University of Padua

Master Degree in Computer Engineering

A.Y. 2019/2020

Operations Research Course Report

Travelling Salesman Problem

Barison Marco

Moratello Matteo

Obetti Christian

Contents

Chapter 1

Introduction

1.1 Computational complexity theory

First of all we want to start this report with a brief introduction to the computational complexity theory and in particular to the P and NP classes. The computational complexity theory aim to establish how difficult is a certain problem. In order to do this, it is required a system to describe the complexity of the problem that has general validity, so that is independent from the computer that we are using. For this reason an algorithm is studied with respect to the number of elementary operations that are required to complete it if it's executed by a Turing machine. So for example if we say that an algorithm is $O(n)$ where n is the size of the instance, it means that it requires in the worst case a number of operation to finish that it's linear with respect to the problem size itself.
We need to spend a few words for the Turing machine. For simplicity we can consider a deterministic Turing machine (DTM) the equivalent of a deterministic computer and a nondeterministic Turing machine (NTM) the equivalent of an ideal nondeterministic computer that is able to exploit multiple action at the same time. Please notice that this is a great simplification, an in-depth discussion about Turing machine and the computational complexity theory exceeds the purpose of this introduction, that is only to give a general idea. Now, in an informal way, we can say that a problem belong the the P class if exist an algorithm that resolve it in a polynomial time on a deterministic Turing machine. Moreover we say that a problem belong to the NP class (nondeterministic polynomial) if exist an algorithm able to resolve the problem in polynomial time on a nondeterministic Turing machine.
A DTM can simulate a NTM, but in the worst case this require an exponential number of operation (again there is a theorem that guarantee this, omitted on this report). This means that a problem that belong to the NP class, can require an exponential number of operation to be resolved on a deterministic computer, making it infeasible if the instance is not small.
There are no theoretical proofs that $P \neq NP$ (This is one of the *Millennium Prize Problems*) so this mean that NP problems may be exponential only because we don't know an efficient algorithm to resolve them, however nowadays this is considered

unlikely and it's widely believed that some problems are intrinsically exponential to resolve.
In conclusion there are some problem that are considered even more difficult than the ones that belong to the *NP* class. These are called *NP-hard* and they are of particular interest because lots of problem that have practical application in reality belong to this typology [1].

1.2 Traveling salesman problem

The traveling salesman problem (TSP) consist in finding the Hamiltonian path [2] of lowest cost given an oriented graph $G = (V, A)$. The problem can be also defined in an analogous way on a non oriented graph if the cost of an edge doesn't depend on the direction of the edge itself. This problem is very common in several real life situations, for example a courier that has to deliver parcels on different locations. However TSP is NP-hard and this is the reason why we can obtain exact solutions only for small instances (up to 600-800 nodes with the most advanced algorithms) while we have to look for different heuristic strategies if the instances are large. A possible integer linear programming model can be the following:

$$\min \underbrace{\sum_{(i,j)\in A} c_{ij}x_{ij}}_{\text{circuit cost}} \tag{1.1}$$

$$\underbrace{\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1}_{\text{one edge incoming in j}}, \quad j \in V \tag{1.2}$$

$$\underbrace{\sum_{(i,j)\in\delta^+(j)} x_{ij} = 1}_{\text{one edge outgoing in i}}, \quad i \in V \tag{1.3}$$

$$\underbrace{\sum_{(i,j)\in\delta^+(S)} x_{ij} \geq 1}_{\text{subtour eliminator}}, \quad S \subset V \ : \ 1 \in S \tag{1.4}$$

$$x_{ij} \geq 0 \text{ integer}, (i,j) \in A \tag{1.5}$$

where x_{ij} are the decision variables defined in this way:

$$x_{ij} = \begin{cases} 1 & \text{if the edge } (i,j) \in A \text{ is selected in the optimal circuit} \\ 0 & \text{otherwise} \end{cases}$$

[1] For more detailed information consult ???

[2] In the mathematical field of graph theory, a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once.

The 1.2 means that the sum of all the outgoing edges in each vertex must be equals to 1, similar the 1.3 says that the sum of all the edges entering a vertex must be equals to one. So if we consider both 1.2 and 1.3 we have that each vertex must have only one incoming and one outgoing edge.
The 1.4 says that each vertex v must be reachable from the vertex 1, so that the solution must be connected and must visit all the vertices (subtour are not allowed). These constraints are $O(2^n)$, on the following chapter we will show some possible solution to handle them.

In the next chapters we are going to present all our algorithms together with their implementations and tests. In particular, this report is structured as follows:

- in Chapter 2 we will present some Compact models that we have studied and implemented, showing for each the mathematical formulation, a little description and a performance comparison between them;
- in Chapter 3 we will present Exact algorithms which can solve the TSP problem to optimum using the DJF formulation, handling the $O(2^n)$ SECs constraints with different approaches;
- in Chapter 4 we will present two heuristic algorithms based on CPLEX that can handle problems with thousands of nodes;
- in Chapter 5 we will present some TSP solution builders and a variety of metaheuristic algorithms;
- in Chapter 6 we will present the conclusions of our work.

All the source code we developed is available at:

https://github.com/Morat96/RO2-homework [3].

[3]For more informations on how compile and run the code, see the README.

Chapter 2

Compact Models

Here we introduce different formulations for the traveling salesman problem that we implemented. For all the following model we will use the decision variable x_{ij} already defined in 1.2.

2.1 Sequential formulation

For this version, formulated by Miller, Tucker and Zemlin in 1960, we introduce the continuous variable

$$u_i = \text{sequence in which point i is visited } (i \neq 1)$$

and the constraint 2.4. In total this formulation has $n^2 - n + 2$ constraints, $n(n-1)$ 0-1 variables and $(n-1)$ continuous variables.

$$\min \underbrace{\sum_{(i,j)\in A} c_{ij} x_{ij}}_{\text{circuit cost}} \tag{2.1a}$$

$$\underbrace{\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1\,, \quad j \in V}_{\text{one edge incoming in j}} \tag{2.1b}$$

$$\underbrace{\sum_{(i,j)\in\delta^+(j)} x_{ij} = 1\,, \quad i \in V}_{\text{one edge outgoing in i}} \tag{2.1c}$$

$$u_i - u_j + n x_{ij} \leq n - 1 \quad \forall i, j \in N - \{1\},\ i \neq j \tag{2.1d}$$

2.2 Flow-based formulations

2.2.1 Single commodity flow

Provided by Gavish and Graves in 1978, it adds continuous variables:

$$y_{ij} = \text{flow in arch } (i,j)\ i \neq j$$

In total it has $n(n+2)$ constraints, $n(n-1)$ 0-1 variables and $n(n-1)$ continuous variables. Note that the 2.2d can be improved substituting the second term with $(n-1)x_{ij}$

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.2a}$$

$$\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1, \quad j \in V \tag{2.2b}$$

$$\sum_{(i,j)\in\delta^+(j)} x_{ij} = 1, \quad i \in V \tag{2.2c}$$

$$y_{ij} \leq (n-1)x_{ij} \quad \forall i,j \in N,\ i \neq j \tag{2.2d}$$

$$\sum_{j,\ j\neq 1} y_{1j} = n-1 \tag{2.2e}$$

$$\sum_{i,\ i\neq j} y_{ij} - \sum_{k,\ i\neq k} y_{jk} = 1 \quad \forall j \in N - \{1\} \tag{2.2f}$$

2.2.2 Two commodity flow

Created by Finke, Claus and Gunn in 1983. It maintains constraint 2.1b and 2.1c (from this point they and the objective function will not be shown in the formulation) and introduce the continuous variables:

$$y_{ij} = \text{flow of commodity 1 in arc } (i,j)\ i \neq j$$

$$z_{ij} = \text{flow of commodity 2 in arc } (i,j)\ i \neq j$$

and add the constraints:

$$\sum_{j,\ j\neq 1} (y_{1j} - y_{j1}) = n-1 \tag{2.3a}$$

$$\sum_{j} (y_{ij} - y_{ji}) = 1 \quad \forall i \in N - \{1\},\ i \neq j \tag{2.3b}$$

$$\sum_{j,\ j\neq 1} (z_{1j} - z_{j1}) = -(n-1) \tag{2.3c}$$

$$\sum_j (z_{ij} - z_{ji}) = -1 \quad \forall i \in N - \{1\},\ i \neq j \tag{2.3d}$$

$$\sum_j (y_{ij} + z_{ij}) = n - 1 \quad \forall i \in N \tag{2.3e}$$

$$y_{ij} + z_{ij} = (n-1)x_{ij} \quad \forall i, j \in N \tag{2.3f}$$

In total it has $n(n+4)$ constraints, $n(n-1)$ 0-1 variables and $2n(n-1)$.

2.2.3 Multi-commodity flow

Proposed by Wong and Claus in 1984. Again constraint 2.1b and 2.1c are maintained. It introduce the continuous variable:

$$y_{ij}^k = \text{flow of commodity k in arc } (i,j) \in N - \{1\}$$

and the following constraints:

$$y_{ij} \leq x_{ij} \quad \forall i, j, k \in N,\ k \neq 1 \tag{2.4a}$$

$$\sum_i y_{1i}^k = 1 \quad \forall k \in N - \{1\} \tag{2.4b}$$

$$\sum_i y_{i1}^k = 0 \quad \forall k \in N - \{1\} \tag{2.4c}$$

$$\sum_i y_{ik}^k = 1 \quad \forall k \in N - \{1\} \tag{2.4d}$$

$$\sum_j y_{kj}^k = 0 \quad \forall k \in N - \{1\} \tag{2.4e}$$

$$\sum_i y_{ij}^k - \sum_i y_{ji}^k = 0 \quad \forall j, k \in N - \{1\},\ j \neq k \tag{2.4f}$$

In total there are $n^3 + n^2 + 6n - 3$ constraints, $n(n-1)$ 0-1 variables and $n(n-1)^2$

2.3 Time staged formulations

2.3.1 First stage dependent

Introduced by Fox, Gavish and Graves in 1980. Constraint 2.1b and 2.1c are maintained, in addition we have 0-1 integer variables:

$$y_{ij}^t = \begin{cases} 1 & \text{if the edge } (i,j) \text{ is traversed at stage t} \\ 0 & \text{otherwise} \end{cases}$$

and constraints below, for a total of $n(n+2)$ constraints and $n(n-1)(n+1)$ 0-1 variables.

$$\sum_{i,j,t} y_{ij}^t = n \tag{2.5a}$$

$$\sum_{j,t\ t\geq 2} t y_{ij}^t - \sum_{k,t} t y_{ki}^t = 1 \quad \forall i \in N - \{1\} \tag{2.5b}$$

$$x_{ij} - \sum_t x_{ij}^t = 0 \quad \forall i,j \in N,\ i \neq j \tag{2.5c}$$

with the condition

$$y_{il}^t = 0\ \forall t \neq n, \quad y_{ij}^t = 0\ \forall t \neq 1, \quad y_{ij}^l = 0\ \forall i \neq 1, \quad i \neq j \tag{2.5d}$$

2.3.2 Second stage dependent

Provided by Fox, Gavish and Graves in 1980. It uses the same variables of first stage and constraints 2.1b and 2.1c and it adds:

$$\sum_{i,t\ i\neq j} y_{ij}^t = 1 \quad \forall j \in N \tag{2.6a}$$

$$\sum_{j,t\ j\neq i} y_{ij}^t = 1 \quad \forall i \in N \tag{2.6b}$$

$$\sum_{i\ j\neq i} y_{ij}^t = 1 \quad \forall t \in N \tag{2.6c}$$

$$\sum_{j,t\ t\geq 2} t y_{ij}^t - \sum_{k,t} t y_{ki}^t = 1 \quad \forall i \in N - \{1\} \tag{2.6d}$$

In totals there are $4n-1$ constraints and $n(n-1)(n+1)$ 0-1 variables.

2.3.3 Third stage dependent

Again we have the same variables of first stage and constraints 2.1b and 2.1c. In addition, for a total of $2n^2 - n + 3$ constraints and $n(n-1)(n+1)$ 0-1 variables we have:

$$\sum_j y_{1j}^1 = 1 \tag{2.7a}$$

$$\sum_i y_{i1}^n = 1 \tag{2.7b}$$

$$\sum_j y_{ij}^t - \sum_k y_{ki}^{t-1} = 0 \quad \forall i,t \in N - \{1\} \tag{2.7c}$$

Chapter 3

Exact Algorithms

On this chapter we will describe exact algorithms based on the Dantzig–Fulkerson–Johnson formulation, described in chapter 1. The main problem of this model is that the formulation has $O(2^n)$ SECs, so it is impossibile to implement all the constraints in the model since it would result in a very high execution time and memory usage even for small graphs.
For this reason, SECs constraints are added only when necessary in the model, in order to discard solutions with cycles. Using this technique, hopefully, only a small part of $O(2^n)$ SECs are added to the model.
The main algorithms that make use of this technique are Loop methods and CPLEX callbacks.

3.1 Loop Methods

On this section we will describe two iterative approaches in order to solve the DFJ model without adding an exponential number of Cycle constraints.

3.1.1 Simple Loop

The first approach, that we called Simple Loop, is presented in Algorithm 1.
In line 2, we initialize the model with the variables, the degree constraints and the objective function. The algorithm then proceeds solving the problem within a while loop (lines 3-4), and in each iteration we check if the the optimal solution has more than one component (line 5). If so, we add, for each component, the corresponding subtour elimination. Instead, If the optimal solution has one component (lines 7-8), i.e. a Hamiltonian cycle, then we return the optimal solution found (line 9).
This algorithm turns out very efficient, especially with recent solvers. Its efficiency is due to the fact that the solution is solved very fast, especially in first iterations, and only small number of SECs are added (line 6) at each iteration. These constraints permits to prevent a lot solutions with subtours to the original problem.
At worst, since the problem is NP-hard, all constraints will be added to the model, requiring exponential computational time, but in most cases this is not the case.

It may seem that this method is not very efficient, but with recent, increasingly sophisticated solvers, this method can lead to the optimal solution, even with many nodes, in a few seconds.

