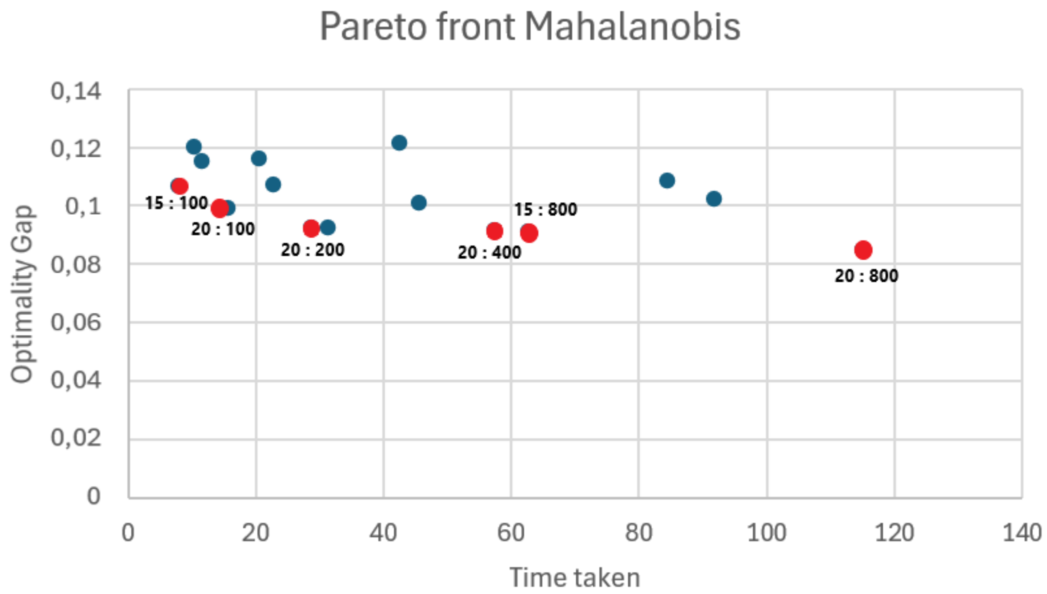



Figure 8.9: Pareto Front for the Mahalanobis KNN

### 8.1.4 Square

As for the previous methods, here are the results for the method using the square division of the nodes. Figure 8.10 here below shows the time taken on average, Figure 8.11 shows the average difference with the optimal solution while Figure 8.12 shows the proportion of edges also found in the optimal solution.

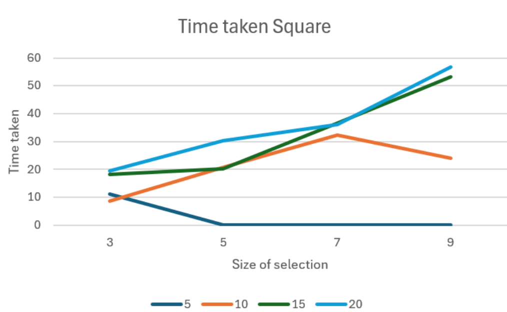



Figure 8.10: Time taken for the Square division method

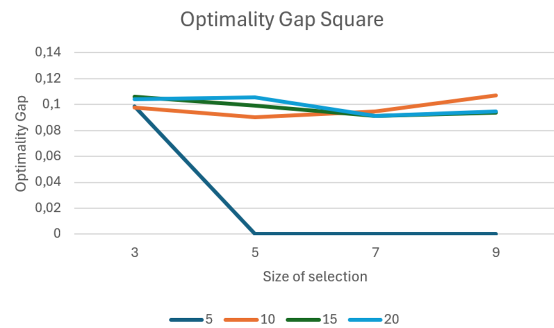



Figure 8.11: Accuracy for the Square division method

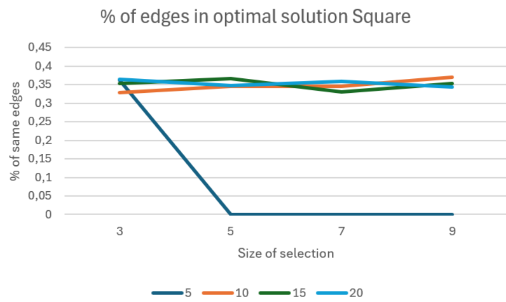


Figure 8.12: Percentage of edges present in the optimal solution

The Figure 8.13 below show the Pareto Front for the square division method. For the rest of the evaluations for this method, the number of vertical and horizontal division will be 10 (so 100 division in total) and the size of small area selection will be a square with a side of 3 small divisions length.

