Università degli Studi di Padova

Department of Information Engineering - DEI

Operation Research 2

Solutions for Travelling Salesman Problem

Miroljub Mihailovic (2021445)
Gianmarco Bortolami (1239293)

Academic year: 2020/21

Contents

List of Tables

1 Introduction Travelling Salesman Problem

1.1 History of the problem

The travelling salesman problem (also called the travelling salesman problem or TSP) asks, "Given a list of cities and the distances between each pair of cities, what is the shortest possible path that visits each city exactly once and returns to the city of origin?"[1], This is an NP-hard combinatorial optimization problem that is important in theoretical computer science and operations research.

The origins of the travelling salesman problem are not really clear. An 1832 travelling salesman's handbook mentions the problem and includes examples of travel through Germany and Switzerland, but contains no mathematical treatment. The travelling salesman problem was formulated mathematically in 1800 by Irish mathematician W.R. Hamilton and British mathematician Thomas Kirkman. Hamilton's Icosaian game was a recreational puzzle based on finding a Hamiltonian cycle. The general form of the TSP seems to have been first studied by mathematicians in the 1930s in Vienna and Harvard, particularly by Karl Menger, who defined the problem, considered the obvious brute-force algorithm, and observed the suboptimality of the nearest neighbour heuristic.

In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the United States after the RAND Corporation of Santa Monica offered prizes for steps in solving the problem. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson, and Selmer M. Johnson of the RAND Corporation, who expressed the problem as an integer linear program and developed the cutting plane method for its solution. They wrote what is considered the seminal paper on the subject, in which using these new methods they solved an instance with 49 cities optimally by constructing one round and showing that no other round could be shorter. Dantzig, Fulkerson, and Johnson, however, hypothesized that given a near-optimal solution, we — might be able to find optimality or prove optimality by adding a few extra inequalities (cuts). They used this idea to solve their initial 49-city problem using a string model.

1.2 Problem formulation

We can define the problem discussed above in a mathematical way, in order to be able to solve it with Research Operation tools[5]. In all our formulations we

will take the set of cities as $N = 1, 2, \ldots, n$ and define variables

$$\begin{aligned} x_{ij} &= 1 \qquad \text{iff } arc(i,j) \text{ is a link on the tour, } (i \neq j) \\ x_{ij} &= 0 \qquad \text{otherwise} \end{aligned}$$

and

$$c_{ij} \text{ will be taken as the length of } arc(i,j)$$

The objective function will be

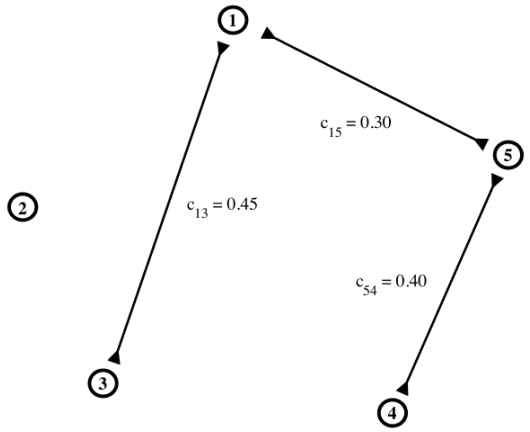


Figure 1: Lengths of some arcs of the graph

$$\min \sum_{\substack{i,j \\ i \neq j}} c_{ij} x_{ij}$$

Here we can split two cases: *directed graph* and *undirected graph*. On the first case,

$$x_{ij} = 1 \not\Longrightarrow x_{ji} = 1, \text{ clearly it's the same for the 0}$$

while in the second case,

$$x_{ij} = 1 \iff x_{ji} = 1, \text{ clearly it's the same for the 0.}$$

On the directed graph environment, the basic constraints added to the model are

$$\sum_{\substack{j \\ j \neq i}} x_{ij} = 1 \qquad \forall i \in N$$

so, only one out-edge from node i and

$$\sum_{\substack{i \\ i \neq j}} x_{ij} = 1 \qquad \forall j \in N$$

so, only one in-edge to node i.

While, in the case of undirected graph, we actually merge together the two above equations into only one

$$\sum_{\substack{j \\ j \neq i}} x_{ij} = 2 \qquad \forall i \in N.$$

From now on, these two definitions will be our basic formulations where we

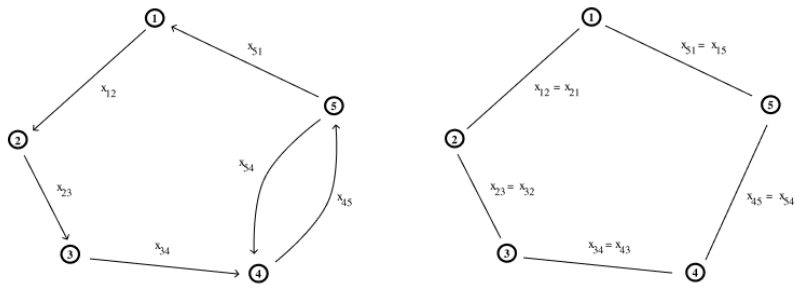


Figure 2: Directed and undirected graphs

will always add new variables and new constraints, in order to find the optimal solution as quickly as possible.

2 Software design

In the following chapter, we will describe the software design we have implemented to find a feasible solution for the TSP. In particular, we report the libraries, the data, the organization of the code that is developed in C.

2.1 Libraries

2.1.1 CPLEX

CPLEX or IBM ILOG CPLEX Optimization Studio is an optimization software that solves integer programming problems, large linear programming problems using either primal or dual variants of the simplex method or the barrier interior point method, convex and non-convex quadratic programming problems, and convex quadratically constrained problems. We use the latest version realized: 20.1 in such a way to solve the compact models that we are going to describe in the following chapters.

2.1.2 Gnuplot

Gnuplot is a command-line and GUI program that can generate two- and three-dimensional plots of functions or data. The program runs on all major computers and operating systems (GNU/Linux, Unix, Microsoft Windows, macOS, FreeDOS, and many others). It is a program with a fairly long history, dating back to 1986. Despite its name, this software is not part of the GNU Project. This library is implemented for plotting results of the TSP implementation.

2.2 Dataset generation

A Dataset generation for TSP problem is a process that requires a number of nodes and the name desired in such a way to create new problems that will be used for testing phase. In this way, we are able to test the different methods and to choose the size of the problem in order to have a good computation between the techniques. It is based on the following steps:

1. Concatenation of the $Name$, $Type$, $Dimension$ (it is given in input) and $Edge\_Weight\_Type$.
2. The *node_coordinations* is selected randomly, and it is verified if it is select by others vertices.
3. Save the values found into a file.