Algorithm 1 Simple Loop

Input: $G = (V, E), c : E \to \mathbb{R}^+$
Output: z^* optimal solution to TSP on the input graph G

```
1: done ← false
2: model ← ∗ Initialize variables and objective function ∗
3: while !done do
4:     z* ← optimal_solution(model)
5:     if components(z*) > 1 then
6:         ∗ Add SECs for each connected component to the model ∗
7:     else
8:         done ← true
9: return z*
```

3.1.2 Heuristic Loop

The Algorithm 1 has a drawback, since at each iteration we solve to the optimum the TSP problem, even if the constraints collected until that point, are not sufficient to achieve the true optimum. In other words, we must wait that the solver finds out the optimal solution, even if the constraints avaiable may allow solutions with subtours.
For this reason, we implemented an "heuristic" version of the method seen before. This method is composed of two phases. The first phase is used to collecting some SEC constraints, relaxing the optimality concept, setting for example an higher value for the solution GAP or limiting the solution in the root node of the branching tree. This phase permits add useful constraints to the model, without computing the optimal solution in each iteration. In fact, with this phase we can obtain a lot of useful SECs in a short time, especially in the first iterations. This phase is active until the solution found has more than one component, from then on, will be activated the second phase. In the second phase, the "relaxation" of the solver is removed, solving essentially the problem with the Simple Loop method and this ensure that the solution computed is the true optimal solution.
The Heuristic Loop, is presented in Algorithm 2. In line 4, as with the first method, we initialize the model with the variables, the degree constraints and the objective function. At beginning the first phase is active (line 2), which means that some CPLEX parameters have to be setted (line 5) in order to obtain a non-optimal solution, speeding up the computational time. In our case, we decided to set two CLEX parameters, that is, the solution's GAP and the NODE lim. The first parameter force CPLEX to stop, when the solution found until that point has a GAP of a certain value (for example 5%). The second parameter specifies the node of the

branching tree until which CPLEX can compute the solution, for example, if the value of the parameter is setted to 0, CPLEX can compute the solution only in the root node. The algorithm then proceeds solving the problem within a while loop (line 6). As in the Simple Loop, at each iteration we check if the solution has more than one component, and if so, we add, for each component, the corresponding sub-tour elimination. Instead, If the optimal solution has one component, we set the flag *second_phase* to true (line 15) and reset the solver settings, thus ensuring that the solution found from now on it will be an optimal solution. During the first phase, we have collected, hopefully, a number of useful SEC constraints in a small amount of time, with respect to found the optimal solution at each iteration. For the second phase, please refer to the Simple Loop (Algorithm 1).

Algorithm 2 Heuristic Loop

Input: $G = (V, E), c : E \to \mathbb{R}^+$
Output: z^* optimal solution to TSP on the input graph G

1: *done* ← *false*
2: *first_phase* ← *true*
3: *second_phase* ← *false*
4: model ← ∗ **Initialize variables and objective function** ∗
5: CPXsetintparam ← ∗ **Set MIP Gap and/or Node lim in order to obtain a non-optimal solution** ∗
6: **while** !done **do**
7: **if** *second_phase* **then**
8: CPXsetintparam ← ∗ **Return to original solver setting in order to obtain an optimal solution** ∗
9: *second_phase* ← *false*
10: z^* ← optimal_solution(model)
11: **if** $components(z^*) > 1$ **then**
12: ∗ **Add SECs for each connected component to the model** ∗
13: **if** $components(z^*) == 1$ & $first\_phase$ **then**
14: *first_phase* ← *false*
15: *second_phase* ← *true*
16: **if** $components(z^*) == 1$ & $!first\_phase$ **then**
17: *done* ← *true*
18: **return** z^*

3.1.3 Final comparison between Loop Methods

3.2 Callbacks

On this section we will describe how to solve the DFJ model using CPLEX callbacks. Until now, the algorithms we described resolve the TSP problem with an iterative approach, that is, solve multiple times the problem of a relaxed model, adding the SECs constraints when the solution found has multiple tours. Since CPLEX uses the branch-and-cut technique to solve exactly the model, with this approach multiple decision trees will be created from scratch, losing much of the information from previous trees. Another way to handle the SECs constraints is to exploit the branch-and-cut technique. In particular, when an integer solution is found by CPLEX during the decision tree, we can reject it if contains multiple tours, adding for each of them a SEC constraint. With this method we develop only a single decision tree, and this results, hopefully, in a faster computation. To this end, CPLEX provides a series of callbacks that can be used for adding constraints and solutions on the fly.

3.2.1 Lazy Callback

The lazy callback permits to resolve the TSP problem in a very fast and natural way. As described previously, in order to solve the problem we must reject solutions that have multiple tours, adding SECs constraints on the fly. This leads to Algorithm 3 in which we instantiate the model and the objective function as usual (line 2). Then we instantiate a procedure called LazyCallback (line 3). A LazyCallback is a function called by the optimizer when an integer solution is found during the branch-and-cut algorithm. The task of the callback is to check if solution passed has more than one connected components (line 8) and if it is the case, to reject the solution adding the SECs constraints to the model. Thereafter, the resolution resumes (lines 9 - 11). The routine continues by optimally solving the model and returning the solution found (lines 4 - 5).

Algorithm 3 Lazy Callback

Output: $\boldsymbol{z}$***** optimal solution to TSP on the input graph G

1: **function** TSPOPT($G = (V, E), c : E \to \mathbb{R}^+$)
2: model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
3: optimizer $\leftarrow$ $*$ **Initialize the LazyCallback function within the optimizer** $*$
4: $\boldsymbol{z}$***** $\leftarrow$ optimizer(model)
5: **return** $\boldsymbol{z}$*****
6:
7: **function** LAZYCALLBACK($\boldsymbol{x}$, model)
8: comp $\leftarrow$ $*$ **number of components in the integer solution** $\boldsymbol{x}$ $*$
9: **if** ($comp > 1$) **then**
10: $*$ **Add SECs for each connected component to the model** $*$
11: **return**

3.2.2 UserCut Callback

Another interesting callback is the UserCut which is used to generate SECs constraints on fractional solutions. The idea is pretty the same as the Lazy callback; when a fractional solution is found during the exploration of the decision tree by the solver, the UserCutCallback function checks the solution and adds constraints to the model on the fly. The procedure is described in algorithm 4. In this case both LazyCallback and UserCutCallback must be defined (line 3). In particular, the UserCutCallback function checks how many components there are in the solution (line 14), if there are many connected components then it adds the SECs for each connected component (lines 15 - 16). Instead, when the graph is connected, we apply to the model violated cuts, looking for sections with a capacity less than a certain threshold. The cutoff value is set to 2.0 - ϵ, with $\epsilon = 0.1$ in our implementation (lines 17 - 18). We used the *Concorde*[1] algorithms in order to fast computing the connected components and violated cuts in fractional solutions. Due to the fact that fractional solutions are more than integer solutions and that the UserCutCallback function employs several time consuming algorithms, the callback is called only when the depth of the decision tree is less than or equal to 10, which is a reasonable tradeoff between number of constraints applied to the model and time spent in executing flow algorithms.

Algorithm 4 UserCut Callback

Output: $\boldsymbol{z}^*$ optimal solution to TSP on the input graph G

1: **function** TSPOPT($G = (V, E), c : E \to \mathbb{R}^+$)
2: model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
3: optimizer $\leftarrow$ $*$ **Initialize the callback functions within the solver** $*$
4: $\boldsymbol{z}^*$ $\leftarrow$ optimizer(model)
5: **return** $\boldsymbol{z}^*$
6:
7: **function** LAZYCALLBACK($\boldsymbol{x}$, model)
8: comp $\leftarrow$ $*$ **number of components in the integer solution** $\boldsymbol{x}$ $*$
9: **if** ($comp > 1$) **then**
10: $*$ **Add SECs for each connected component to the model** $*$
11: **return**
12:
13: **function** USERCUTCALLBACK($\boldsymbol{x}$, model)
14: comp $\leftarrow$ $*$ **number of components in the fractional solution** $\boldsymbol{x}$ $*$
15: **if** ($comp > 1$) **then**
16: $*$ **Add SECs for each connected component to the model** $*$
17: **if** ($comp = 1$) **then**
18: $*$ **Add SECs on the separated fractionary solution** $*$
19: **return**

[1] http://www.math.uwaterloo.ca/tsp/concorde.html

3.2.3 Heuristic Callback

The last callback that we present is the Heuristic which is used to provide to the solver heuristic feasible solutions from fractional or infeasible solutions. In our case, since we are trying to solve the TSP problem, we generate first a tour from integer solutions with multiple connected components and then we add them to the solver using the CPLEX heuristic callback, which permits to add solutions and to automatically check their feasibility. This technique is very useful in the early stages of the branch-and-cut algorithm because, by providing a TSP solution to the solver, we can provide an upper-bound to the optimal solution, thus cutting many branches of the decision tree. This led us to define the Algorithm 5. The main differences with respect to Algorithm 4 are the addition of a new function called HeuristicCallback and a new algorithm to compute a tour starting from an integer solution with multiple components provided in the LazyCallback function. Since we want the algorithms to be thread safe, we must keep track of the index of each thread that produce a specific solution. To to this, for every integer solution with multiple connected components provided by the Lazy callback, we compute a new solution composed by a single tour (i.e. a TSP solution), saving in addition the index of the thread that has computed that solution (lines 11 - 12). The algorithm to compute a tour starting from an infeasible solution is presented in Algorithm 6. The idea is very simple, melt connected components in a single tour optimizing the total length by merging the most nearest components. In particular, the algorithm starts by computing the number of components of the solution (line 2). This value correspond to the iterations that the algorithm must perform in order to merge all the components of the solution (line 3). Inside the while loop, the algorithm compute the minimum delta for each pair of nodes of the graph that are in different components. The delta correpond to the incremental cost of the objective function due to a specific swap of edges. Two cases are possible (lines 6 - 13); in the first case, the edge composed of $(a, a' = succ(a))$ is replaced by (a, b) and the edge (b, b') is replaced by (b', a'), increasing the objective function of $\Delta(a, b)$. In this case, the order of the edges in the second component must be reversed to preserve the correct direction of the graph (Figure 3.1 a - b). The second case is simpler and is obtained by crossing the edges as in Figure 3.1 c and d. Once the minimum delta is found, the correct swap is applied to the graph (lines 14 - 20) and the number of components is reduced by 1 (line 21 - 22). Concluding the description of algorithm 5, when the HeuristicCallback function is called by the solver, it checks whether a solution has been saved for the current thread (line 24) and, if a solution exists, it adds it to the solver, leaving the latter to verify its feasibility (line 25).

Algorithm 5 Heuristic Callback

Output: z^* optimal solution to TSP on the input graph G

1: **function** TSPOPT($G = (V, E), c : E \to \mathbb{R}^+$)
2: model $\leftarrow *$ **Initialize variables and objective function** $*$
3: optimizer $\leftarrow *$ **Initialize the callback functions within the solver** $*$

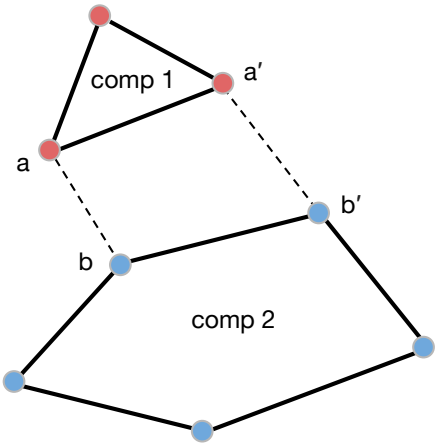
```
    z* ← optimizer(model)
    return z*

function LazyCallback(x, model)
    comp ← * number of components in the integer solution x *
    if (comp > 1) then
        * Add SECs for each connected component to the model *
        x' ← complete_tour(x)
        * Save x' and the thread index *
    return

function UserCutCallback(x, model)
    comp ← * number of components in the fractional solution x *
    if (comp > 1) then
        * Add SECs for each connected component to the model *
    if (comp = 1) then
        * Add SECs on the separated fractionary solution *
    return

function HeuristicCallback(x, model)
    if * there is a solution available in the current thread * then
    * Add the TSP heuristic solution x' to the solver *
    return
```

Algorithm 6 complete tour

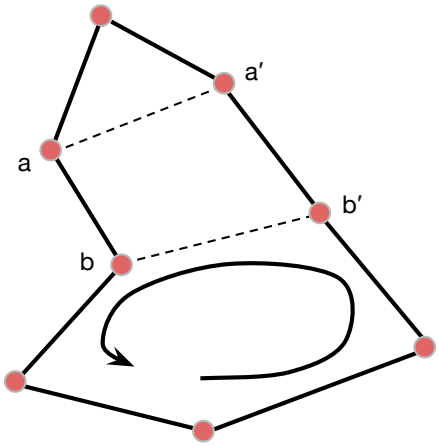
Input: $\boldsymbol{x}$ solution with multiple connected components
Output: $\boldsymbol{x}'$ heuristic TSP solution

```
min ← ∞
ncomp ← components(x)
while (ncomp != 1) do
    for each a, b ∈ V do
        if (comp(a) != comp(b)) then
            Δ(a, b) = dist(a, b') + dist(b, a') − dist(a, a') − dist(b, b')
            if (Δ(a, b) < min) then
                min = Δ(a, b)
                flag ← 0
            Δ(a, b)' = dist(a, b) + dist(b', a') − dist(a, a') − dist(b, b')
            if (Δ(a, b)' < min) then
                min = Δ(a, b)'
                flag ← 1
```

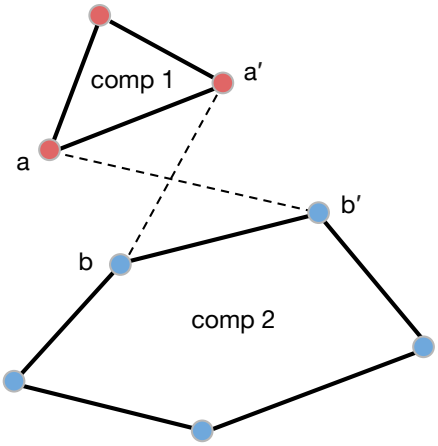
14: **if** $(flag = 1)$ **then**
15: $*$ **reverse the order of edges of the second component** $*$
16: $(a, b) \leftarrow (a, a')$
17: $(b', a') \leftarrow (b, b')$
18: **if** $(flag = 0)$ **then**
19: $(a, b') \leftarrow (a, a')$
20: $(b, a') \leftarrow (b, b')$
21: $*$ **update components of** $\boldsymbol{x}$ $*$
22: ncomp $\leftarrow$ ncomp $- 1$
23: **return** $\boldsymbol{x}$



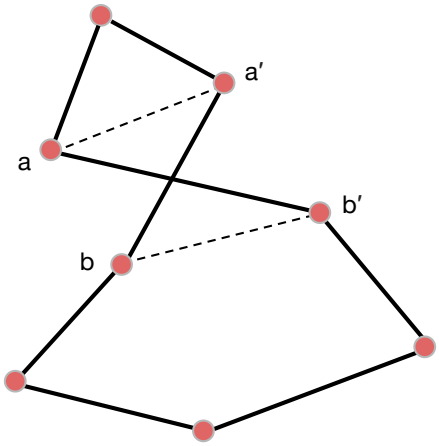
(a) First case: before merging



(b) First case: after merging



(c) Second case: before merging



(d) Second case: after merging

Figure 3.1: The two ways to merge connected components used by the complete_tour algorithm.