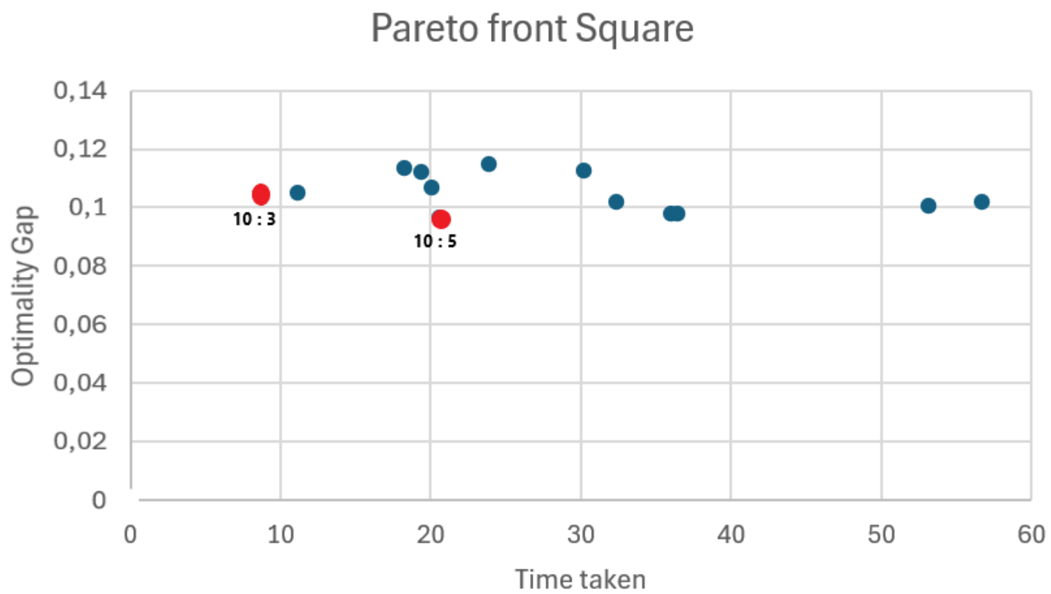


Figure 8.13: Pareto Front for the square division method

### 8.1.5 Line

As for the previous methods, here are the results for the method using the line division of the nodes. Figure 8.31 here below shows the time taken on average, Figure 8.32 shows the average difference with the optimal solution while Figure 8.16 shows the proportion of edges also found in the optimal solution.

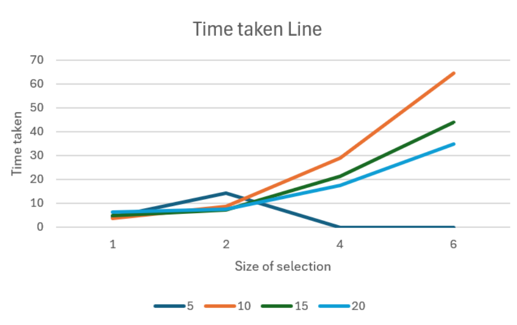



Figure 8.14: Time taken for the Line division method

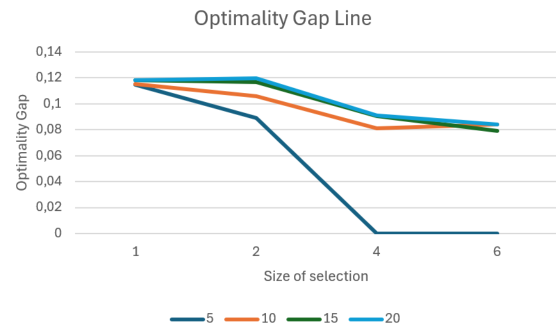



Figure 8.15: Accuracy for the Line division method

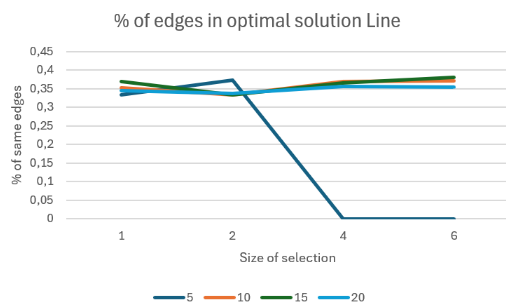



Figure 8.16: Percentage of edges present in the optimal solution

The Figure 8.17 below show the Pareto Front for the square division method. For the rest of the evaluations for this method, the number of vertical and horizontal division will be 10 (so 100 division in total) and the width of the selection will be 4 (we want to keep a sufficiently good accuracy, that's why we prefer the 10:4 solution over the 5:2).