2.3 Test phase

Software testing is an investigation conducted to provide stakeholders with information about the quality of the software product or service under test. In our case, we have compared the different TSP elements within the dataset created thanks to TSP Generator. We use as comparison tool the object cost for the heuristic algorithms and time necessary for solving the problem in case of exact solution, since they achieve the optimal solution. We have compared the following implementations:

- Optimal solutions algorithms: Compact models
- Optimal solutions algorithms: Dantzig, Fulkerson and Johnson based techniques
- Refinement Heuristics
- Constructive Heuristics
- Metaheuristics

2.4 Software management

In the following sub selection, we want to introduce a brief explanation of how the C files are divided and managed.

2.4.1 TSP

This section is composed by all the implementation developed, starting from the construction of the model and continuing through heuristic algorithm, we have a lot of function that are used to solve the problem required and also the additional implementation that are useful for finding a solution to the TSP problem. In end, we have also a section for plotting result and images of the problem solved and the initialization of the variables of the *instance* struct, this last mentioned is important since contains all the information of the problem such as the number of nodes and the coordinates of the vertices.

2.4.2 Chronometer

This C file is a simple implementation created in such a way to compute the time that will be used for analysing the performance of the different algorithm and TSP problems.

2.4.3 TSP generator

A TSP generator is the implementation of the idea describe in the previous selection where the goal consist of creating a TSP problem for testing phase.

2.4.4 Utils

The last file is important since it is used for additional functions that we will define using the following list:

- Reading the input, here the TSP file is decomposed in order to save inside the variables all the values.
- Parsing the command line, in this function we read and save the information from command line, the most important information are the TSP file, random seed, that is used with the CPLEX algorithm, and the time limit fundamental for heuristic implementation.
- Testing, this last section is used for testing a set of files with the respective method and then to save the information that will be analysed.

3 Optimal solutions algorithms

3.1 Compact models

Taking in account the formulation of the problem gave on Section 1.2 we can add specific constraints in order to improve the efficiency of the solution for the Travelling Salesman Problem.

3.1.1 Miller, Tucker and Zemlin models

Miller, Tucker and Zemlin proposed in 1960 a model also known as Sequential Formulation. The idea is that to each node is associated a specific value that represent the order in which these nodes are visited one by one during the tour. So, once a node is visited, it's marked with an incremental value that keeps tracking of the tour made. Each node visited can't have an edge linked to another node that was visited previously, i.e. which has a marked value smaller than the previous one. In this way, they actually impose a constraint that doesn't allow the making of subtour between the nodes.

The constraint described above requires the use of a simple trick in order to enable and disable this constraint based on the linking condition. In other words,

the constraint should have effect only if it's applied to two nodes connected, while when the two nodes aren't linked with an edge the constraint should be satisfied by whatever value of the variables. The technique used in this setting is called Big-M Trick. Intuitively, if we want that $a > b$ (with $a > 0$) only under some value of a binary variable c (activation variable, i.e. if $c = 1$ means that the constraint is active, otherwise if $c = 0$ means that the constraint is inactive), we can rewrite the disequality in this way: $a > b - M(1 - c)$, where M is a "big" number such that can inhibit the constraint imposed by b. Relaxing the disequality, we can see it like,

- If c is 1, then the disequality becomes $a > b$;
- If c is 0, then the disequality becomes $a > -M$, accounting that $M \gg b$ so $b - M \simeq -M$. Observe that the disequality is satisfied for each "valid" value of a (i.e. $a > 0$), so it's trivially "disable".

Smaller the Big-M variable is, until it works, and better it's the formulation of the problem, because we will actually lie on the convex hall.

Formally, the model adds to the basic formulation of the problem the $n - 1$ variables

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

in other words, one for each node of the problem to solve (except for the node 1), and $n^2 - n + 1$ constraints

$$u_i - u_j + nx_{ij} \leq n - 1 \qquad \forall i, j \in N - \{1\}, i \neq j.$$

The variable n works as Big-M variable.

In contrast to the cuts that IBM ILOG CPLEX may automatically add while solving a problem. Lazy constraints are constraints that we know are unlikely to be violated, and in consequence, we want them applied lazily, that is, only as necessary or not before needed. In other words, they are constraints added to the list of constraints that should be added to the LP subproblem of a MIP optimization if they are violated. Lazy constraints are constraints not specified in the constraint matrix of the MIP problem, but that must be not be violated in a solution. For this reason, in order to improve the performance to compute the optimal solution, the lasts constraints were added as lazy constraints. So, they will be included in the model only when they are violated.

Once an edge between node a and node b is chosen and set as active, we naturally understand that the reverse edge, i.e. the link between node b and node a, must be inactive, otherwise we surely meet a subtour. These constraints

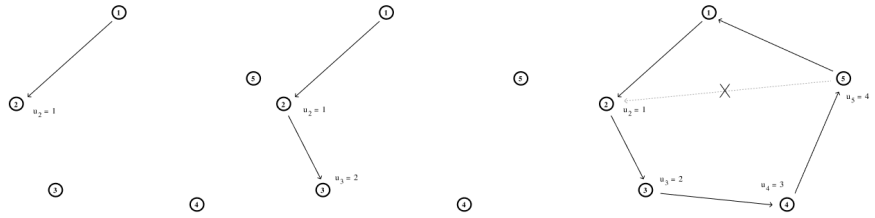


Figure 3: Miller, Tucker and Zemlin solution

actually don't exist, so the search of the solution can consider and try also this edge as possibly belonging to the final optimal solution. To help the model to take in account only reasonable edges, we can add to it the so-called Subtour Elimination Constraint (SEC). This kind of constraint simply impose the impossibility to be active on the same solution, the edges between a and b and vice versa. However, the SECs will rarely be violated, so it's a good idea to set them as lazy constraints like the previous ones.

Let defines the constraints in a more formal way,

$$x_{i,j} + x_{j,i} \leq 1 \qquad \forall i, j \in N - 1, i \neq j$$

so there are actually $n(n-1)$ constraints like this, one for each pair of different nodes.

3.1.2 Gavish and Graves model

Gavish and Graves published in 1978 a flow based model with a single commodity flow (F1). The philosophy under this approach is that if we start from node 1 with n commodity (n is the number of nodes of the graph taken), if we drop one commodity on each node we actually finish the tour when we haven't others commodities yet. So, measuring the flow between each step, we can see that it decrease step by step, until it reaches 0. Clearly, we have to add some other constraints to make this formulation consistent. Each node must have an in-flow equal to the out-flow unless the commodity dropped there and at the starting node the number of commodity must be equal to the number of nodes

of the problem accounted. Let's start to formalize all these continuous variables

$$y_{ij} = \text{ Flow in an } arc(i,j), i \neq j$$

and constraints,

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

that works like an upper bound with a variable value based on activation or deactivation of this arc. If the arc is set to 1, then the flow must be less than the maximum flow, while if the arc is set to 0, then the flow must be 0.

$$\sum_{\substack{j \\ j \neq 1}} y_{1j} = n-1,$$

$$\sum_{\substack{i \\ i \neq j}} y_{ij} - \sum_{\substack{k \\ i \neq k}} y_{jk} = 1, \qquad \forall j \in N - \{1\}$$

which mean that the first node start with all commodity, but step by step the flow decrease by one.