Since the overall algorithm is $O(n^2)$, the Heuristic Callback is called only when the depth of the decision tree is less than or equal to 10.

3.2.4 Generic Callback

The Generic Callback is a relatively new function that contains essentially two new features with respect to previous callbacks. The first one is that the callback, during a call to a user-defined function, doesn't stop the execution of the *Dynamic Search* algorithm. The latter is a branch-and-cut based algorithm, which is kept secret by CPLEX and which guarantee better performance than the traditional but robust branch-and-cut algorithm. The second feature is that with a single callback we can define each previous callback using the *context*. The context is a CPLEX structure that defines in which part of the brach-and-cut algorithm the solver calls the callback. An example of the use of Generic Callback is presented in Algorithm 7. This algorithm includes the three callbacks implemented in Algorithm 6, namely, Lazy callback, UserCut Callback and Heuristic Callback. The solver calls the generic callback function with the context CANDIDATE when an integer solution is found (line 8). In this case, we simply collected all function included in the Lazy callback and in the Heuristic callback (lines 9 - 15). On the other hand, after each fractional solution found by the solver the context used is RELAXATION (line 16). In this part of the algorithm, we collected all the function included in the UserCut callback together with the Concorde algorithms (lines 17 - 21).

Algorithm 7 Generic Callback

```
    Output: z* optimal solution to TSP on the input graph G
 1: function TSPopt(G = (V, E), c : E → ℝ⁺)
 2:     model ← ∗ Initialize variables and objective function ∗
 3:     optimizer ← ∗ Initialize the callback function within the solver ∗
 4:     z* ← optimizer(model)
 5:     return z*
 6:
 7: function GenericCallback(x, context, model)
 8:     if (context = CANDIDATE) then
 9:         comp ← ∗ number of components in the integer solution x ∗
10:         if (comp > 1) then
11:             ∗ Add SECs for each connected component to the model ∗
12:             x' ← complete_tour(x)
13:             ∗ Save x' and the thread index ∗
14:         if ∗ there is a solution available in the current thread ∗ then
15:         ∗ Add the TSP heuristic solution x' to the solver ∗
16:     if (context = RELAXATION) then
17:         comp ← ∗ number of components in the fractional solution x ∗
18:         if (comp > 1) then
19:             ∗ Add SECs for each connected component to the model ∗
```

20: **if** $(comp = 1)$ **then**
21: $*$ **Add SECs on the separated fractionary solution** $*$
22: **return**

3.2.5 Final comparison between callbacks

Chapter 4

Matheuristic

On this chapter we will illustrate two heuristic solution for the TSP problem that are based on CPLEX.

4.1 Hard fixing

When we have to resolve a large instance of TSP, CPLEX can require a large amount of time. Moreover in some practical application we may have to resolve the problem in a certain amount of time that we cannot exceed. So for this reason we can set a time limit for CPLEX, that means that the solver will return the best solution found within that limit. The hard fixing algorithm starts from this point. Let's set a time limit (10 minutes for example) and consider the solution obtained. Of course this solution with high probability is not the optimum solution, but it may contains some edges that belong to it. So the idea is to fix some edges, and pass the new model to CPLEX. In order to do this we can simply work on the bounds of the variables: if we set $1 \leq x(i,j) \leq 1$ the corresponding edge is fixed on the solution. The new model of course is simpler, because we have a smaller amount of variables, so when we resolve it with CPLEX (again with a fixed time limit) it may find a better solution. This procedure can also be iterate until a final time limit (like 1 hour) is reached: each time we release the variables that were fixed on the previous iteration (just reset the bounds to $0 \leq x(i,j) \leq 1$) and we choose a new set of edges to fix. Note that when the variables are fixed, due to the fact that the instance is smaller, CPLEX may resolve that model to its optimum before the time limit exceed. Of course the choice of which edge to fix and their number is quite relevant. In general a good idea is to set a fixing percentage and randomly choose the variables to fix. We can also decide to variate the percentage over the iterations, for example starting with an high value (like 90%) and then decrease it until zero. This mean that on the initial iterations the problem to resolve is very small so we may get a significant improve, while on the last iterations we have a big problem but lot of freedom to refine the solution. Of course this is only an example and other strategies can be implemented.

Our implementation of the Hard Fixing is presented in Algorithm 3. We used two

variables to manage the time limit of the algorithm. The variable *remaining_time* keeps track of the time left to the algorithm to compute the solution. At beginning the variable assumes the value of the original time limit given by the user (line 2). The variable *time_x_cycle* is the inner time limit and it is initially fixed to *time_limit/5* (line 3). Since with the inner time limit is very hard that an initial good solution will be computed, we provide to CPLEX a heuristic solution builded with the algorithm GRASP and refined with the 2-OPT algorithm (lines 4-6). The algorithm then proceeds solving the problem within a while loop (line 7). At each iteration, we initially set the inner time limit as CPLEX environment parameter (lines 8 - 11). Then the solution, within the time limit, is computed (line 13) with the time needed to compute it (lines 12 and 14). At this point, the variable *remaining_time* is updated subtracting from it the time taken by the solver (line 15).
The variable *percentage*, initially set to 90% (line 1), is updated only if after two consecutive iterations the solution found doesn't improve of at least the 10%. The update consists in subtracting 15% from the previous percentage of fixed variables (line 18). Then the algorithm proceeds by resetting the lower bound of all variables to 0.0 (line 19), randomly selecting the indices of variables to be fixed and setting the lower bound of them to 1.0 (line 20). This sequence is repeated as long as the user-set time limit is reached.

Algorithm 8 Hard Fixing

Input: $G = (V, E), c : E \to \mathbb{R}$
Output: A solution for the TSP problem

1: *percentage* ← *0.9*
2: *remaining_time* ← *timelimit*
3: *time_x_cycle* ← *timelimit / 5*
4: model ← ∗ **Initialize variables and objective function** ∗
5: first_solution ← ∗ **Heuristic solution with GRASP + 2-OPT** ∗
6: model ← CPXaddmipstarts(first_solution)
7: **while** (*remaining_time* > 0) **do**
8: **if** (*remaining_time* > *time_x_cycle*) **then**
9: *model* ← CPX_PARAM_TILIM(*time_x_cycle*)
10: **else**
11: *model* ← CPX_PARAM_TILIM(*remaining_time*)
12: *t1* ← *second()*
13: $\boldsymbol{x}$ ← solution(model)
14: *t2* ← *second()*
15: *remaining_time* ← *remaining_time* − *(t2* − *t1)*
16: **if** (*remaining_time* <= 0) **then**
17: **break**
18: ∗ **If the solution doesn't improve (+10%) twice in succession then** *percentage* −= 0.15. ∗
19: ∗ **Restore the default bounds for each variable** ∗

20: $*$ **Hard fixing of each variable according to *percentage*** $*$
21: **return** $\boldsymbol{x}$

4.2 Local branching

The strategy of this algorithm is similar to the one of hard fixing: each iteration fix some edges in order to give to CPLEX a simpler version of the problem. However, rather then randomly choose the variable to fix, we let CPLEX decide which edges to maintain. We can achieve this with the introduction of a new constraint: let's consider an initial solution H provided by CPLEX within a time limit, let x_e^H be the variables equals to one on this solution, we want the sum of these variables to be greater or equals than a percentage of the total number of edges. More formally:

$$\sum_{e \in E,\, x_e^H = 1} x_e \geq \alpha n$$

where α is the selected percentage and n is the size of the instance. In other word with this constraint we are telling CPLEX that we want a percentage of variables that are in the initial solution to be also in the final solution. At this point we can give the new model to CPLEX that will resolve it within the time limit, automatically deciding which variables to maintain. Of course this is not a valid constraint for the problem, cause we are not sure that the variables equals to one on a solution belong to the optimum.
Again we can iterate the procedure, each time removing the old constrain and generating a new one, based on the new solution, until a final time limit is reached. The fact that the selection of the variables to fix is not random, means that the CPLEX solution is deterministic, so it is not wise to execute two iterations with the same fixing percentage, cause we may resolve two times the same problem obtaining the same solution within the time limit. Finally, another shrewdness that is required, is that this algorithm works only if the fixing percentage is very high, so iteration with percentage of 50-60 or less should be avoided.

Our implementation of the Local branching is presented in Algorithm 4. The implementation is pretty similar to Algorithm 3, for this reason we only emphasise the differences between the two algorithms. At each iteration of the while loop (line 7), the variables of the solution are saved and a new local branching constraint is added to the model, removing the previous one. The known term of the constraint, that is the variable *percentage* in the algorithm, is initially set to: number of nodes $*$ 0.95 (line 1) and is updated with the same conditions as Algorithm 3, but in this case subtracting from the variable *percentage* the value: number of nodes $*$ 0.05 (lines 18 - 21). As in algorithm 3, the sequence is repeated as long as the user-set time limit is reached.

Algorithm 9 Local branching

Input: $G = (V, E), c : E \to \mathbb{R}$
Output: A solution for the TSP problem

percentage $\leftarrow |V| * 0.95$
remaining_time $\leftarrow$ *timelimit*
time_x_cycle $\leftarrow$ *timelimit / 5*
model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
first_solution $\leftarrow$ $*$ **Heuristic solution with GRASP + 2-OPT** $*$
model $\leftarrow$ CPXaddmipstarts(first_solution)
while (*remaining_time* > 0) **do**
    if (*remaining_time* $>$ *time_x_cycle*) **then**
        model $\leftarrow$ CPX_PARAM_TILIM(*time_x_cycle*)
    else
        model $\leftarrow$ CPX_PARAM_TILIM(*remaining_time*)
    t1 $\leftarrow$ *second()*
    $\boldsymbol{x} \leftarrow$ optimal_solution(model)
    t2 $\leftarrow$ *second()*
    remaining_time $\leftarrow$ *remaining_time* $-$ *(t2 − t1)*
    if (*remaining_time* $<= 0$) **then**
        break
    $*$ **If the solution doesn't improve (+10%) twice in succession then** *percentage* $-=$ $(0.05 * percentage)$. $*$
    $*$ **Save the indices of the solution variables.** $*$
    $*$ **If is not the first cycle remove the last row of the model.** $*$
    $*$ **Add the new local branching constraint with** $\alpha n = percentage$. $*$
return $\boldsymbol{x}$

4.2.1 Final comparison between Matheuristics

Chapter 5

Stand-alone heuristics

On this chapter we will illustrate some stand-alone heuristics. These algorithms are so called because they provide an heuristic solution without relying on CPLEX or any other solver. The literature in this case is huge: there are plenty of different approaches and each of them can have multiple variations. On this report we will cover some of the major ideas.
For all the experiments to collect data for the performance profiles and the comparison between different algorithms we used a dataset of 20 instances, with a size between 400 and 3000 points. A detailed list of all the instances used and their dimension can be found on the tables of the section 5.4.

5.1 Heuristic solution builder

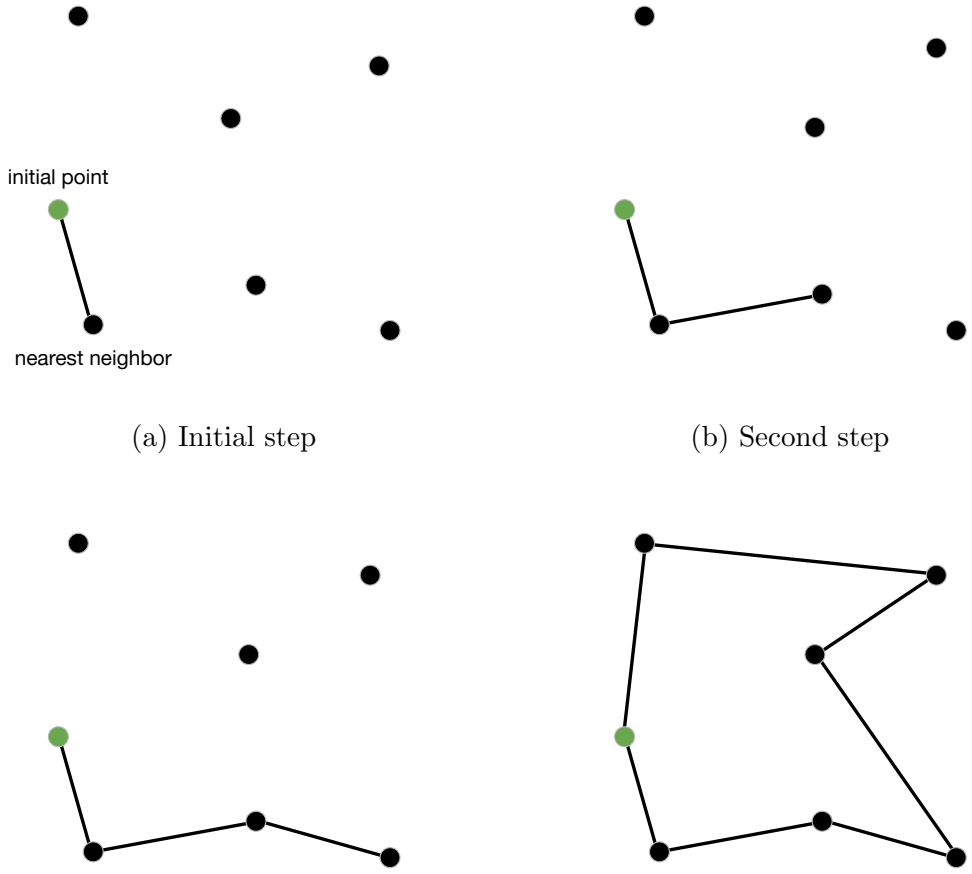
On this section we will illustrate some heuristics algorithm to generate in a rapid way a solution for the TSP problem. Of course the results provided by these algorithms are admissible solutions but not the optimum solution.

5.1.1 Nearest neighborhood

Nearest neighborhood is a greedy algorithm [1] that works in the following way: start from a random point that is considered the first visited node. Then pick as next node the nearest to the last visited (greedy choice) and iterate until all the nodes are visited. This procedure can be repeated until a time limit is reached, each time starting from a different random point, keeping in memory the best solution and eventually update it if a new one is found. The algorithm is pretty simple but provide an admissible solution (that of course is not the best one and in the very most of the cases is quite far from the optimum) in a short time.

On our implementation we decided to trade space for time. Since the operation to compute the nearest point is repeated several time, we decide to store for each point

[1] algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.