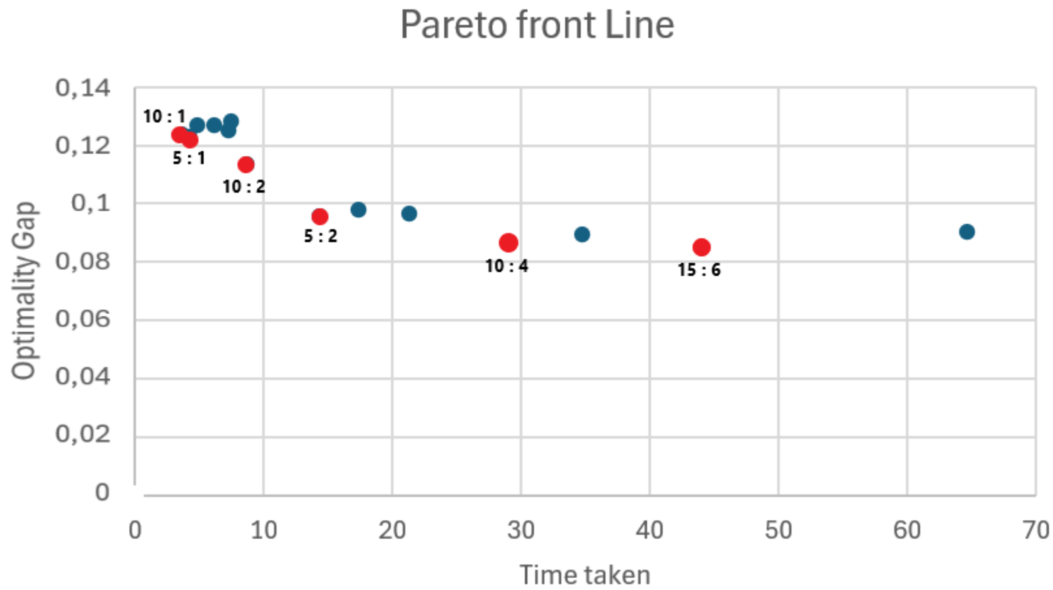



Figure 8.17: Pareto Front for the line division method

## 8.2 Variation based on TSP size

Now that the optimal parameters for each sub-TSP node selection method have been set, it is time to measure them against each other to be able to find their strengths and weaknesses.
In this section, the size of the TSP is the element on which the evaluations focus. To do this, TSP of size 100, 200, 400 and 800 were solved with the 4 sub-TSP selection methods with the parameters set in the previous section.
You can see the variation of the time taken, the accuracy as well as the proportion of edges present in the optimal solution in Figure 8.18, 8.19 and 8.20 respectively.