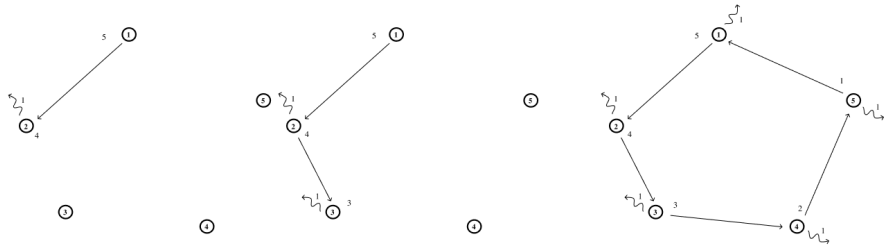


Figure 4: Gavish and Graves solution

3.2 Dantzig, Fulkerson and Johnson based techniques

3.2.1 Benders method

Until now, we always add new constraints to the model during the building phase, even if we used the lazy constraints these are included necessarily during the first stage of the model definition. What if we take lazy behaviour

to the extreme, introducing the constraints at the fly? So, they will be really added to the model only when they are needed (i.e. violated by the solution computed). Jacobus Franciscus (Jacques) Benders formalizes this idea in a well-defined method, also called *Loop Method*. At the first stage, the model is a simple model made by the variables and the constraints talked about in Section 1.2, i.e. each node can have only one in-edge and only one out-edge. These restrictions still allow the building of subtours on the graph, so the solution of this naive model with high probability could have several subtours.

At the next stages, we add step by step all subtour elimination constraints in order to compute a new solution without the subtours there exist on the previous one. After some step, we definitively add all SECs needed by the model to avoid any kind of subtour, and we can hope that some useless constraints aren't added. On the best case, we are including to the model only the minimum useful constraints to find the optimal solution. In Figure 5 we can see a step by step computation for the solution of a graph of 48 nodes. At each step, the algorithm tries to reduce the number of subtours made by the previous model adding new SECs, until it achieves the optimal solution without subtours at the end of the computation.

Algorithm 1: Benders method

Data: *model*
Result: x^*, optimal solution for TSP
1 $x \leftarrow$ find the optimal solution of the current model
2 **while** $has\_subtours(x) > 1$ **do**
3 | $addSEC() \leftarrow$ add each component to the model
4 | $x \leftarrow$ find the optimal solution of the model with new SECs
5 **end**
6 **return** x

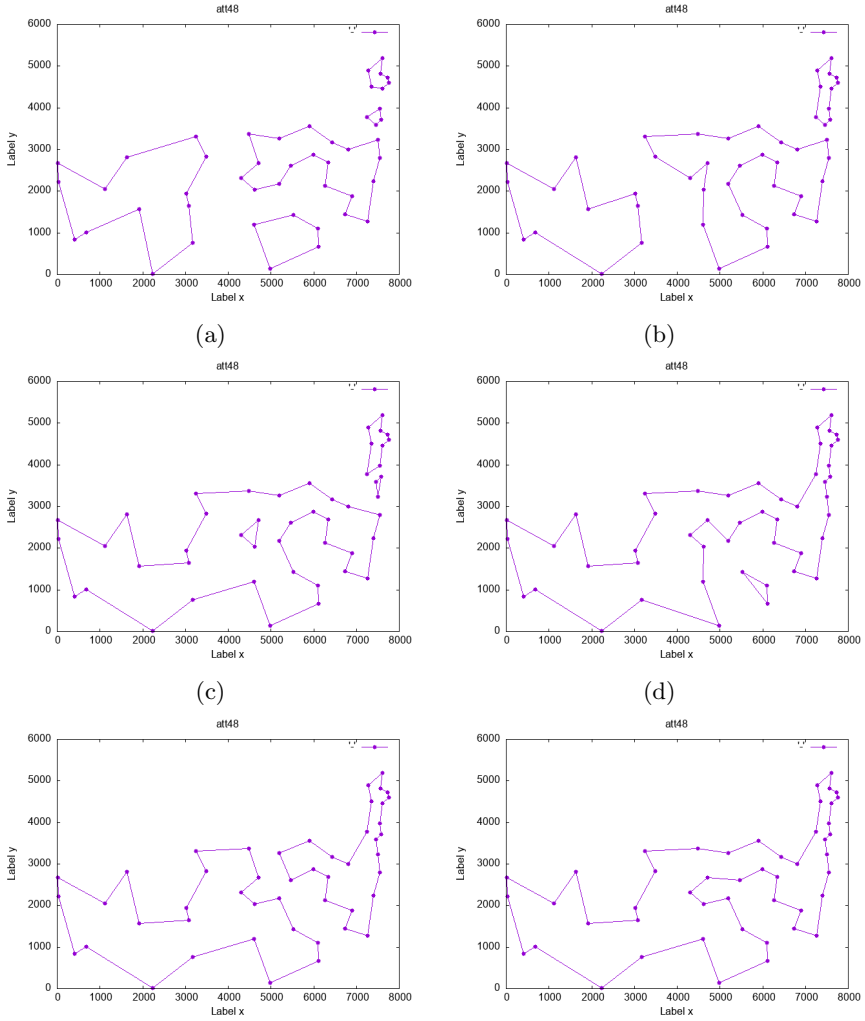


Figure 5: Benders method step by step.

3.2.2 CPLEX Callback

From previous chapter we have seen how to solve the TSP on relaxed instances, adding the sub-tour elimination constraints. There is different implementation that can be applied, the concept is based on Branch and cut algorithm in which we solve the problem without constraint and from this situation we find different integer solution, the idea is to add the sub-tour elimination constraints when the integer solution, obtained from the previous point, is found with a

sub-tours. This implementation permits us to have a fast computation, since we have to consider only one tree instead of multiple trees implementation. Therefore, the algorithm implemented receive in input the model initialized and with the objective function and the integer solution, the method is called until the component variable is equal to one, otherwise if it is not true we add SEC to the model, at the end the implementation returns the optimal solution for the TSP.

Algorithm 2: Callback

Data: *model*, x
Result: x^*, optimal solution for TSP
1 *component* $\leftarrow$ number of components of x
2 **if** *component* ≥ 2 **then**
3 | $addSEC() \leftarrow$ *add each component to the model*
4 **end**
5 **return**

3.2.3 Concorde

The Concorde implementation is one of the best methods developed to solve the TSP problem, this technique is open source, so the library is available at the following link http://www.math.uwaterloo.ca/tsp/concorde.html. Before explaining the behaviour, it is important to note that the following files (after some modification) are only needed for this part: *allocrus.c*, *cut.h*, *cut_st.c*, *macrorus.h*,*mincut.c*,*Shrink.c*,*sortrus.c*,*util.h*. The algorithm implemented is very similar to the previous case, but we have also to define the UsrCutCallback and to implement the function *CCcut_connect_components* and *CCcut_violated_cuts*. The first mentioned function is used for searching connected components in such a way to determine if a solution is connected or not, it required the following parameters:

- ncount, number of nodes n.
- ecount, $n * (n-1)/2$.
- elist, vector that specified the vertices.
- x, the solution in which the components are computed.
- ncomp, number of connected components.