(a) Initial step

(b) Second step

(c) Third step

(d) Final result

Figure 5.1: Execution of nearest neighbor algorithm

the list of all others points sorted by their distances from that one. This require $O(n^2)$ additional space. Considering the time, instead, to sort the points we implemented a simple *insertion sort*, that is $O(n^2)$, so in total we need an initial $O(n^3)$ (This part could be further improved with a better sort algorithm like *merge sort* that is $O(n \cdot log(n))$, but since the initial time to order the nodes on our experiments was negligible, we decided to use insertion sort that was easier to implement). So, by paying this initial cost we were able to speed up each iteration of nearest neighborhood, in fact each time we needed to find the nearest point we just had to pick the first available one from the ordered list. Of course in the worst case this is $O(n)$, because it can happen that the firsts elements of the ordered list have already been visited, so I have to run across most of the list until I find an available point. However, what we saw in practice is that, except when there are left few nodes to visit, the first available element is always on the firsts position of the ordered list so the operation to find the nearest neighbor became $\Theta(1)$ in most of the cases (for

example, we tried on multiple instances between 100 and 500 points and the average number of access to the ordered list to find the nearest point was between 3 and 6).

The major problem of this technique is that visiting always the next closest point leads to the creation of lots of intersection between edges, that of course are inefficient. A possible solution to improve this initial result will be shown on the next section dedicated to the refining algorithms.
Another problem is that this algorithm is deterministic, so this mean that if it is run twice starting from the same initial point, it produce the same solution. The consequence of this is that the number of different circuit that it can explore is limited to the total number of nodes.

5.1.2 GRASP

GRASP is a variation of nearest neighborhood that introduce a random component. On each iteration, rather than picking the nearest point to the last visited, select the three closest points and choose one of them at random as next visited vertex. (Or alternatively assign a different probability to each of the three points, for example 0.6 to the closest point and 0.3 and 0.1 respectively to the second and the third).

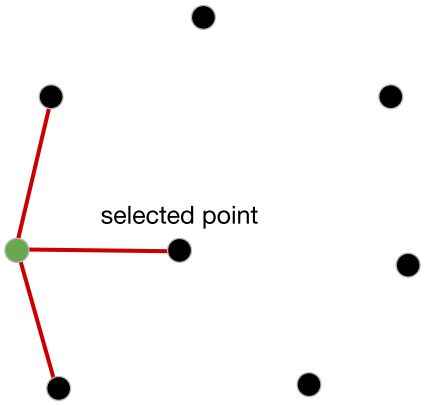
Algorithm 10 GRASP

Input: Instance I of the TSP
Output: Admissible solution for the instance

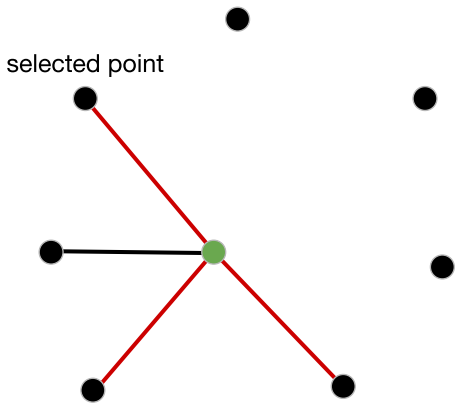
```
1:  cost(bestSolution) ← ∞
2:  while ! termination condition do
3:      S ← empty list
4:      notVisitedList ← all vertices
5:      currentVertex ← pick 1 random vertex from I
6:      notVisitedList ← remove(currentVertex)
7:      S ← add(currentVertex)
8:      for i = 1, i < N, i++ do
9:          v1, v2, v3 ← pick 3 closest and not visited vertex from currentVertex
10:         nextVertex ← choose at random one between v1, v2, v3
11:         notVisitedList ← remove(nextVertex)
12:         S ← add(nextVertex)
13:         currentVertex ← nextVertex
14:     if cost(S) < cost(bestSolution) then
15:         bestSolution ← S
16: return S
```

Our implementation was almost the same of nearest neighborhood, with the construction for each point of the initial list of all other points sorted by their distance from that point, with the only difference that first three available points are picked from the list, rather than only one, and the next visited point is chosen at random

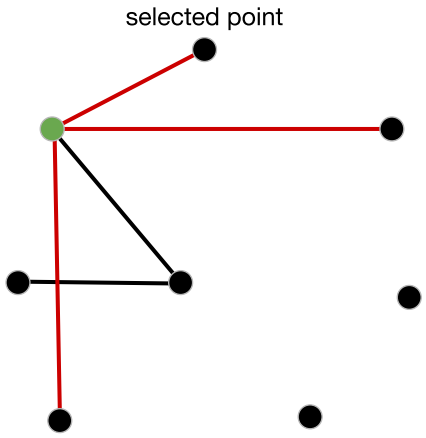
from them. The introduction of the random component means that the probability to repeat the same sequence starting from the same point is very minimal, so GRASP can explore lots of different circuit resolving the problem of determinism of nearest neighborhood.



(a) Initial step



(b) Second step



(c) Third step

Figure 5.2: Execution of 3 steps of GRASP. Each time the 3 closest points are considered and one is picked at random.

5.1.3 Insertion heuristic

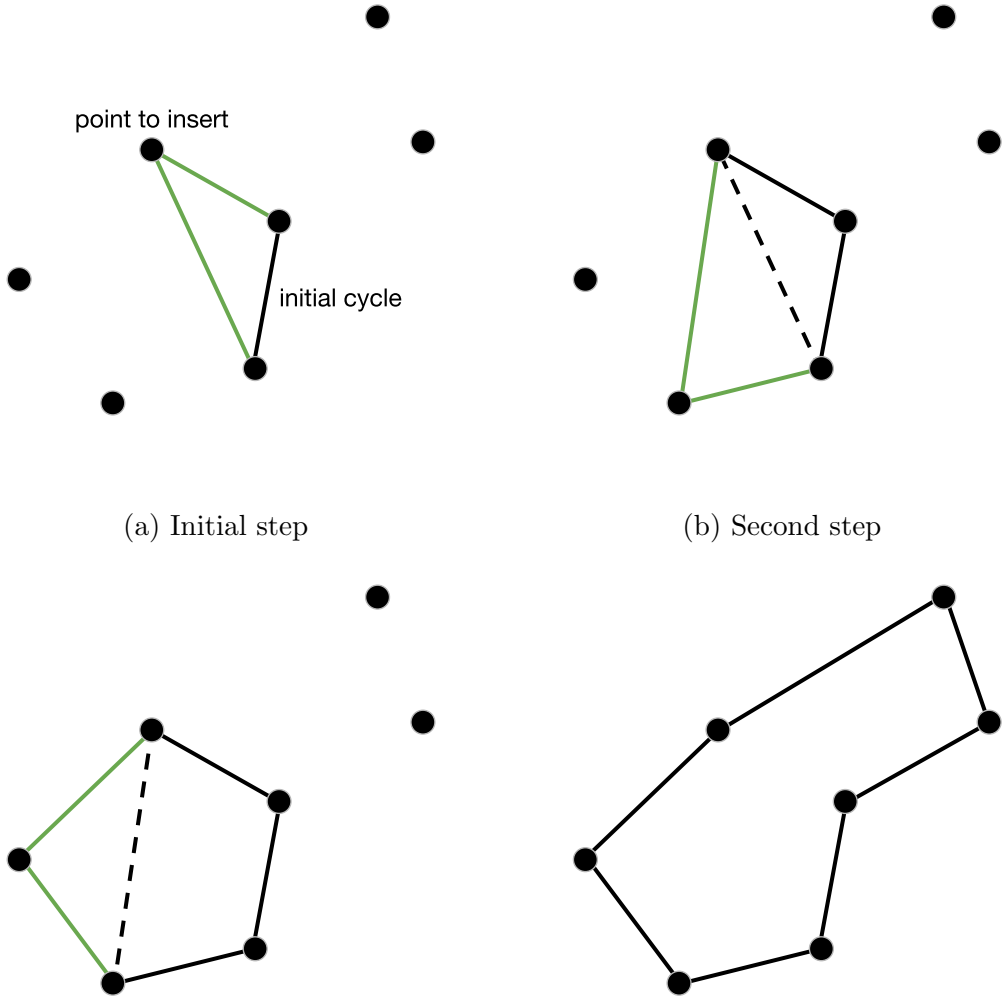
This algorithm starts from a simple circuit made with only two points. At each iteration, for each of the remaining vertices to insert, compute the minimal extra mileage. The extra mileage is the cost to insert a point on the circuit. So, for

example, if we want to insert the vertex z between vertex x and vertex y, the extra mileage is: $c_{xz} + c_{zy} - c_{xy}$, where c_{ij} is the cost of the edge (i, j). The fact that for each point we search the minimal extra mileage means that we are looking to the best position where insert that point. Insert vertex that has the best extra mileage into the partial circuit and repeat the procedure until all the points are inserted (Figure 5.3). The algorithm than can be executed several time to explore new solutions, starting every time from different initial points and updating the incumbent if necessary.
The cost to compute the best extra mileage is $O(n^2)$, so this mean that the entire algorithm is $O(n^3)$.
It is also possible to add the GRASP rule, so rather then selecting the point with the best extra mileage, find the three smallest extra mileages and choose at random between them, in this way the algorithm doesn't converge to the same solution starting from the same initial circuit.

Algorithm 11 Insertion heuristic

```
    Input: Instance I of the TSP
    Output: Admissible solution for the instance
 1: cost(bestSolution) ← ∞
 2: while ! termination condition do
 3:     S ← empty list
 4:     S ← add 2 initial vertices
 5:     while ! all point are inserted do
 6:         bestExtraMileage ← ∞
 7:         for each point p to insert do
 8:             minExtraMileage ← ∞
 9:             for each edge e ∈ S do
10:                 extraMileage ← computeExtraMileage(p, e)
11:                 if extraMileage < minExtraMileage then
12:                     minExtraMileage ← extraMileage
13:             if minExtraMileage < bestExtraMileage then
14:                 bestExtraMileage ← minExtraMileage
15:         S ← insert point with bestExtraMileage in the best poisiton
16:     if cost(S) < cost(bestSolution) then
17:         bestSolution ← S
18: return bestSolution
```

On our implementation to select the initial circuit we decided to pick at random one point and then to select the furthest point from that one. (Of course this is not the only solution: other possibility are, for example, to pick a vertex and its closest one, or to choose two points completely at random.) I addition, at the beginning of the algorithm, we built a matrix where the element i j contains the distance between vertices i and j (So in total we payed an additional $O(n^2)$ in space). We made



(a) Initial step

(b) Second step

(c) Third step

(d) Final result

Figure 5.3: Execution of insertion heuristic algorithm

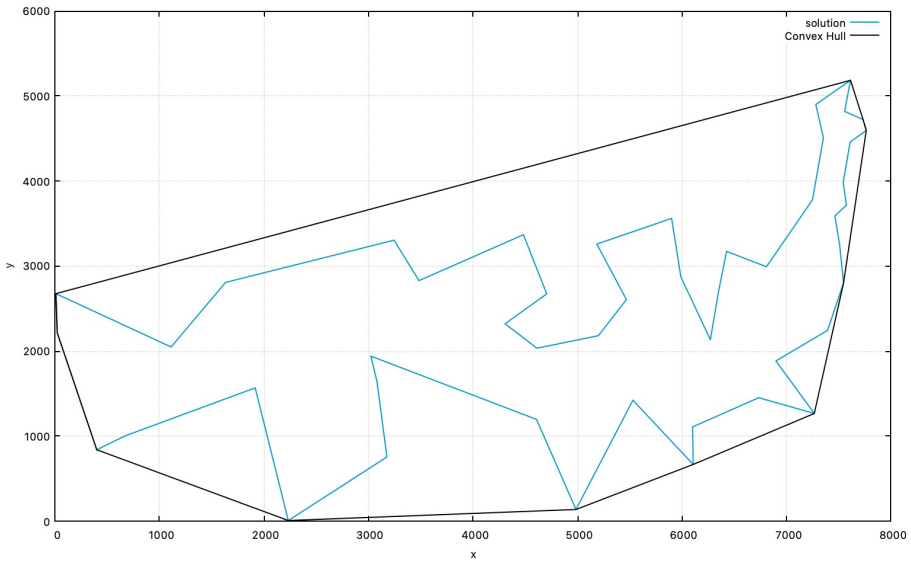
this choice because on this algorithm distances between points are computed several times and in addition the same distance can be calculated more than once.

5.1.4 Insertion with convex hull

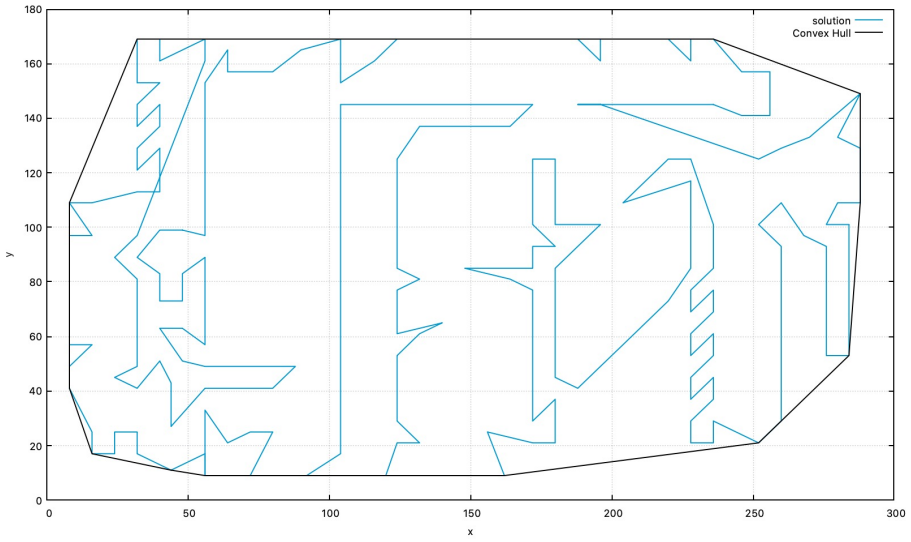
This is a variation of the insertion algorithm. It works in the same way but the initialization is different: rather than starting from an initial cycle made of two random points, compute the convex hull [2] of the instance. Then iterate and insert

[2]In geometry, the convex hull or convex envelope or convex closure of a shape is the smallest convex set that contains it.

all the remaining points, that are all inside the convex hull, in the same way of Insertion heuristic (Figure 5.4).
On our implementation, to find the initial convex hull, we used the Graham Scan algorithm, freely provided by the website www.geeksforgeeks.org. This algorithm is $O(n \cdot log(n))$ and it is performed only once at the beginning so it doesn't impact the overall execution time for the insertion that is $O(n^3)$ as explained on the previous subsection. For more information about the Graham Scan consult `https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/`



(a) problem att48



(b) problem a280

Figure 5.4: Execution of the insertion with convex hull on 2 different problems. It is possible to see the initial convex hull and the final result when all the points inside it are inserted

5.1.5 Final comparison between solution builder

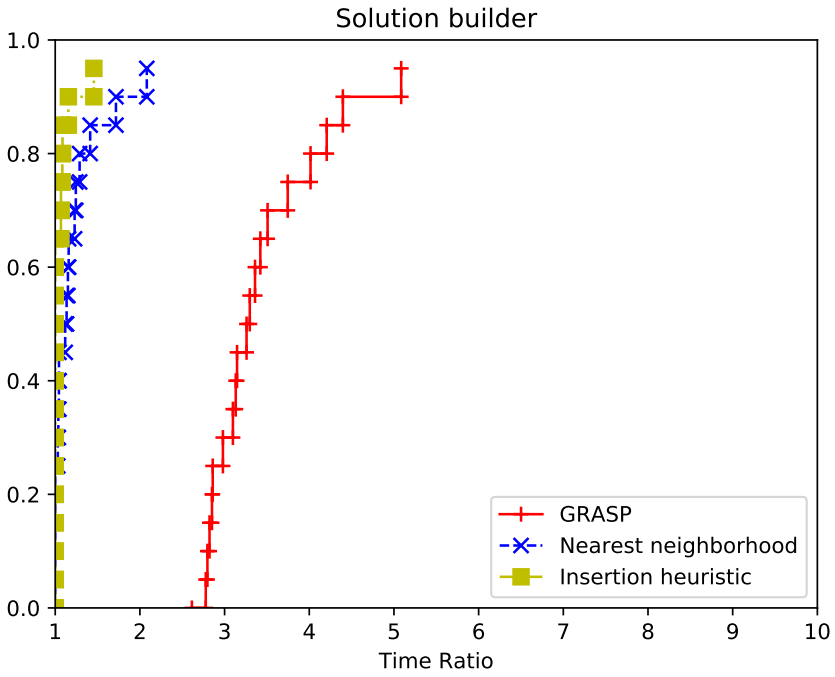


Figure 5.5: Performance profile of the solution builders

On Figure 5.5 the final confrontation between the heuristic solution builders. To obtain these result we test the algorithms on the dataset of 20 instances with a time limit of 5 minutes.