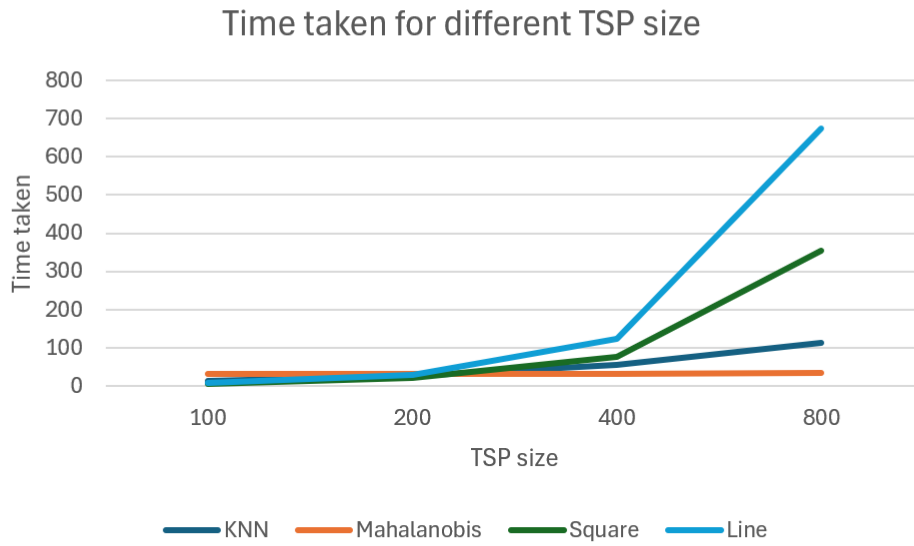



Figure 8.18: Time taken for each methods for different size of TSP

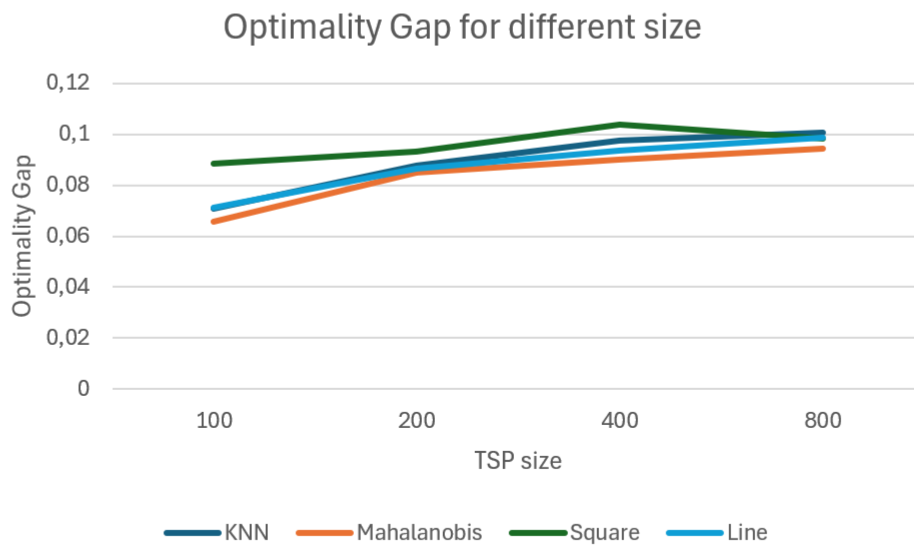



Figure 8.19: Accuracy for each methods for different size of TSP

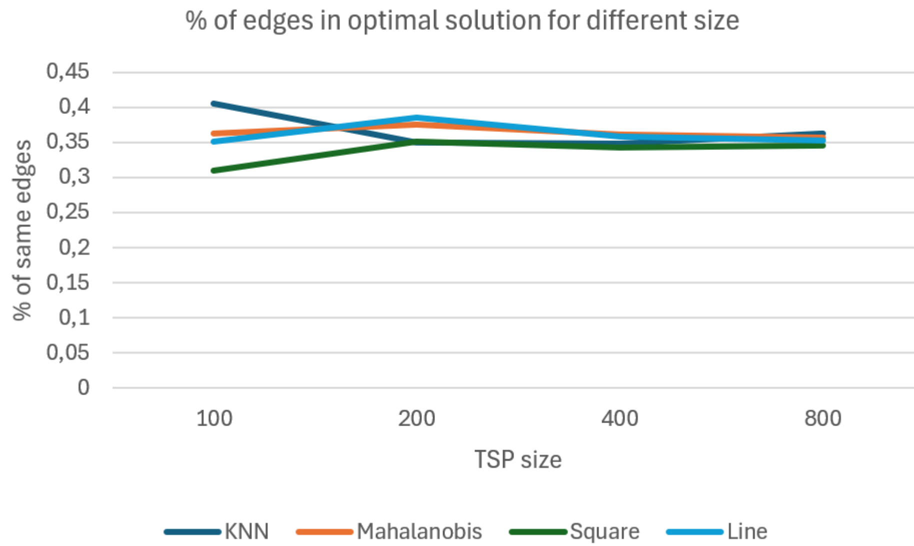



Figure 8.20: Percentage of edges present in the optimal solution for each methods for different size of TSP

## 8.3 Method used to solve sub-TSP

Until now, to solve the sub-TSPs selected with the 4 methods previously analyzed, a Simulated Annealing method had been chosen. But this is not the only existing method, so this section focuses on the results of measurements comparing the 3 different sub-TSP resolution methods for each 4 sub-TSP selection methods.
The section 8.3.1 shows the time taken for each 3 resolution method for all 4 selection methods, while section 8.3.2 show the accuracy and section 8.3.3 show the proportion of same edges.