- compscount, vector of vectors that include the number of nodes for each connected component.
- comps, vector of vectors that include the indices of the nodes inside the components.

The second method, *CCcut_violated_cuts*, It used as a search function for sections with a capacity below a certain threshold. It requires:

- ncount, ecount, elist, defined previously.
- dlen, solution vector.
- cutoff, Threshold value to determine if a cut is breached or not, this value is set to $2 - EPSILON$ where EPSILON is a small value, usually equal to 0.1.
- doit_fn, function that is executed when a cut is found.
- parms, struct developed by the user.

The following algorithm reports the idea adopted for solving the TSP using the Concorde library.

Algorithm 3: Concorde

Data: *model*, x
Result: x^*, optimal solution for TSP

```
component ← number of components of x
if component ≥ 2 then
    addSEC() ← add each component to the model
end
if component = 1 then
    addSEC() ← add SEC on each fractionary solution
end
return
```

4 Heuristics

4.1 Refinement Heuristics (Matheuristics)

The Matheuristics are model-based meta-heurisitcs in which it exploits the existing of mathematical programming model, we do not have the certain the

solution, it means that it could be optimal or not. This kind of implementation are used because they permit to have a faster computation, but usually they perform worse than the MIP solver, and it is important to notice that the optimal solution is not guaranteed. In the following selection, we are going to present the two approach that we have seen, the matheuristics combine the mathematical programming or also called MP and the heuristic implementation. The two techniques implemented are: Hard fixing and soft fixing (local).

4.1.1 Hard Fixing

Hard fixing or variable fixing is a matherusitic implementation in which performs local search around a given integer feasible solution, in our case it is any tour obtained by an approximation algorithm, we use the root solution from callback algorithm described in the previous chapter. The idea of this procedure consists of fixing some variables in the MP and then to solve the new version of the problem, it follows that the solver is able to manage the formulation in easier way. The variables belong to the set of edges E, we choose one at random, then we fix it by putting its value equal to one. Now, it is important to highlight that each element inside E is associate using a probability, then we pick all the variables that are greater than a value that is imposed previously. This value changes based on time, for example it can initially be 0.9 and decreases after a few seconds to 0.8, everything has the aim of establishing more variables over time in order to find a solution. The procedure is described in the following, in which we have an initial solution found using callback method, and we also initialize the current time. Accordingly, the algorithm repeat the following procedure until a time limit, that is passed in input, is reached. The steps required are: random fixing that was introduced previously that requites the initial model and a probability rate, the last mentioned could be seen as a vector of values from (e.g. $[0.9, 0.8, 0.7, 0.6, 0.5]$). Therefore, we try to solve the solution and comparing the cost of the current solution with the best solution found until now. When a new outcome is achieved, this becomes the best result, and we reset the probability rate to initial value. At the end of the while loop, each variable is settled to default bound.

Algorithm 4: Hard fixing

```
Data: model, probability_rate, time_limit
Result: final solution for TSP
x_inc ← feasible solution found using callback
current_time = 0;
while time_limit > current_time do
    randomFixing(model, probability_rate)
    x ←solution for the model with fixing
    if cost(x) < cost(x_inc) then
        x_inc = x
        probability_rate ← re-inizialize to start value
    end
    defaultRandomFixing() ← default bounds for each variable
end
return
```

4.1.2 Soft Fixing

The second matherisitc algorithm that was implemented is soft fixing or local branching, which is based on local search starting from a given integer feasible solution as in case of Hard fixing. The idea of local branching consists on adding constraint (local branching constraint) on the model, this application permits to fix some variables. Differently from the previous case, we do not fix in advance the variables, but the implementation forces the model to attach some elements of the reference solution. The constraint mentioned is defined in the following way:

$$\sum_{e:x_e^{ref}=1} (1 - x_e) + \sum_{e:x_e^{ref}=0} x_e \leq k$$

Where, given the incumbent (x_{ref}), first sum represent the number of variables flipping their value from one to zero, i.e. their value was equal to one, and they are not part of the current solution. The second part symbolizes the number of variables flipping their value from zero to 1, i.e. the complementary situation of the previous case. The right side is composed by the k value that represent how many variations the new solution allows, this variable can change according to the time if there are not an effective improvement of the new outcomes in order to enlarge the set of variations. We can rewrite the already cited equation with the subsequent formulation:

$$\sum_{e:x_e^{ref}=1} x_e \geq \sum_{e:x_e^{ref}=1} 1 - k = n - k$$

In this case, we add also the variable n that define the cardinality of the set E (edges). The algorithm illustrated below is very similar to hard fixing case, in which instead of probability values, we have to select a vector of k values ($[2, 3, 5, 7, 10]$) that change according to time. Therefore, the implementation stay in loop until the time limit is reached and for each iteration a local branching constraint is added to the model according to the k value selected then, after comparing the cost of the best solution found and the current solution, it is important to remove the constrained added previously otherwise all the variables will be blocked after some iterations.

Algorithm 5: Soft fixing

```
Data: model, k_values, time_limit
Result: final solution for TSP
x_inc ← feasible solution found using callback
current_time = 0;
while time_limit > current_time do
    localBranchingConstraint(model, k_values)
    x ←solution for the model with LBC
    if cost(x) < cost(x_inc) then
        x_inc = x
        k_values ← re-inizialize to start value
    end
    defaultLocalBranchingConstraint() ← Remove the LBC
end
return
```

4.2 Constructive Heuristics

A constructive heuristic is a type of heuristic method that instead of starting with an MP solution, we create it from the beginning. In our case, we saw the Nearest Neighbour (Greedy), GRASP (Randomized Greedy) and 2-OPT implementation.