5.2 Refining algorithms

Refining algorithm are so called because they start from an admissible solution that could by provided by any other algorithm and they try to improve it with a series of iteration.

5.2.1 TWO-OPT

The first algorithm we present is TWO-OPT (or 2-OPT). To explain this technique, let's consider two edges, the first delimited by vertices a and b and with cost c_{ab}, the second delimited by c and d and with cost c_{cd} (Both edges belong to an admissible solution provided by any algorithm). Now let's remove these edges and reconnect the vertices in a different way: as result we obtain the edges a, c and b, d with cost respectively c_{ac} and c_{bd}. Now lets compute:

$$\Delta = c_{ab} + c_{cd} - (c_{ac} + c_{bd})$$

that is the cost of the initial removed edges minus the cost of the new edges. If $\Delta > 0$, it means that the new edges are better than the older and the swap is worth because it reduce (improve) the objective function. Of course the new solution produced is still admissible (Figure 5.6).

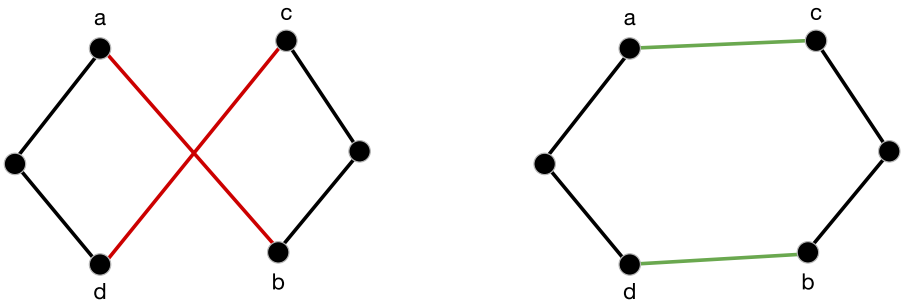


Figure 5.6: The 2-OPT move, edges in red are removed and replaced by green edges.

2-OPT works in this way: start from an admissible solution, consider all the possible couple of edges (that are in total $O(n^2)$) and for each of them compute the Δ (That can be calculated in $\Theta(1)$). Then pick the largest Δ, remove the relative edges and reconnect the vertices as previously explained. After this operation, one of the two parts of the cycle that has been divided by the swap, need to be retraced in order to invert the visit sequence of the vertices and reconnect it to the new edge (This is $O(n)$, but since it is done after a $O(n^2)$ doesn't change the overall cost of the swap). Repeat the procedure until all the Δ are negative, or a fixed time limit is reached.

On our implementation we followed the procedure described above, with the addition of a matrix to store all the distances between points, to avoid to recompute

Algorithm 12 2-OPT

Input: Admissible solution S
Output: Best solution found

```
1: while ! termination condition do
2:     best∆ ← ∞
3:     for each couples of edge v_ab, v_cd ∈ S do
4:         ∆ ← c_ab + c_cd − (c_ac + c_bd)
5:         if ∆ < Best∆ then
6:             best∆ ← ∆
7:     if best∆ > 0 then
8:         S ← perform move with best∆
9:     else
10:        return S
11: return S
```

them more than once.
This algorithm is very good to remove edge intersections, like the ones produced by nearest neighborhood (or GRASP). Here in Table 5.1 some examples.

Instance	**GRASP** (5 sec)	**2-OPT** (5 sec)	**Improve**
att48	41806	35522	15%
eil101	879	691	21%
ch130	9801	6347	35%
ch150	10477	7278	30%
a280	4546	2886	36%
lin318	76940	46235	39%
att532	155010	97487	37%
pr1002	492311	285078	42%

Table 5.1: Values of the objective functions after running on each instance GRASP algorithm for 5 seconds and then 2-OPT for other 5 seconds.

These results were obtained after running GRASP for five seconds and then refining the solution with 2-OPT for other five seconds. Of course this dataset is very small but it gives the idea of the power of this technique, we have, in fact, a significant improve of the objective function in all the cases. To notice that the improve increase with large problem.
It is possible to visualize the action of 2-OPT by plotting the solution before and after the refining like in Figure 5.7.

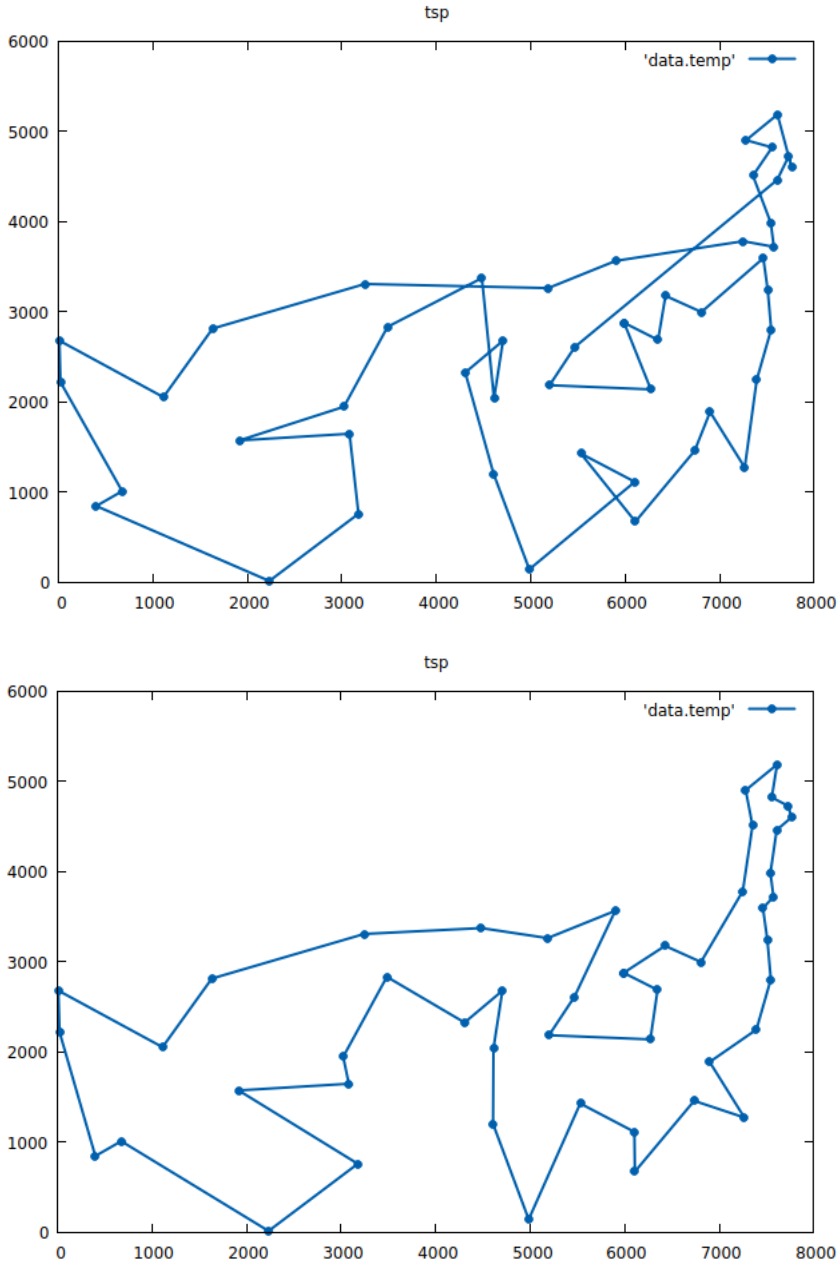


Figure 5.7: Solution of 5 sec of GRASP before (top image) and after (bottom image) a refining of 5 sec

5.2.2 THREE-OPT

3-OPT is a variation of 2-OPT that consider 3 edges for the swap rather than only two. This time there are multiple way to reconnect the vertices of the 3 removed edges, in particular for a triad there are 7 possible moves (Figure 5.8). Moreover, we have to consider all the possible triad to find the best one and this is $O(n^3)$. So in general 3-OPT is more complicated to implement and due to the fact that is

cubic doesn't work well with big problems, for example with instances with more than 10000 points.

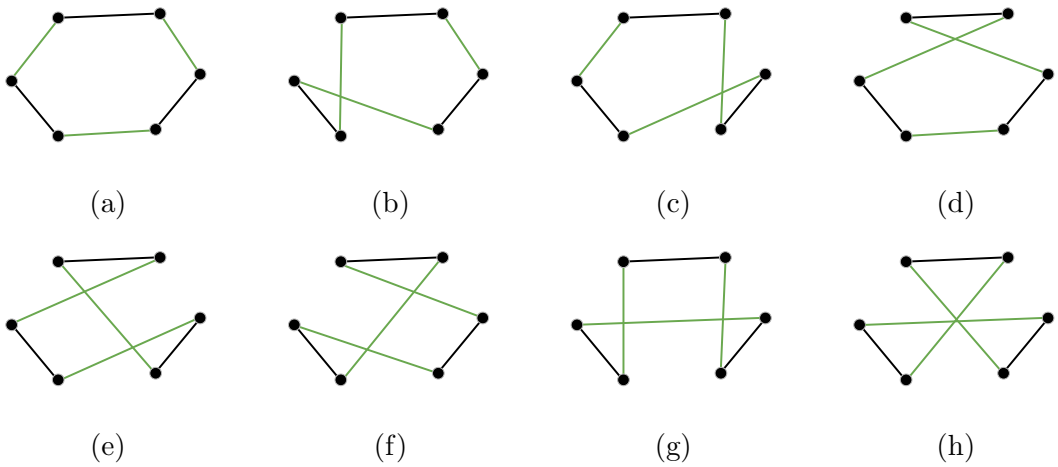


Figure 5.8: All the possible 3-OPT moves given a triad of edges.

We also repeated the same experiment we did for the 2-OPT, so we run on different problems GRASP for 5 seconds and then 3-OPT on the result for other 5 second. The results are shown on Table 5.2. Again the dataset is small because the only goal was to show the power of this refinement algorithm and not to compare it with other techniques. Observing the table we can notice that the improve for the smaller problems is almost the same, or sometimes better than the one with 2-OPT, but more importantly we can notice that the algorithm scales bad with big problems due to the fact that it's cubic. In fact 5 seconds is a relatively small amount of time and in this short period the number of 3-OPT moves that the algorithm is able to perform is small so the improve is poor.

Instance	**GRASP** (5 sec)	**3-OPT** (5 sec)	**Improve**
att48	42086	33588	20%
eil101	875	665	25%
ch130	9894	6373	35%
ch150	10503	6828	35%
a280	4588	3247	29%
lin318	75526	56496	25%
att532	153954	144295	6%
pr1002	487157	481381	1%

Table 5.2: Values of the objective functions after running on each instance GRASP algorithm for 5 seconds and then 3-OPT for other 5 seconds.

5.3 Metaheuristics

A metaheuristic is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity [3]

5.3.1 Multi-start

After performing a refinement technique, like 2-OPT, if the time limit is not reached, the algorithm stop on a local minima of the objective function. Of course at this point we want to escape from this local minima and search for a better one that hopefully is also the optimum. A possible solution is the multi-start approach, so each time we restart from the beginning, providing another initial solution and refining it. On each iteration, if the algorithm that provide the initial solution is well randomized (like GRASP) we will stop on a different local minima. So we can keep track of the best one and return it when the time limit for this procedure is reached. The main problem of this technique is that is not very efficient because each time I restart all the information we obtained on the previous local minima is unused. More powerful technique will be shown on the following subsections.

5.3.2 Variable neighborhood search

Variable neighborhood search (VNS) provide a simple yet powerful way to escape the local minima. Let's assume that we have completed the refinement using 2-OPT. At this point we can perform a random move with a bigger research radius, like a 3-OPT or a 5-OPT for example. Since this move is completely random, it will make the solution worse but it will also provide the "kick" that we need to escape the local minima. Now we can restart with 2-OPT until a new minima is found (that could also be the optimum).
In synthesis, VNS, works with two phases that alternate continuously: an intensification phase (2-OPT) where the algorithm move toward a minima and a diversification phase where VNS perform a k-OPT random move to exit from the minima.
On our implementation we use the technique showed in Figure 5.10. We select the edges to remove (that in our example are three) and then we reconnect the vertices in a specific way, so that the reversal of some edges to reconstruct the cycle (the one described in 5.2.1) is never required. In particular, because we can assign to the cycle a direction of travel, for each edge that we want to remove we can distinguish an initial vertex (the ones in red) and a final vertex (the ones in green). The idea is to always reconnect an initial vertex to a final vertex so the direction of travel is preserved. So for example we can start from the initial vertex of the first edge to

[3] R. Balamurugan; A.M. Natarajan; K. Premalatha (2015). "Stellar-Mass Black Hole Optimization for Biclustering Microarray Gene Expression Data". Applied Artificial Intelligence an International Journal.