### 8.3.1 Time of computation

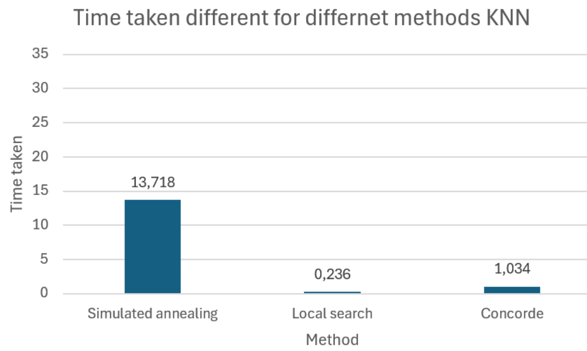



Figure 8.21: Time taken for the Line division method

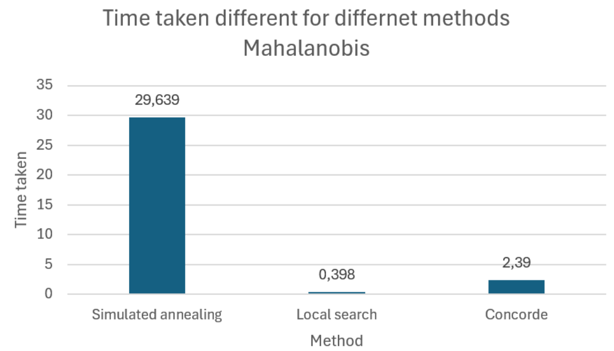



Figure 8.22: Accuracy for the Line division method

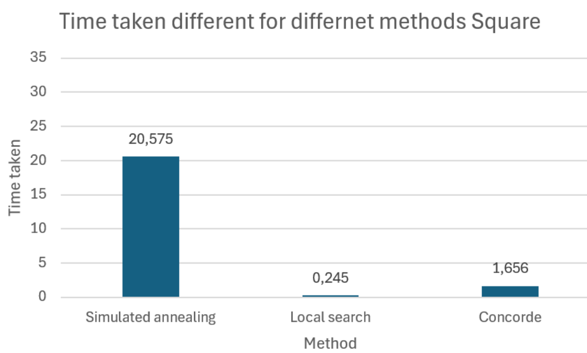



Figure 8.23: Time taken for the Line division method

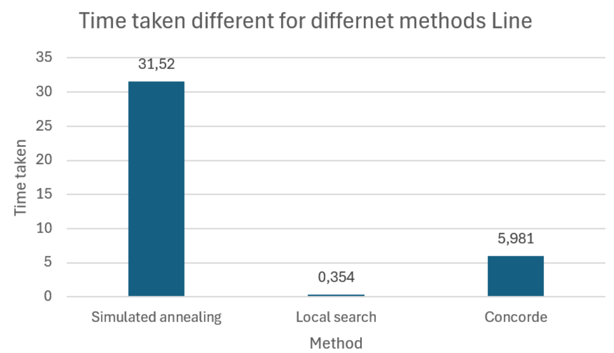



Figure 8.24: Accuracy for the Line division method

### 8.3.2 Optimality Gap

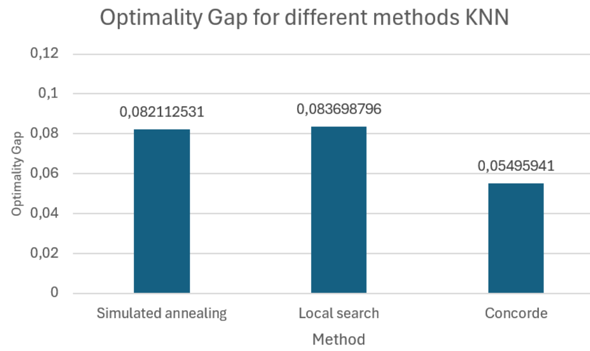



Figure 8.25: Time taken for the Line division method

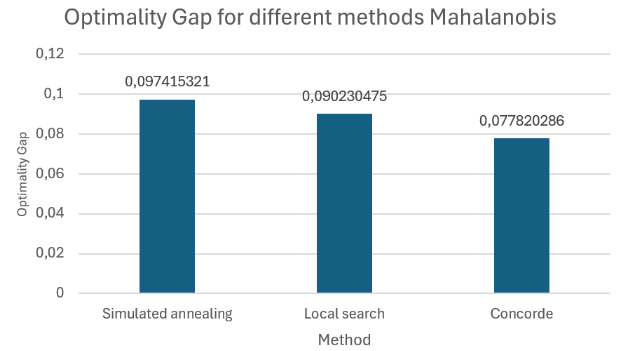



Figure 8.26: Accuracy for the Line division method

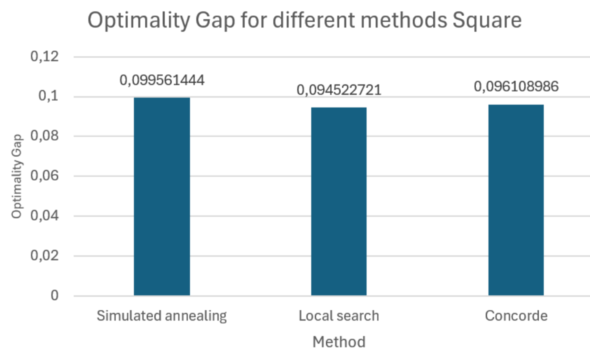



Figure 8.27: Time taken for the Line division method

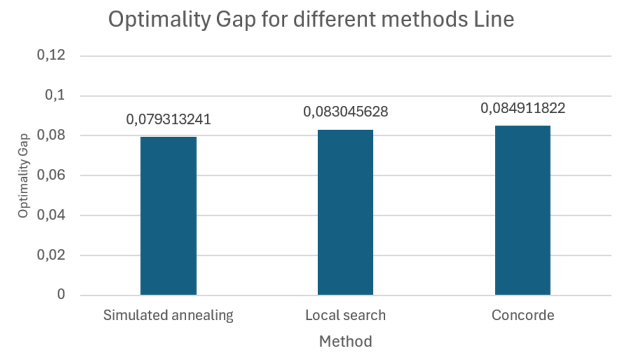



Figure 8.28: Accuracy for the Line division method

### 8.3.3 Proportion of same edges

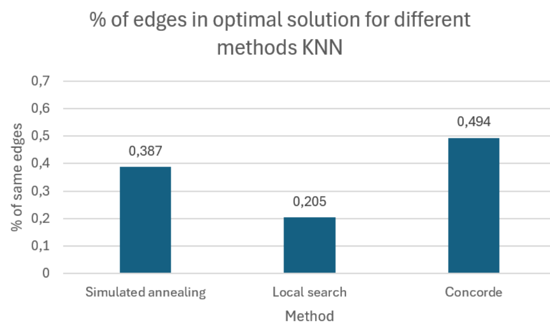



Figure 8.29: Time taken for the Line division method

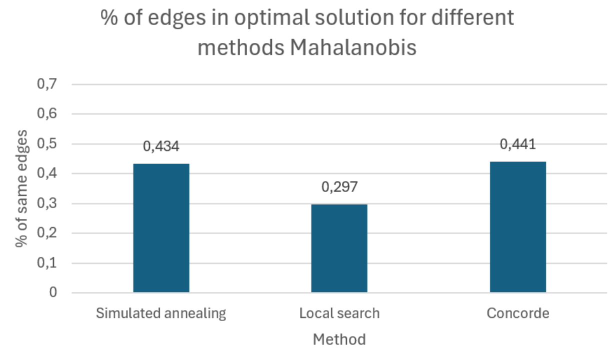



Figure 8.30: Accuracy for the Line division method

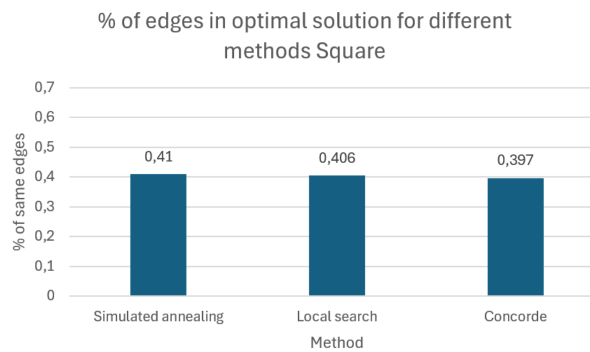



Figure 8.31: Time taken for the Line division method

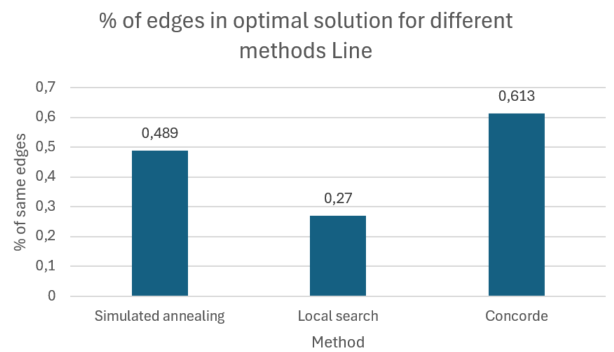



Figure 8.32: Accuracy for the Line division method

# Chapter 9

# Discussion

## 9.1 Result analysis

Now that the tests carried out have been explained and the results obtained, it is time to analyze all these results to derive the characteristics of the models.

### 9.1.1 Influence of the parameters:

The first point to analyze is the influence of parameters on the execution of the models. Each method of selecting nodes for sub-TSPs will be discussed in turn.

#### Euclidean KNN

For the Euclidean KNN, there were 2 parameters to take into account: the size of the sub-TSPs and the number of sub-TSP to resolve.
The first element that can quickly be observed is that the execution time is linked to the size of the sub-TSPs, which seems logical, because solving a TSP with 50 nodes takes more time than solving a TSP with 10 nodes. So the selected sub-TSPs are larger, the higher the execution time will be.