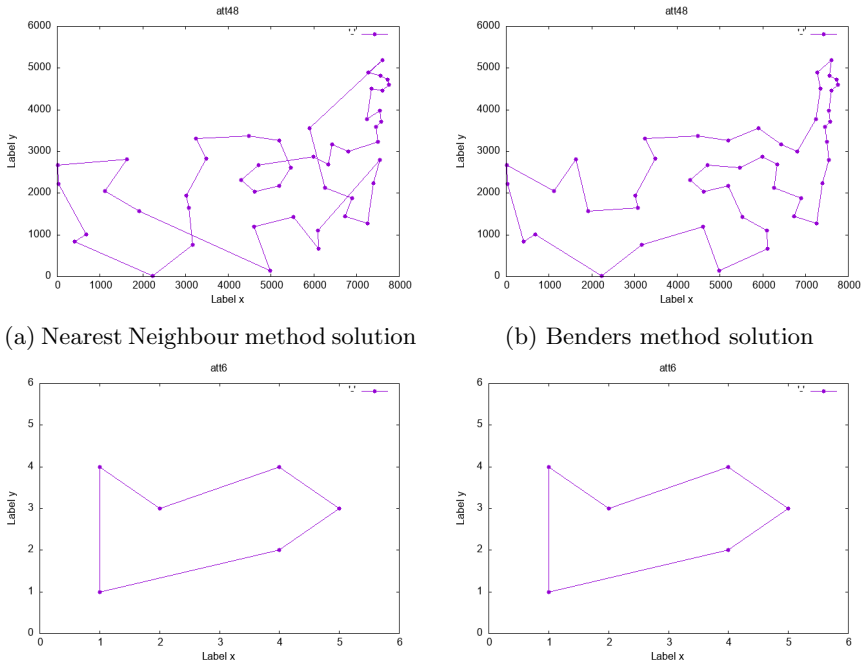
4.2.1 Nearest neighbor (Greedy)

A first greedy and really naive approach to solve the Travelling Salesman Problem is the Nearest Neighbour method. Here we actually don't use any CPLEX

tools, because each incumbent is computed using only a distance function and linking the nearest nodes. The idea under this method is that, with high probability and except some, but not so rarely, cases, a certain node will be connected to the nearest one in order to achieve the minimum path. This approach it's so powerful with simple graphs (i.e. such graphs without too much sparse nodes), but when the graph start to become more and more complex and with too much near node, this naive idea could be not so smart. As we can see on the Figure 6 the Nearest Neighbour approach perform well on simple problem, and it actually doesn't require any CPLEX licence, while on more complex problem it gets stuck with high probability in local minima. One way to try to achieve better results it's to start the method with several starting points (one each time), ideally all nodes available. However, the solutions will be still not so satisfactory.

Algorithm 6: Nearest Neighbour method (Greedy)

```
   Data: nodes
   Result: x, solution for TSP
 1 best_objective_function ← ∞
 2 foreach node in the graph do
 3 |   objective_function ← 0
 4 |   current_node ← node
 5 |   while all nodes aren't connected do
 6 |   |   nearest_node ← nearest_node(current_node)
 7 |   |   connect the current_node with the nearest_node
 8 |   |   objective_function ← objective_function + dist(nearest_node,
   |   |     current_node)
 9 |   end
10 |   if objective_function < best_objective_function then
11 |   |   best_objective_function ← objective_function
12 |   end
13 end
14 return x
```



(a) Nearest Neighbour method solution

(b) Benders method solution

(c) Nearest Neighbour method solution

(d) Benders method solution

Figure 6: Complex vs. easy graphs with Nearest Neighbors and Benders methods

4.2.2 GRASP (Randomized Greedy)

Greedy randomized adaptive search procedure or GRASP, this algorithm is a randomized version of Greedy that was presented before but instead of starting from the first node we select a random one and chose to visit the node that is closest to the current position, this procedure is not mandatory since we have implemented a random perturbation according to a given probability that could change the next node to second or third (or higher order) closest to actual. The expiation made could be seen in the following way:

1. Initialize all nodes.
2. Set them as not visited.
3. Choose one random node and set it as visited and as current node.
4. Select the next node from a list of the closest vertices according to a given probability and set it as visited.

5. Set the node selected from the previous step as current.

6. Repeat the procedure until all nodes are visited.

The biggest problem of this implementation is the inefficiency due to the costs and the non-achievement of the optimal solution, how it could be seen in the figure there are some crossing arcs which can be removed using $2 - OPT$ procedure.

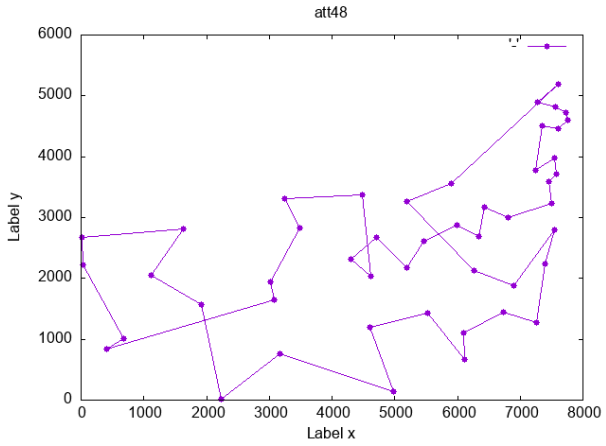


Figure 7: GRASP solution for *att*48.*tsp* problem

The algorithm requires:

- Number of nodes(*number_nodes*).
- *probability*, this variable permits to apply some perturbations, basically if it is greater than a random value (between 0 and 1) the procedure does not use the closest distances, we have empirically seen that the best value is 0.15.
- *time_limit* represent the amount of time used for running the algorithm.
- *number_distances* is the cardinality of the set of the closest distances, we fix it equal to 5 since we obtain good results with this value.

The next steps are done until the *time_limit* is reached, first it is important to initialize the starting random node, the list of visited node and the current solution x. After adding the first node, it follows a loop where the function $Next()$ returns the next node, for the reason explained before, it is chosen in respect to *probability* and for obvious reason the next vertex must be not visited.

At the end, we compare the cost with the best solution and after reaching the *time_limit* it returns the best solution found.

Algorithm 7: GRASP

Data: *number_nodes*, *probability*, *time_limit*, *number_distances*
Result: final solution for TSP

```
1  x_best ← initialization
2  current_time = 0
3  while time_limit > current_time do
4  |  current_node ← initialization random starting point
5  |  visited ← initialization
6  |  visited.Add(current_node)
7  |  x.Add(current_node)
8  |  while visited.numberNodes() < number_nodes do
9  |  |  current_node←Next(visited,current_node,probability,number_distances)
10 |  |  visited.Add(current_node)
11 |  |  x.Add(current_node)
12 |  end
13 |  if cost(x) < cost(x_best) then
14 |  |  x_best = x
15 |  end
16 end
17 return
```

4.2.3 2-OPT

The 2-Opt algorithm is used to solve the crossing problem presented with the Greedy and Grasp heuristic solution, below is reported an example where given the first arc (between the node j and $j+1$) and the second arc (between the node $i-1$ and i) that are in a clearly crossing situation, the implementation switch them to arc $i-1$ - j and i - $j+1$. The way that is used for finding this state is given by the following formula:

$$\Delta(j, j+1) = [cost(j, j+1) + cost(i-1, i)] - [cost(j, i+1) + cost(i-1, j)] < 0$$

If the condition is true, i.e. the deltas of the candidate edges are less than zero, we can swap them using the procedure described above. There are two ways to choose the possible delta:

- Loop all possible arcs and whenever the algorithm finds a delta lower than zero then we can switch them.
- Loop all possible arcs and switch the minimum delta found.