Algorithm 13 VNS

Input: Admissible solution S
Output: Best solution found

```
1:  bestSolution ← S
2:  while ! termination condition do
3:      bestMove ← find best 2-OPT move of S
4:      if Δ(bestMove) > 0 then
5:          S ← perform bestMove
6:      else
7:          S ← perform random k-OPT move
8:      if cost(S) < cost(bestSolution) then
9:          bestSolution ← S
10: return bestSolution
```

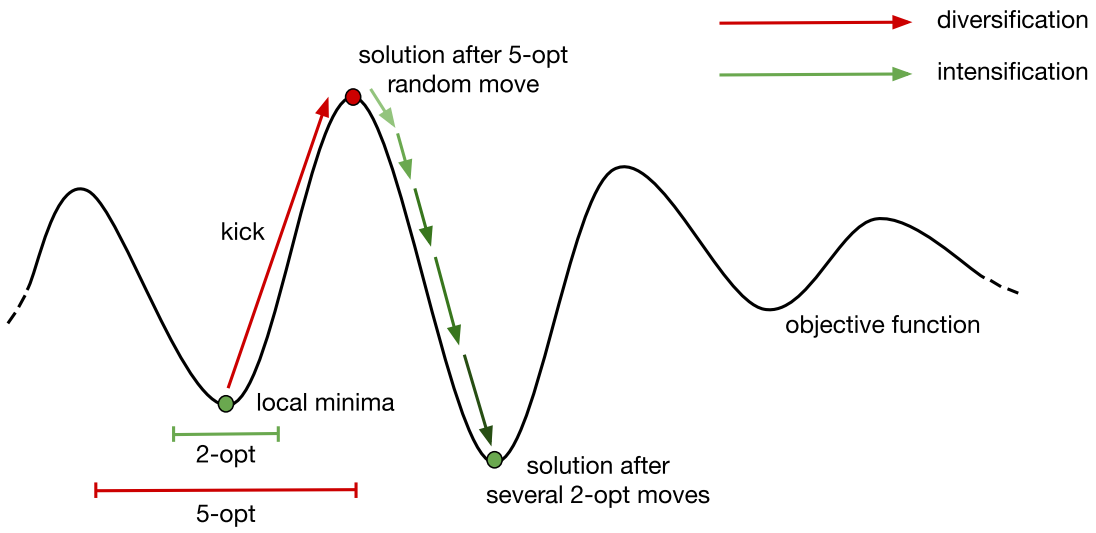


Figure 5.9: Representation of an objective function during VNS and the "kick" of a 5-OPT random move

remove and connect it to the final vertex of the second edge to remove. Then we pick the initial vertex of the second edge and we connect it to the final vertex of the third and so on, until the cycle is reconstructed. Of course this is valid for k edges, so on our implementation we can perform a k-opt random move, where k is establish at the beginning of the execution.

5.3.3 Tabu search

Created by Fred W. Glover during the 80s, Tabu search provide an alternative procedure to VNS to escape from local minima. Let's assume that we have performed

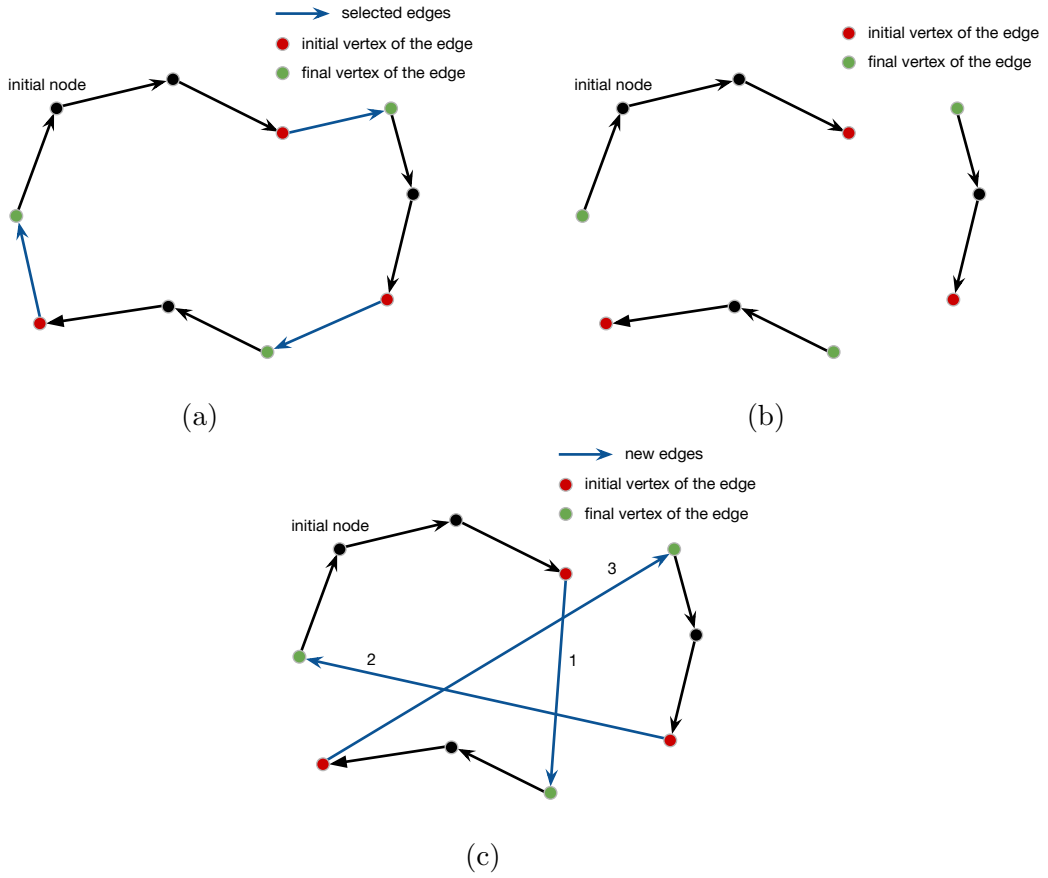


Figure 5.10: Passages for a k-OPT random move (In this case a 3-OPT move).

a refinement and we have reached a local minima, let's call this solution x_k. At this point, every 2-OPT move will make the objective function worse. So we choose and perform the least pejorative move, obtaining the solution x_{k+1}. Now, if we make another move following the 2-OPT rule, the algorithm will return to the solution x_k by simply swapping again the edges used on the last step, cause this is the only move that improve the objective function. To avoid this, the idea is to insert these edges on a tabu list that is a list of edges that cannot be used for a 2-OPT move. This means that the move from x_k to x_{k+1} became forbidden. So we repeat this last step, performing the least pejorative move and each time inserting the edges used in the tabu list, until we reach a point where the objective function will return to improve. Of course we don't insert edges into the tabu list if the move is not pejorative.
To describe the algorithm we asserted that the tabu list is made of couples of edges, but this was only for a description purpose. In fact the general idea for this algorithm is that the tabu list can be made of any element that prevent to return back after a pejorative move. So for the TSP problem we can use, for example, couples of edges, a single edge or a single point.

Algorithm 14 Tabu search

Input: Admissible solution S
Output: Best solution found

```
1: bestSolution ← S
2: while ! termination condition do
3:     if tabuList is full then
4:         tabuList ← empty list
5:     bestMove ← find best and not forbidden 2-OPT move of S
6:     if Δ(bestMove) > 0 then
7:         S ← perform bestMove
8:     else
9:         S ← perform bestMove
10:        tabuList ← add(bestMove)
11:    if cost(S) < cost(bestSolution) then
12:        bestSolution ← S
13: return bestSolution
```

The handle of the tabu list is very important and different strategies can have a big impact on the algorithm. Of course if we continue inserting element we will reach a point where there aren't possible moves. So we limit the size of the tabu list. The size is called tenure. When the list is full, the older element are removed. The tenure is another parameter to set. A possible way to do this is to set a minimal and a maximal value and let the tenure oscillate among them during the algorithm. So we will have diversification phases when the tenure is big and there are lot of forbidden move and intensification phases when the tenure is small and the algorithm has lot of freedom. This is called reactive tabu search. Another shrewdness is that if we encounter a move that is forbidden but that improves the incumbent, we will not perform that move but we will update the value of the incumbent in any case (Aspiration criterion).

On our implementation we use a fixed-size tabu list, with the tenure that is a parameter decided by the user. The list records only one vertex involved in the move: so for example if we perform a pejorative move and we substitute the edges v_{ab} and v_{cd} with v_{ac} and v_{bd}, only the vertex a is stored in the tabù list. When we find a good move, we check that all the 4 vertices involved are not in the tabu list, so even if we record only one point per move, this is sufficient to avoid to return back after a pejorative move. The check if a move is forbidden require a $O(k)$, where k is the tenure. So we perform this control only to moves that are better than the best move found until that moment (It's useless to check pejorative moves or moves that are not candidate to be selected as best move).
Finally we also implemented the aspiration criterion, so if a forbidden move leads to an improve of the incumbent, we keep that solution and we update the incumbent too.

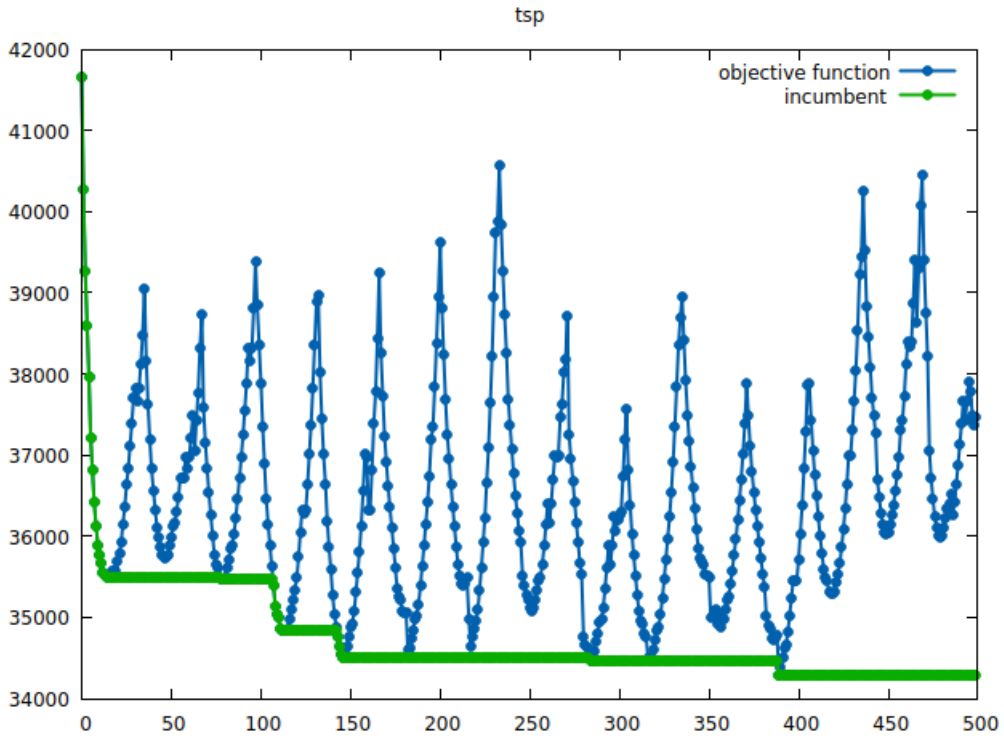


Figure 5.11: Objective function and incumbent during the execution of tabu search

5.3.4 Simulated Annealing

This metaheuristic algorithm take inspiration from the annealing technique in metallurgy that consist on heating the material and then cooling it in a controlled way. When the temperature of the materials is high, the atoms inside it it vibrates and move in a chaotic way. As the temperature decreases, the atoms reduce their state of agitation until a stable point is reached.
The simulated annealing works in a similar way. Let's establish a parameter, the temperature T, that initially has an high value that gradually reduce during the iterations of the algorithm. The algorithm starts from an already provided solution. On each iteration we perform a random 2-OPT (or 3-OPT) move that of course can be also pejorative. The fundamental idea is that the probability to accept the move is related to the temperature: when T is high, the system is chaotic and we accept with high probability all the random move. As the system get cooler we increase the probability to reject a bad move (so we increase the probability to perform a good move). When the temperature is close to 0, we only make meliorative moves until we reach a minima.
Considering the objective function, we have that, when T is high, the algorithm is randomly exploring solutions. Only when T became low the algorithm moves toward a minima that hopefully is the optimum.
Of course the choose of the cooling schedule and the function that regulate the probability to accept a move with respect to the temperature are pivotal and can

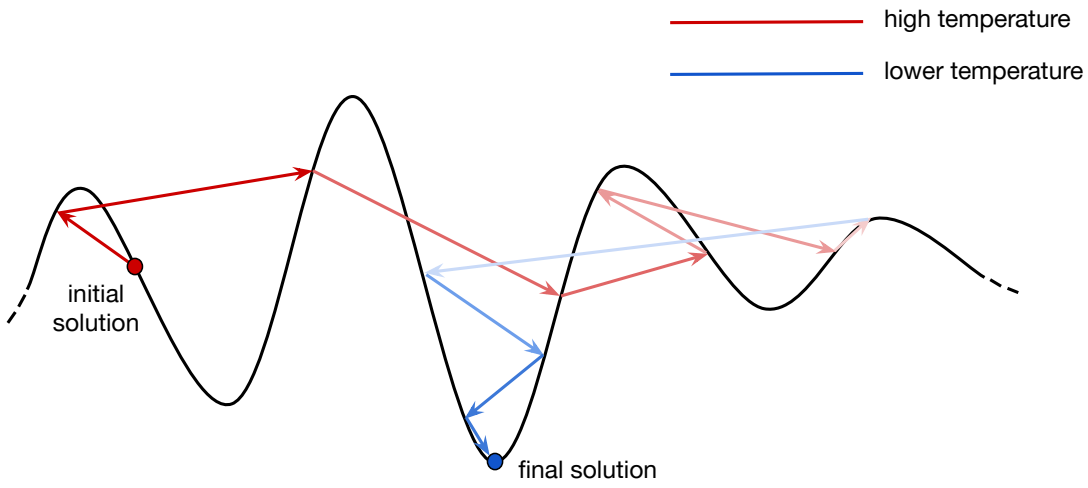


Figure 5.12: Example of the work of the simulated annealing. Note the initial random exploration with high temperature and the move toward the minima with a lower temperature.

have a big impact on the performance of the algorithm itself.
On our implementation we followed the procedure showed in the paper that we are going to briefly explain. Let X be the initial solution and Y a solution produced after a random 2-OPT move on x, let $f(X)$, $f(Y)$ be the cost respectively of X and Y, let p_0 be the initial probability. On each iteration, until the temperature list is full, generate a new Y. If $f(Y) > f(X)$ (pejorative move) compute a temperature with the following formula:

$$t = \frac{-|f(y) - f(x)|}{\ln(p_0)}$$

otherwise, if $f(Y) < f(X)$, put X = Y and then compute t in the same way. Now to describe the remaining part of our implementation let's consider N, that is a parameter decided by the user that indicate the maximum number of iteration where we use the same temperature. Let also i = 0, 1... N be a counter for those iteration. We start by extracting T_{max} from the temperature list, this will be the temperature of the system for the next following N iteration. On each iteration we generate a random 2-OPT move: if it is meliorative the solution is simply updated. If it is pejorative we compute a probability p_i defined as:

$$p_i = \exp\left(\frac{-(f(Y) - f(X))}{t_{max}}\right)$$

then we generate a random number $r_i \in [0, 1)$ and if $r_i < p_i$ the move is accepted. At this point, if we accept a bad move, we need to calculate another temperature T_i, defined as:

$$T_i = \frac{-d_i}{p_i}$$

Algorithm 15 Simulated Annealing

Input: Admissible solution S
Output: Best solution found

```
1: let p be the probability to accept a bad move
2: let T be the temperature
3: while ! termination condition do
4:     Δ ← random 2-OPT move
5:     if Δ > 0 then
6:         S ← perform the move
7:     else
8:         p ← f(T)
9:         r ← random number ∈ [0, 1)
10:        if r < p then
11:            S ← perform the move
12:    if temperature change condition then
13:        decrease T
14: return S
```

where d_i is the difference between $f(Y)$ and $f(X)$ on iteration i. We need also to keep in memory a counter c for each time we have performed a pejorative move. When we reach N iteration, we have to decrease the temperature: we compute the mean

$$T_{mean} = \frac{1}{c} \sum T_i$$

and we substitute T_{max} with T_{mean}. Since $T_i < T_{max}$ we have also that $T_{mean} < T_{max}$ so with the substitution we decrease the temperature. Now we pick another T_{max} and we repeat the entire procedure until the algorithm terminate.