It is also directly linearly correlated to the number of sub-TSPs to be solved. This is very easily explained by the fact that if there are twice as many sub-TSPs to solve, twice as many operations are performed and therefore the execution time is doubled, no surprise there.
The second important point is the variation of the optimality gap as a function of the number of sub-TSPs. For this measurable, the results are less explicit than for the execution time. Indeed, for large sub-TSP sizes, increasing the number of sub-TSPs reduces the optimality gap (which means that they get closer to the optimal solution), but on the other hand the effect seems opposite. for the case with sub-TSPs with a size of 5 nodes.

Lately, the proportion of similar edges between the solution obtained and the final solution increases slightly when the number of sub-TSPs increases, but without being very significant. Furthermore, contrary to what might have been expected, the proportion of similar edges does not seem to be correlated with the optimality gap.

#### Mahalanobis KNN

For the KNN using a Mahalanobis distance, we observe that the execution time is directly correlated in a linear manner to the number of sub-TSPs, this is explained in the same way as for the Euclidean KNN. On the other hand we observe a strange event, the sub-TSPs of size 15 are faster to be resolved than those of sizes 5 and 10, which is not logical here. But this can be explained by the method used to solve the sub-TSPs which is a method coming from a solver using simulated annealing, this method takes more time for small instances of TSp, because it searches for the optimal solution while for larger instances it only opts for a local solution.
Regarding the optimality gap, it is very clear that increasing the size of the sub-TSPs or the number of sub-TSPs resolved increases the precision of the algorithm (and therefore reduces the optimality gap).
Just as for the Euclidean TSP, the proportion of similar edge between the solution obtained and the final solution only improves very little when the number of sub-TSP resolutions increases and it does not seem to be correlated with the optimality gap.