We have implemented the second option since it performs better compared to the first.

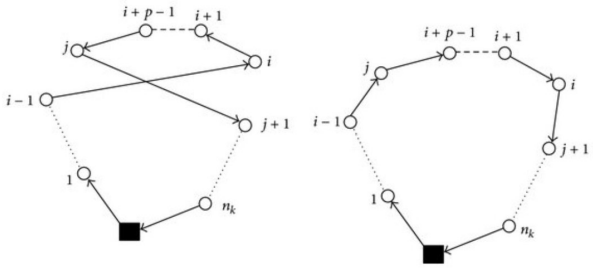


Figure 8: Example of crossing arcs

Now, we are going to introduce the steps required for 2-OPT algorithm, starting from an initial solution found implementing constructive heuristic, we are opted for Greedy method. Consequently, as we did with the previous algorithms, we set a *time_limit* and proceed with the initialization of, Δ_{min} in which we save the minimum delta with the respective two candidate nodes for switching. For every pair of nodes within the set vertex V, we compute the Δ using the formulation cited before, and we also check if it is the minimum value.

At the end of the $foreach$ the algorithm swaps the edges found previous using the function $swapEdges()$ and then verify if the cost of the current solution is lower compared to the cost of the best solution. In the end, we can conclude highlighting that the solution returned is without any crossing pair, but the optimality is not guaranteed for this reason other methods are applied in order to escape from local minima.

Algorithm 8: 2-OPT

```
   Data: V, time_limit
   Result: final solution for TSP
 1 x_best ←initial solution found using Greedy
 2 current_time = 0
 3 while time_limit > current_time do
 4 |   Δ_min = 0
 5 |   v_jmin = 0
 6 |   v_imin = 0
 7 |   foreach (v_j, v_i)(V × V) (j, i) : j, i ∈ V, j ≥ i do
 8 |   |   x = x_best
 9 |   |   Δ_+ = cost(v_j, v_{j+1}) + cost(v_{i−1}, i)
10 |   |   Δ_− = cost(v_j, v_{i+1}) + cost(v_{i−1}, j)
11 |   |   Δ = Δ_+ − Δ_−
12 |   |   if Δ < 0 & Δ < Δ_min then
13 |   |   |   Δ_min = Δ
14 |   |   |   v_jmin = v_j
15 |   |   |   v_imin = v_i
16 |   |   end
17 |   end
18 |   if Δ_min < 0 then
19 |   |   x ← swapEdges(v_imin, v_jmin, x)
20 |   end
21 |   if cost(x) < cost(x_best) then
22 |   |   x_best = x
23 |   end
24 end
25 return
```

4.3 Metaheuristics

Metaheurisitic algorithms are developed for finding, generating and selecting a heuristic in such a way to obtain a good solution for the optimization problem, in our case for TSP. This kind of implementation do not guarantee the globally optimal solution, and there are also some algorithms in which they implement a set of random variables. In combinatorial optimization, searching on a huge set of feasible solutions, metaheuristic implementations are able to provide a good solution in a restricted amount of time and this is the reason why they are used for optimization problems. Now we are going to present metaheuristic algorithms that we have tested: Variable Neighbourhood Search (VNS), Tabu Search and Genetic Algorithm.

4.3.1 Variable Neighbourhood Search (VNS)

Variable Neighbourhood Search or VNS is a technique in which we start from a local optimal solution, we have opted for 2-OPT heuristic, then a perturbation phase is necessary for escaping from the corresponding valley. This last implementation can be done searching between the neighbourhood of the current solution and if there is not any improvement of the objective function it follows that the radius of the neighbourhood becomes greater, therefore it is possible to get out from local optimal. The procedure that we have adopted for applying VNS is the following: firstly, find an initial local optimal solution using the $Greedy$ heuristic and then removing the crossing arcs with $2-OPT$, after that, it follows an initialization of the k value, this variable represents the number of random edges that are going to be switched randomly in such a way to make some perturbations and trying to find a better solution between the neighbourhood. As we have done for others heuristic implication, we set a $time\_limit$ and try to improve the x_{best} solution using a loop. Starting from the best solution found until now, we select k random edges using the function $randomEdges$ and then applying $swapEdges$ we are able to perform some perturbations on the current solution. Solving the problem by executing again the $2-Opt$ algorithm, we can compare the costs and if there is an improvement the best solution is updated and the k value is reinitialized otherwise it follows an enlarging region of neighbourhood by incrementing the number of edges that will be switched. It is important to highlight that the maximum value of k is 4 since if we enlarge we can jump far from the current region.

Algorithm 9: VNS

Data: *time_limit*
Result: final solution for TSP

$x_{best} \leftarrow$ initial solution found with 2-OPT
$k = 2$
$current\_time = 0$
while $time\_limit > current\_time$ **do**
    $x = x_{best}$
    $edges = randomEdges(x, k)$
    $swapEdges(x, k, edges)$
    $x = 2Opt(x)$
    if $cost(x) < cost(x_{best})$ **then**
        $x_{best} = x$
        $k = 2$
    end
    else
        $k = k + 1$ (max size of k is 4)
    end
end
return

I want also to remember that are six possible ways for switching the edges when $k = 3$ as it is reported in the picture below, in this case we opted for implementing the corresponding image "e".

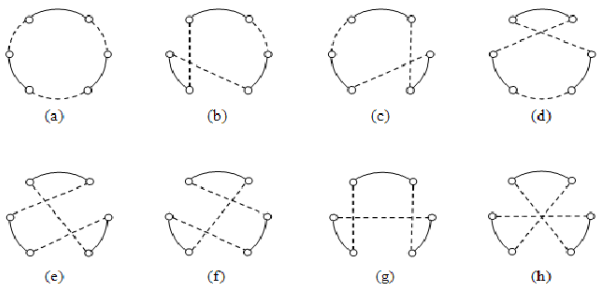


Figure 9: All possible combination of $k = 3$

4.3.2 Tabu Search

Tabu search is a metaheursitic algorithm in which the word "tabu" means a sequence of action that have to be avoided. The basic idea of this implementation consists of exploring the solution space using a tabu lists that permit us to apply some not allowed action in such a way to escape from local optimal solution. One of the main components of Tabu Search is its use of adaptive memory, which creates a more flexible search behaviour, these strategies are therefore the main concept of tabu search approaches. Now, we are going to introduce the steps required for developing the algorithm cited previously:

1. The technique described need as in the previous cases a *time_limit* for the execution and two other variables: *long_length* and *short_length* that represent the two different size that will be implemented during the dynamic changing of the tabu list length and the *time_changing_length* value that could be seen as slot time in which one of the two length mentioned is settled.