5.3.5 Genetic Algorithm

The genetic algorithm is inspired by the evolution theory, formulated by Charles Darwin during the 18^{th} century. The algorithm starts with a population where each individual that belong to the population itself is a solution to the problem, so in our case each individual is a solution for the TSP. Each individual has also a fitness that is defined as the cost of the solution. On each iteration (also called epoch), the individuals that have the highest fitness are removed from the population: maintaining the parallelism with the evolution theory we can say that the individuals that didn't adapt to the environment (high fitness) died. On each iteration we have also that couples of individuals reproduce and the new elements substitute the ones that we have removed (so the population size remain constant). The children generated inherit characteristics from both their parents: a simple mechanism is just to pick part of the edges from one parent and the remaining from the other. Of course in this way the children is not a solution of the TSP problem cause some node may be visited more than once or may miss, so a repair function is required. In addition,

at this point, a mutation function can be added: this mean that a children with probability x (that in general is low) can suffer a modify that hasn't any relation with its parents, like a random exchange of two vertices in the visit sequence. We can introduce this shrewdness to emulate in the algorithm the mutation mechanism that happens also in real life
After some iteration we have that the average fitness improve due to the fact that we remove bad solution and we generate new solution from the better ones. When the algorithm end, the champion of the population, that is the solution with the best fitness, is the final result.

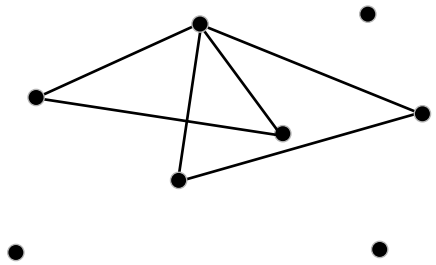
Algorithm 16 Genetic

Input: Population P_t of solutions S, N number of new element per epoch
Output: Best individual of the population

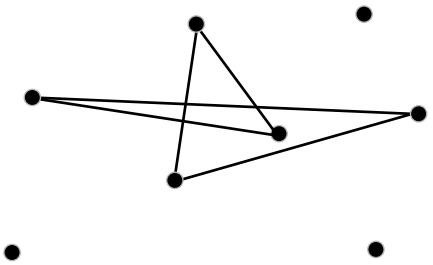
```
1:  t ← 0
2:  bestSolution
3:  fitnessList
4:  fitnessList, bestSolution ← computeFitness(P_t)
5:  while ! condition of termination do
6:      for i = 0, i < N, i++ do
7:          removeElement(P_t)
8:          update(fitnessList)
9:      for i = 0, i < N, i++ do
10:         S ← generateNewChildren(P_t)
11:         repair(S)
12:         fitnessList ← add(fitness(S))
13:         if fitness(S) < fitness(bestSolution) then
14:             bestSolution ← S
15:     t ← t+1
16: return bestSolution
```

Our first implementation of this algorithm was pretty simple. We generated a population of a size decided by the user using the GRASP algorithm. We created a probability mass function (PMF) normalizing the sum of the fitness, so on each iteration we randomly chose the elements to remove based on that mass function. This means that during the firsts iterations, when all the individual have almost the same fitness, the element are almost randomly remove, while when the algorithm progress and there is a more evident difference between good and bad individuals, with high probability only the worst solutions are removed. The number of element to remove on each iteration (that is equal to the number of children to generate) is a parameter decided by the user.
To reproduce two elements we used the following technique: we chose a random number x between 0.3 and 0.7. Then we picked the first x visited node from one of the parent and the remaining from the other. At this point we used a repair function

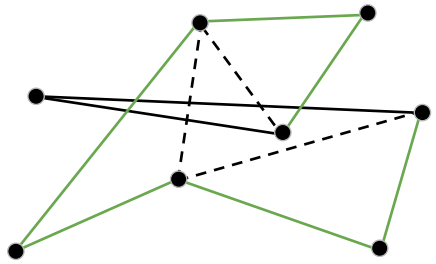
(Figure 5.13) that remove all the points visited twice and insert the missing ones using the insertion algorithm described in 5.1.3.



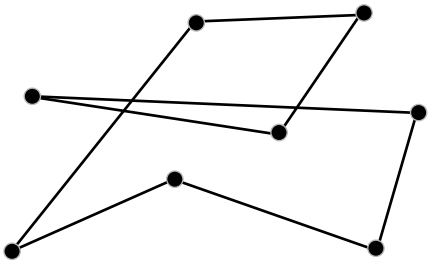
(a) Non repaired children



(b) Twice-visited vertices removal



(c) Insertion of remaining element



(d) Final result

Figure 5.13: Example of the steps of a repair function

The couples to reproduce are simply chosen at random from the population and we didn't implement any mechanism to control that same individual aren't choose more than once. We made this choice because on this algorithm the population have often big size, in general more than 10000 individuals, so the probability to pick the exactly same individuals more than once is very low. After some run, plotting the average fitness and the best fitness, we could observe the expected behaviour (Figure 5.14): the average fitness continuously decrease until it get closer to the best fitness (that improve only sometimes).

The problem of our implementation became evident after testing it on different instances: the solutions had lot of intersection (like the ones generated by the GRASP algorithm) that of course don't belong to the optimum. An example of this can be seen on Figure 5.15. So we tried to introduce the 2-OPT someway inside the genetic. From this point, to distinguish the different versions of the genetic that we realized, we will call the implementation just described *genetic 1*.
A possibility that we first considered but we immediately discarded was to just ap-

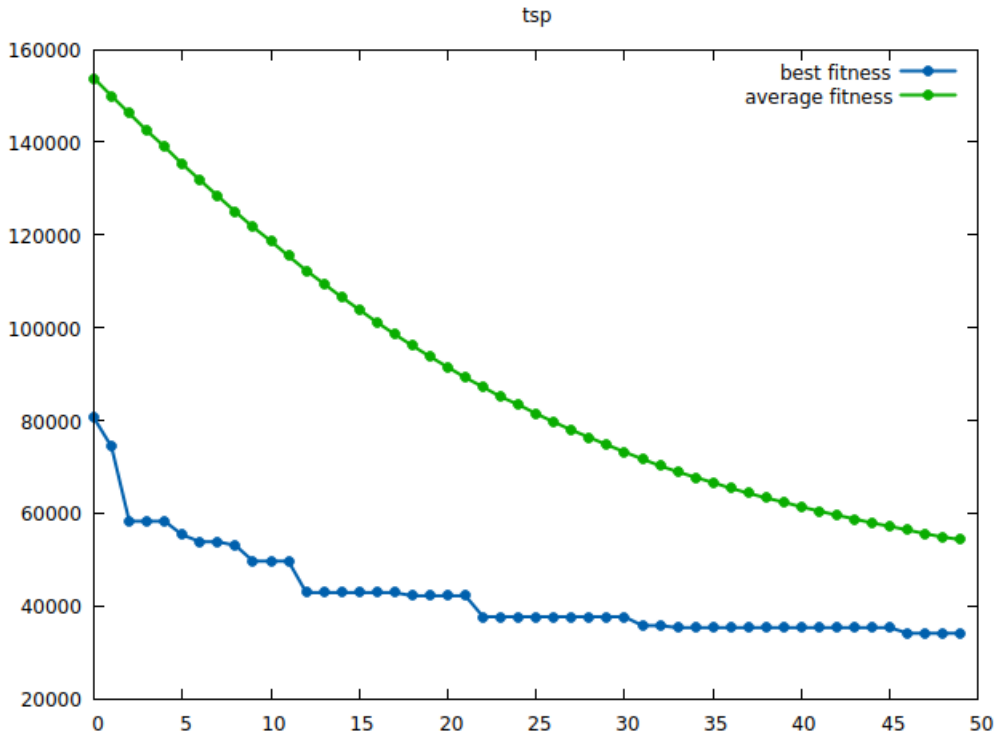


Figure 5.14: Plot of the average fitness and best fitness during the execution of a genetic algorithm.

ply the 2-OPT algorithm on the final solution provided by the genetic. This didn't make lot of sense because we did a lot of effort to produce a result with the genetic and then we discard a consistent part of it using the 2-OPT: a simple GRASP plus 2-OPT would be more efficient. Then we decided to add the 2-OPT to the repair function. So each children, after being repaired was also refined using the 2-OPT rule. Let's call this version *genetic 2*. We immediately noticed a large reduction of the number of epochs executed in the same amount of time with respect to genetic 1. This is because the populations used in the genetic algorithm are large and even if 2-OPT is quite fast it has to be executed a lot of time.
Finally we tried an hybrid approach between genetic 1 and genetic 2: after the repair function the refinement is executed with a probability x decided by the user. With this version each epoch is faster than genetic 2 because the 2-OPT is executed less time, however the price of this is the introduction of another parameter that is the probability x.

We tested these three versions on the dataset to find which ones is the best. We establish a time limit of 10 minutes and a population size of 2000 elements with 200 new elements each epoch. The 2-OPT, when applied inside the repair function, has a time limit of one second. Note that we were forced to use a population with a limited size, in fact after few tests we discovered that it's very difficult to handle a

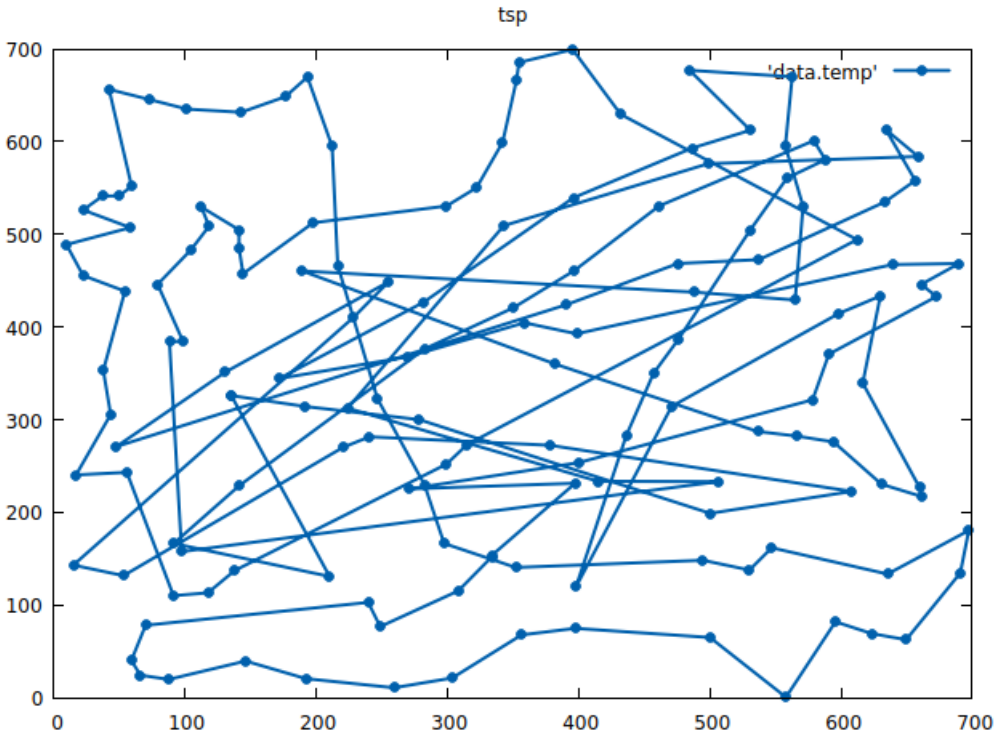


Figure 5.15: Solution of the instance ch150 provided by our first implementation of the genetic. Note the big number of edge intersections.

big population (like 10000 or more elements) on a single machine, cause the algorithm became incredibly slow. The Table 5.3 report the final values of the objective functions obtained by each algorithm and the number of epoch that it was able to execute within the time limit. The first thing that we can notice, as expected, is that the introduction of the 2-OPT involves a big decrease of the epochs performed. However in both genetic 2 and 3 results are better, so even if we pay an increase of execution time for each epoch, the addition of the refinement is worth. This became more evident if we consider the performance profile (Figure 5.16), where genetic 1 is far the worst algorithm.
Again, looking to the performance profile, genetic 2 seems to be the best, followed by genetic 3 that is slightly worse. However we chose genetic 3 as the winner of this first "eliminatory phase" for the following reasons: first of all the difference in the performance profile is tiny and since the dataset used is quite small we cannot say with absolute certainty that genetic 2 is the best. Moreover if we perform a 2-OPT on each new element, we move away from the idea of the genetic, in fact each children has to pass through a repair function and then a refinement, so with high probability it will maintain very few elements of its parents. Finally we execute very few epochs with respect to genetic 1 and 3 and this is against the principle of the evolution of the population. So this is why we selected the third version of the algorithm to compete with the others metaheuristics.