#### Square division

For the selection of sub-TSP nodes using the square selection method, the results are less telling than for that using the 2 KNNs. First, it must be clarified that for a division of space into 5*5 zones, only the selection with a size of 3*3 could be evaluated, because it is not possible to select a set of 5*5, 7*7 or 9*9 if there are only 5*5 zones existing (taking a selection of 5*5 in a zone of 5*5 amounts to resolving the entire TSP at once and therefore not apply the model, only apply ones the solver used for the small-TSP resolution).
To talk about execution time, it is better to take a smaller selection of sub-area and/or divide the space as little as possible to have more nodes in these selections. What can, however, be very clearly observed is the absence of linear correlation in the time-related results.
When we look at the optimality gap, an interesting phenomenon can be observed, a kind of central symmetry seems to appear, the results with a small selection on

a small division of space (few sub-area) have the same results as large selections on large divisions of space (more sub-area). This is explained by the fact that taking many smaller zones actually corresponds to taking a smaller number of larger zones, geometrically, it therefore amounts to the same, the sub-TSPs thus selected therefore have the same number of nodes.
But if this gives equivalent sub-TSPs, why is this symmetry not present in the graph showing the execution time? Simply because when taking a large selection on a very divided space leads to a greater number of sub-TSPs, although their size is equal to the case in which there is a smaller selection of larger area.

**Linear division**

For linear selection, we observe that the more the number of divisions increases, the less the calculation time, this is easily explained by the fact that there are fewer nodes selected on a line if it is thinner . On the other hand, increasing the width of this line also increases the calculation time, which still makes sense as well. However, it is necessary to point out that the variation in calculation time is not linear!
It is also possible to observe that the variation in the optimality gap is linked to the size of the selection (the width of the line), the wider it is, the better the solution obtained (smaller optimality gap) but the number of divisions is also important, the more divisions there are (and therefore smaller), the less good the solution will be.
This case is very interesting because it clearly shows that it is not possible to have a solution quickly while asking for the best solution, you have to choose between accuracy and computation time.

### 9.1.2 Influence of the TSP size

It is now time to analyze the impact of varying the size of the TSPs on the different algorithms created.
The first point to analyze is the impact on execution time. We immediately see that KNNs have linear calculation times depending on the size of the TSPs, which is perfectly explained by the fact that the number of selected sub-TSPs varies directly depending on the number of nodes, and that if the size of these sub-TSPs does not vary, taking times more sub-TSPs only doubles the co time. On the other hand, for the two models using a division of space (Square and Line), as said in the previous section, their calculation time was not linear, which is very visibly observed here. This case can be explained by the fact that the number of sub-TSP does not change

when the size of the original TSP change, but the size of the small TSP vary in fact, and given that the solver used to solve the sub-TSP is not linear, augmenting the size of the sub-TSP causes the calculation time to vary non-linearly.
As for the optimality gap, the larger the TSP, the more difficult it is to opt for a small one, this is explained by the fact that the problem is more complex and that there are therefore more solutions possible locality in which the algorithms can fall. On the other hand, the proportion of edges present both in the optimal solution and in the final solution of the algorithms tends towards 35

### 9.1.3 Influence of the method used to solve the sub-TSP

The influence of sub-TSP resolution methods is really relevant for execution speed. Indeed, for the 4 algorithms, 2-opt is the fastest, followed by Concorde, followed by the TSP-solver using simulated annealing.
For the optimality gap, resolution via Concorde is preferable for the 2 KNNs, but not for the 2 solutions using the division of space into equal areas, for which there does not seem to be a preference for the resolution method. sub-TSPs
A possible way to explain this phenomenon is the fact that KNNs have many more sub-TSPs of smaller sizes and that a slight variation in their resolutions can lead to a greater scratch, it is therefore preferable to have the optimal solution for each sub-TSP. Where the 2 division methods have more large sub-TSPs fewer in number, and therefore the impact of the TSP resolution method matters much less than the choice of parameters.
The proportion of edges present in the 2 solutions is very influenced by the sub-TSP resolution method. The best method is Concorde, which actually makes a lot of sense given that if the sub-TSP solution is optimal, it has a much greater chance that its edges belong to the final solution than if the sub-TSP solution is not its optimal solution.