2. As we have done for VNS implementation, we initialize x_{best} using the solution provided by 2-OPT.

3. *tabu_list* is a set of nodes where the elements are considered "tabu", in other words, the two arcs associated with vertex within *tabu_list* are blocked i.e. they remain fixed when the algorithm will try to solve the problem.

4. Now, until the time limit runs out, the algorithm checks, using the values described before, if it is the time to change the length of the *tabu_list*, there is some version in which the size is fixed, but we opted for this implementation because large value give as a "diversification" approach and small value an "intensification" mode.

5. Selecting two random edges and verifying if they are just present into the *tabu_list* as it could be seen on line [7] and [8] of the code reported below, then it is possible to add the first vertex on tabu list, if the list is full, the last element will be replaced like in a queue.

6. Now we can execute the 2OptTabu algorithm, this is just an evolution of the 2-Opt that is described before, in which we avoid the computation of nodes present inside the *tabu_list*.

7. Comparing the cost of the current solution found with x_{best} and if it lowers, it derives that we have found a new best solution, and the tabu list is rested to initial value.

Algorithm 10: Tabu search

Data: $time\_limit, long\_length, short\_length, time\_changing\_length$
Result: final solution for TSP

$x_{best} \leftarrow$ initial solution found with 2-OPT
$tabu\_list \leftarrow$ initialization
$x = x_{best}$
$current\_time = 0$
while $time\_limit > current\_time$ **do**
    $tabu\_list.changeDynamicLength(long\_length, short\_length, time\_changing\_length)$
    $edges = randomEdges(x, k = 2)$
    $checkEdges(x, edges, tabu\_list)$
    $swapEdges(x, edges, tabu\_list)$
    $x^{'} = 2OptTabu(x, tabu\_list)$
    if $cost(x^{'}) < cost(x_{best})$ **then**
        $x_{best} = x^{'}$
        $x = x^{'}$
        $tabu\_list \leftarrow$ reset tabu list
    end
end
return

The concept of 2-Opt for Tabu search algorithm is similar with the one described above in which we check if the two nodes, v_j and v_i, are preset inside the $tabu\_list$. The next step consists of finding the minimum Δ and the nodes associated with this variable, as we have done previously. It follows a step for swapping edges in such a way to remove the crossing pairs, these procedures are repeated until the maximum number of improvement is reached.

Algorithm 11: 2OptTabu

Data: V, *time_limit*, *tabu_list*, x_{best}
Result: x without crossing pairs

$\Delta_{min} = 0$
$v_{jmin} = 0$
$v_{imin} = 0$
$x = x_{best}$
while *the cost*(x) *remain the same for 3 iter* **do**
  foreach $(v_j, v_i)(V \times V)\ (j,i) : j, i \in V, j \geq i$ **do**
    if $checkNodes(v_j, v_i, tabu\_list)$ **then**
      continue *Skip these nodes*
    end
    $\Delta_+ = cost(v_j, v_{j+1}) + cost(v_{i-1}, i)$
    $\Delta_- = cost(v_j, v_{i+1}) + cost(v_{i-1}, j)$
    $\Delta = \Delta_+ - \Delta_-$
    if $\Delta < 0\ \ \&\ \ \Delta < \Delta_{min}$ **then**
      $\Delta_{min} = \Delta$
      $v_{jmin} = v_j$
      $v_{imin} = v_i$
    end
  end
  if $\Delta_{min} < 0$ **then**
    $x \leftarrow swapEdges(v_{imin}, v_{jmin}, x)$
  end
end
return

4.3.3 Genetic Algorithm

The genetic algorithm (GA) is a model or abstraction of biological evolution based on Charles Darwin's theory of natural selection, it's a meta-heuristic and belongs to the larger class of evolutionary algorithms (EA)[10]. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection. In nature, only the strong one survive, the process of eliminating the weak is called natural selection. Genetic algorithm use that same principle to eliminate the "weak" solutions and finally produce the best solution. But how the genetic algorithm works? After a random generation of the primitives, i.e. the individual of the first generation, we can divide its flow into 3 main steps that will be repeated several times:

- **Cross-over phase.** The individual that compose the population are split in couples and with a certain probability they give an offspring that will be a merge of the two parents. In this case, the offspring is generated by taken one half of edges from the first parent and the other half of the edges from the second parent. Clearly, it's really likely to obtain, after this computation, a non-solution for the problem, i.e. a tour with isolated nodes or subtours inside. So it will be rearranged, connecting randomly the nodes isolated.

- **Selection phase.** After the birth of offspring the population will be, at maximum, doubled. In order to maintain a population of fixed size for computation and performance aims, we have to delete some individuals from the population. The simpler and more intuitive choice is to eliminate the worse individuals until we reach again the population size desired, however sometimes we can see that maintain some bad samples can get a better final solution. The reason behind this, at first look, not smart idea is that a solution with not so good performance level could hide inside it some excellent achievements. Roughly speaking, two perfect solution don't guarantee that their offspring will be good as well. Sometimes can happen that their offspring will be worse with respect to an offspring generated from two not so good solutions.

- **Mutation phase.** With the previous two phases, we actually change a lot of the population, but each generation strictly depends on the lasts. This kind of behaviour can lead the procedure to a situation of equilibrium due to the high dependencies between previous and next generations.

On the solution side, this equilibrium appears as a reaching of a local optimum. When we want "shake" the individuals we could apply some mutations that change a bit the solution, getting with high probability a worse solution, but on the best case we actually escape from the actual local minimum valley, and we could reach a new minimum's valley to walk through. Ideally, this new minimum could be the global minimum. The mutations, in this case, consist of change of some edges of the solution, also the change of only one edge can lead to a good "shake" behaviour.

The number of time that the above steps will be repeated is based on a hyperparameter called *epochs*. Other hyperparameters that can be tuning are the *cross-over probability*, i.e. the probability that a couple generate an offspring, the *mutation probability*, i.e. the probability that an individual will perform a mutation, and the *population size*, i.e. the fixed number of individual that compose the population, greater it's the population more variability will have the solutions.

The concept of better or worse individuals (i.e. solutions) is based on the *fitness function*. This kind of function can be defined as $f : I \to R$, where I is the set of all possible solutions. Taking again the metaphor with the natural selection, the fitness function tell us how much fit is this individual. Clearly, as we can think, the more fit subject win against the less fit, so during the selection phase we will choose who to eliminate based on this kind of function.

Try to formalize the steps into a more detailed fashion.