Instance	**Genetic 1**	**Epochs**	**Genetic 2**	**Epochs**	**Genetic 3**	**Epochs**
d657	58281	67	51628	6	51485	13
d1291	82270	9	60116	2	61857	3
fl417	13066	247	12040	3	12046	6
fl1400	29775	8	21836	2	21635	3
fl1577	37798	6	27996	1	30190	2
nrw1379	77437	7	65374	2	67865	3
p654	39289	67	34986	2	35216	5
pcb1173	83380	12	63693	2	63075	3
pcb3038	224091	1	217479	0	220982	0
pr1002	342579	19	278606	2	278935	4
pr2392	635460	1	566707	0	576564	0
rl1304	396680	9	316148	2	315956	3
rl1323	430922	8	325125	2	341412	3
rl1889	532907	3	486805	1	490274	1
u724	50800	48	44697	3	44857	7
u1060	309589	17	240649	2	241563	4
u2152	111343	1	98474	0	99573	0
u2319	316450	1	295777	0	297665	0
vm1084	326657	15	255282	2	258157	4
vm1748	502384	3	460075	1	480036	1

Table 5.3: Results of the eliminatory between the different versions of the genetic we implemented

To conclude we noticed that the genetic algorithm is well suited to a distributed approach. On the big data domain, when, for example, we need to perform a clustering algorithm, a possible solution is to split the set of points into smaller chunks, and locate them on different machines that execute the operations on them independently. We can think to a similar solution for the genetic: we can divide a big population into smaller subset, assign each of them to a different machine (or a different core if we have only one computer) and apply the algorithm. Considering the parallelism with the evolution theory, we can think to this as if we have different populations living in different habitats. At this point it is also possible to introduce "migrations" between different populations, this means that on each epoch we can randomly shuffle some elements between different subsets. So with this strategy we can run the genetic on a cluster, solving the problem of having to menage a big population and speeding up the execution of the algorithm.

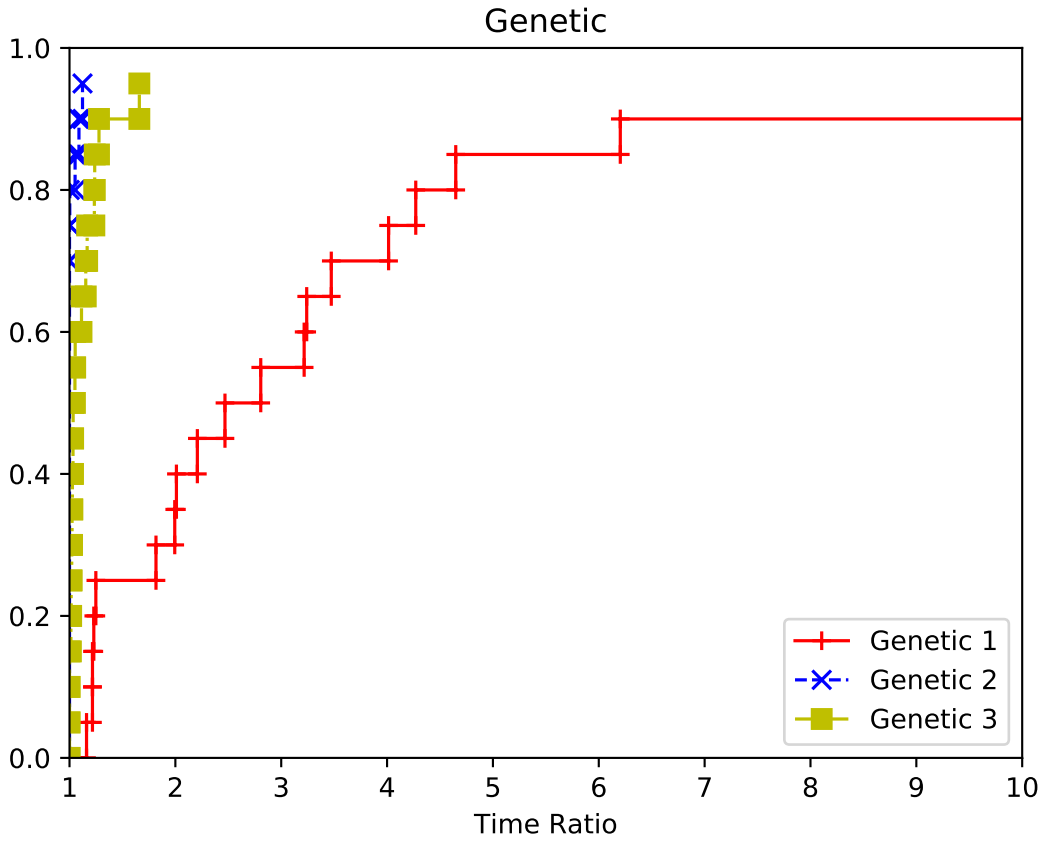


Figure 5.16: Performance profile of the three versions of the genetic algorithm we implemented

5.3.6 Final comparison between metaheuristics

On Figure 5.17 it is possible to see the performance profile of all the metaheuristics we analyzed in this section. We tested each algorithm on the usual dataset with a time limit of 30 minute per instance (all the other details about the experiments are report on section 5.4). The parameter we used are the following:

- **Multi start**: no parameter except for time limit.
- **VNS**: 5-OPT random move for the kick.
- **Tabu search**: tenure equals to 40% of the instance size.
- **Simulated annealing**: temperature list of 5000 elements, 1000 iteration with the same temperature.
- **Genetic**: population size of 2000, 200 new elements per epoch.

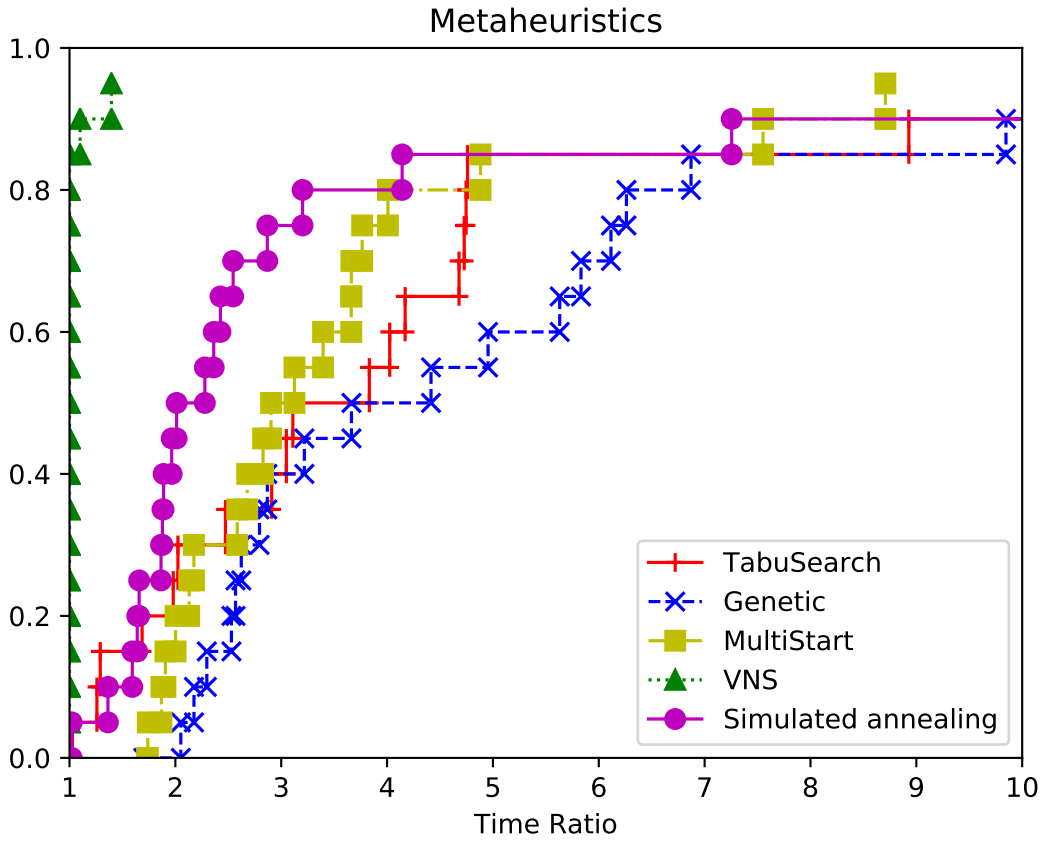


Figure 5.17: Performance profile of the metaheuristics

The first thing we can notice is that the VNS has the best performance. Of course our dataset is quite small, so we cannot say with absolute certainty that VNS is in absolute the best algorithm. However our results are aligned with the ones we saw during the course and the ones obtained by other students, so we can conclude that VNS is a very good heuristic solution for the TSP.
Another thing we can notice are the poor performance of the genetic. This result was expected due to the fact we were forced to use small populations and to introduce the 2-OPT at the expense of the number of epoch for each run.
Finally, noteworthy is the simulates annealing that seems to have very good performance.

5.4 Results Tables

We report here the tables with the results and their details we obtained on the tests of the different algorithms.

Table 5.4: **Multi-start**

Instance	**Time limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	51608	48912	5,22
d1291	30	56577	50801	10,2
fl417	30	12176	11861	2,59
fl1400	30	21291	20127	5,47
fl1577	30	23953	22204	7,3
nrw1379	30	61766	56638	8,3
p654	30	36361	34643	4,7
pcb1173	30	62375	56892	8,79
pcb3038	30	154679	137694	10,98
pr1002	30	281778	259045	8,07
pr2392	30	422804	378032	10,59
rl1304	30	276760	252948	8,60
rl1323	30	298429	270199	9,46
rl1889	30	348391	316536	9,14
u724	30	45316	41910	7,51
u1060	30	240853	224094	6,96
u2152	30	73356	64253	12,40
u2319	30	249621	234256	6,16
vm1084	30	260289	239297	8,06
vm1748	30	368113	336556	8,57

Table 5.5: **VNS**

Instance	**Time limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	49921	48912	2,02
d1291	30	52217	50801	2,71
fl417	30	11970	11861	0,91
fl1400	30	20657	20127	2,57
fl1577	30	22692	22204	2,15
nrw1379	30	59272	56638	4,44
p654	30	34832	34643	0,54
pcb1173	30	58668	56892	3,02
pcb3038	30	149738	137694	8,04
pr1002	30	264881	259045	2,2
pr2392	30	399147	378032	5,29
rl1304	30	255863	252948	1,14
rl1323	30	275537	270199	1,93
rl1889	30	324641	316536	2,5
u724	30	43054	41910	2,66
u1060	30	230070	224094	2,6
u2152	30	69208	64253	7,16
u2319	30	242850	234256	3,54
vm1084	30	244211	239297	2,01
vm1748	30	350364	336556	3,94

Table 5.6: **Tabu Search**

Instance	**Time limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	50222	48912	2,60
d1291	30	58274	50801	12,82
fl417	30	12339	11861	3,87
fl1400	30	21751	20127	7,46
fl1577	30	24728	22204	10,2
nrw1379	30	59349	56638	4,57
p654	30	37672	34643	8,04
pcb1173	30	59146	56892	3,81
pcb3038	30	146111	137694	5,76
pr1002	30	285259	259045	9,19
pr2392	30	423364	378032	10,70
rl1304	30	281595	252948	10,17
rl1323	30	293050	270199	7,8
rl1889	30	350028	316536	9,57
u724	30	45604	41910	8,1
u1060	30	243788	224094	8,08
u2152	30	74865	64253	14,17
u2319	30	249119	234256	5.97
vm1084	30	264646	239297	9,58
vm1748	30	372908	336556	9,75

Table 5.7: **Simulated Annealing**

Instance	**Time limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	50853	48912	3,82
d1291	30	55631	50801	8,68
fl417	30	11960	11861	0,83
fl1400	30	21011	20127	3,61
fl1577	30	23208	22204	4,33
nrw1379	30	60290	56638	6,06
p654	30	36663	34643	5,51
pcb1173	30	61274	56892	7,15
pcb3038	30	160064	137694	13,98
pr1002	30	270144	259045	4,11
pr2392	30	412826	378032	8,43
rl1304	30	275741	252948	8,27
rl1323	30	286098	270199	5,56
rl1889	30	353047	316536	10,34
u724	30	44111	41910	4,99
u1060	30	236150	224094	5,1
u2152	30	72906	64253	11,87
u2319	30	243034	234256	3,61
vm1084	30	252216	239297	5,12
vm1748	30	369762	336556	8,98

Table 5.8: **Genetic**

Instance	**Time limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	51590	48912	5,19
d1291	30	56405	50801	9,94
fl417	30	12030	11861	1,4
fl1400	30	21317	20127	5,58
fl1577	30	26054	22204	14,78
nrw1379	30	63077	56638	10,21
p654	30	35033	34643	1,11
pcb1173	30	62303	56892	8,69
pcb3038	30	215382	137694	36,07
pr1002	30	276044	259045	6,16
pr2392	30	546694	378032	30,85
rl1304	30	284911	252948	11,22
rl1323	30	306511	270199	11,85
rl1889	30	457862	316536	30,87
u724	30	44930	41910	6,72
u1060	30	240476	224094	6,81
u2152	30	93952	64253	31,61
u2319	30	284055	234256	17,53
vm1084	30	255868	239297	6,48
vm1748	30	432545	336556	22,19

Table 5.9: **Comparison between objective functions**

Instance	**Multi-start**	**VNS**	**Tabu search**	**Simulated annealing**	**Genetic**
d657	51608	49921	50222	50853	51590
d1291	56577	52217	58274	55631	56405
fl417	12176	11970	12339	11960	12030
fl1400	21291	20657	21751	21011	21317
fl1577	23953	22692	24728	23208	26054
nrw1379	61766	59272	59349	60290	63077
p654	36361	34832	37672	36663	35033
pcb1173	62375	58668	59146	61274	62303
pcb3038	154679	149738	146111	160064	215382
pr1002	281778	264881	285259	270144	276044
pr2392	422804	399147	423364	412826	546694
rl1304	276760	255863	281595	275741	284911
rl1323	298429	275537	293050	286098	306511
rl1889	348391	324641	350028	353047	457862
u724	45316	43054	45604	44111	44930
u1060	240853	230070	243788	236150	240476
u2152	73356	69208	74865	72906	93952
u2319	249621	242850	249119	243034	284055
vm1084	260289	244211	264646	252216	255868
vm1748	368113	350364	372908	369762	432545

Chapter 6

Conclusions

Prova prova

Bibliography

[1] E. D. Dolan and J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

Chapter 7

Appendix