## 9.2 Comparison against the deep learning approach

Now that the algorithms have been analyzed, it is time to compare them with the deep reinforcement learning model provided by Chaitanya K. Joshi, Quentin Cappart, Louis- Martin Rousseau and Thomas Laurent in the "Learning the Travelling Salesperson Problem Requires Rethinking Generalization" paper [4] which was detailed in Chapter 3.
To evaluate the deep learning model, the same evaluation method as for the previous alogirithms was used, that is to say, measuring the average optimality gap for 100 TSPs of 200 different nodes (the 100 TSPs used were the same than those used to

measure the algorithms in the previous sections).
The average optimality gap obtained for the deep learning algorithm is 0.213, which is much higher than all algorithms regardless of the parameters used! The other major advantage of the algorithmic solution is its ability to be applicable to larger TSPs, which is not the case for the deep learning approach (in fact, the models are no longer consistent if the TSP is too large since training a deep reinforcement learning model on large TSPs takes a huge amount of time).
It is also important to note that the analysis of the time taken to obtain a response was not measured given that the objective of this Master thesis was to opt for a method to achieve similar results without a long phase training, which makes the execution time of the deep learning model irrelevant.

## 9.3 Other advantages of the solution

Other significant advantages of the algorithmic solutions proposed in this Master thesis can also be highlighted.
The first advantage concerns only the two methods of selecting sub-TSP nodes using a KNN. By using a resolution based on one of these 2 methods, the algorithm maintains a linear execution time depending on the size of the TSP, which can be very practical for a TSP having a very large size (10,000 nodes and more) because this keeps the calculation time very short compared to other solutions that are not linear (2-opt, simulated annealing or the Concorde solver)!
The second major advantage also concerns the execution time but this time for the 4 algorithms. Since the resolutions of all subTSPs are all independent of each other, it is possible to distribute the computation across multiple threads. All the results obtained in this master thesis were carried out with a single thread, but if the algorithm runs on a 20-core machine, it could be almost 20 times faster. Once again, this advantage is very interesting for very large TSPs.

# Chapter 10

# Conclusion

It is now time to conclude this Master's thesis by reviewing certain key points from it. First of all, it should be remembered that the traveling salesman problem is a very well-known subject and on which a lot of research has been and will continue to be done, this thesis is therefore a continuation of this research by exploring a new idea which seeks to reproduce the results obtained by Chaitanya K. Joshi, Quentin Cappart, Louis- Martin Rousseau and Thomas Laurent [4] thanks to an algorithmic approach.
The main objectives set in Chapter 2 (no long learning phase, solution close to optimal, acceptable resolution time and possibility of being generalized to larger TSPs without causing accuracy issue) were achieved as discussed in the Chapter 9 thanks to an innovative approach using 4 different algorithms based on the same assumption that can be summarize by: "If we solve multiple sub-TSPs, the edges taken in these sub-solutions are more likely to be in the optimal solution of the original TSP than the other edges." This assumption implicitly suggests that we bias the final result by greatly favoring the edges connecting close nodes together, but as seen in Chapter 6, it is indeed relevant in the context of this Master's thesis. Although the method used is neither better nor faster than the best solution existing to date (the Concorde solver [1]), it explores a new way of doing things and presents certain advantages 9.3 and it has been interesting for me to work on it throughout this year.

# Chapter 11

# GitHub

You can find the all code repository on Github with this link : `https://github.com/ahennecart/TSP-Solver`

# Bibliography

[1] Robert E. Bixby Vasek Chvatal William J. Cook Applegate, David L. Concorde tsp solver, 2003. https://www.math.uwaterloo.ca/tsp/concorde.html [Accessed on 14/01/2024].

[2] Georges A Croes. A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812, 1958.

[3] dmishin. Python-tsp : Suboptimal travelling salesman problem (tsp) solver, 2020. https://github.com/dmishin/tsp-solver [Accessed on 14/01/2024].

[4] Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, 27(1-2):70–98, 2022.

[5] jvkersch. Pyconcorde : a python wrapper around the concorde tsp solver, 2023. https://github.com/jvkersch/pyconcorde [Accessed on 14/01/2024].

[6] Alexander V Lotov and Kaisa Miettinen. Visualizing the pareto frontier. pages 213–243, 2008.

[7] Prasanta Chandra Mahalanobis. On the generalized distance in statistics. *Sankhyā: The Indian Journal of Statistics, Series A (2008-)*, 80:S1–S7, 2018.

[8] Cardiff University. Farthest insertion, 2024. \https://users.cs.cf.ac.uk/C.L.Mumford/howard/FarthestInsertion.html [Accessed on 12/05/2024].

[9] Wikipedia. Pareto front, 2024. https://en.wikipedia.org/wiki/Pareto_front [Accessed on 11/05/2024].

[10] Wikipedia. Simulated annealing, 2024. \https://en.wikipedia.org/wiki/Simulated_annealing [Accessed on 09/01/2024].

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**