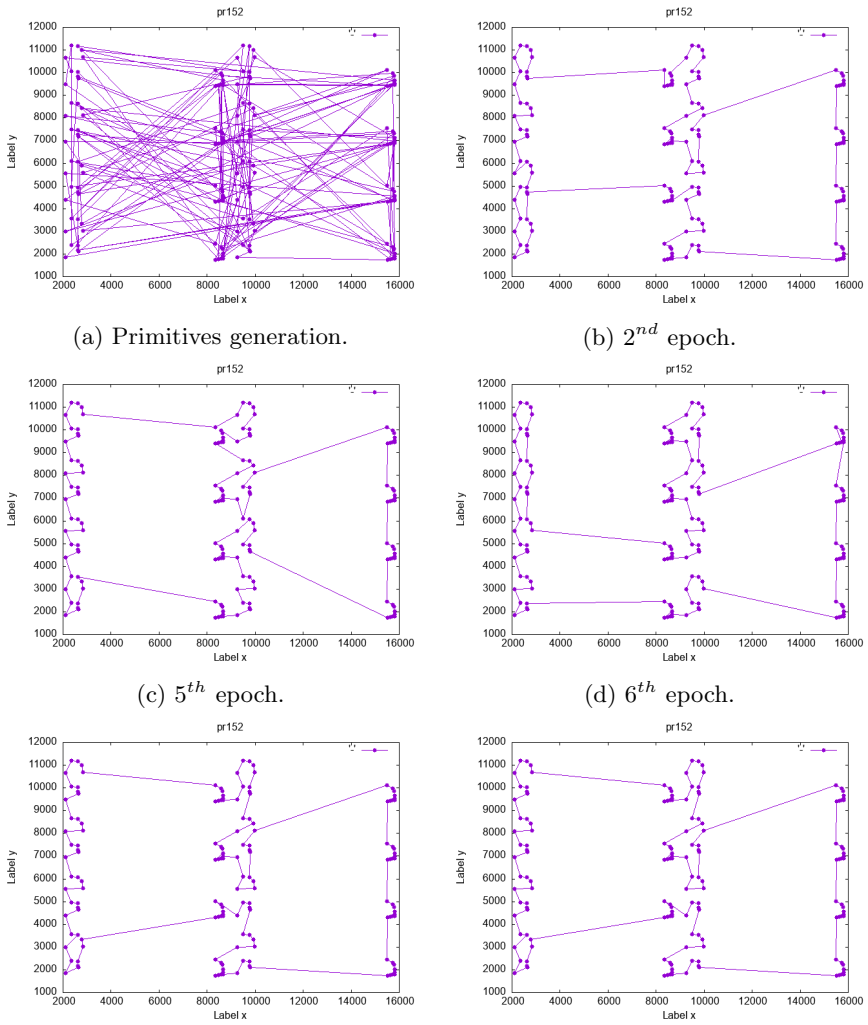
Algorithm 12: Genetic Algorithm (GA)

Data: *nodes*, *epochs*, *crossover_proba*, *mutation_proba*, *population_size*,
Result: x, solution for TSP

```
population ← Generate the primitives randomly
while epochs do
    crossover_phase(population, crossover_proba)
    selection_phase(population)
    mutation_phase(population, mutation_proba)
end
champion ← Take the individual of population with best fitness function
return champion
```

The search of the best solution in Genetic Algorithm doesn't imply a con-

tinuous improvement of the solution, because the steps don't guarantee to find a better solution with respect to the previous generation. As we can see in Figure 10 the solution improves only at some epochs, while in others it will keep the same.



(a) Primitives generation.

(b) 2^{nd} epoch.

(c) 5^{th} epoch.

(d) 6^{th} epoch.

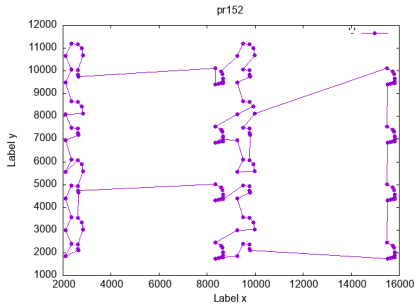
(e) 10^{th} epoch.

(f) 20^{th} epoch.

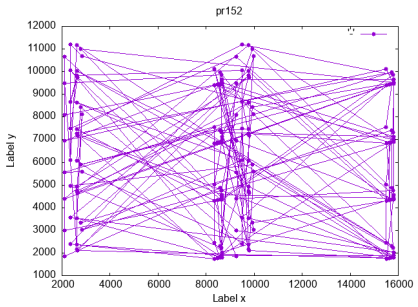
Figure 10: Genetic Algorithm improving epochs.

At first look, the Genetic Algorithm can be viewed as a combinatorial method that tries several random combinations of arcs (i.e. solutions) and hope that the merge of two can lead to a better solution. In order to achieve better results, we

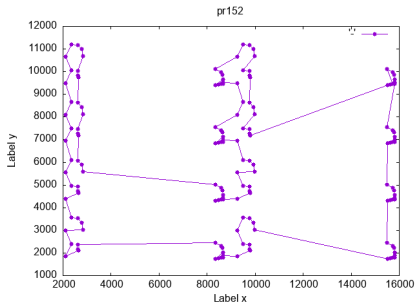
can make a small improvement, but that can get a really better performance. Instead of take and add the offspring as a pure merge of two nodes, we could then apply a quick optimization using the 2-OPT method. Formally, when we generate a new offspring we actually kick the solution from the two parents to a new instance that is, ideally, in another slope of the objective function. One interesting thing will be to discover this new slope and at least reach its minimum, commonly a local minimum. Here, come to play the 2-OPT method that allows to us to search the local minimum of this slope. The new offspring to add will be the merge between the two parent nodes, but with the improvements due to the optimization method. We actually, at each generation, walk from local minimum to local minimum, and we hope to reach a good solution that will be as close as possible to the optimal one. On the Figure 11 we show how the same algorithm will change its behaviour in the case of offspring refined with and without 2-OPT. The algorithm with 2-OPT spent 6749.69 ms, while that without 2-OPT spent, 40654.78 ms.It's immediately recognizable which solution is better. The Genetic Algorithm with 2-OPT is, without any doubts, much better in time and accuracy performance with respect to the algorithm without the refinement method.



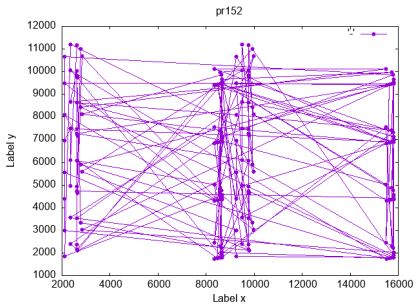
(a) With 2-OPT - 2^{nd} epoch.



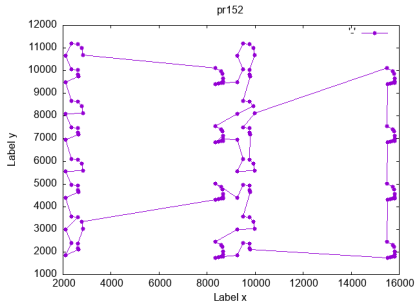
(b) Without 2-OPT - 2^{st} epoch.



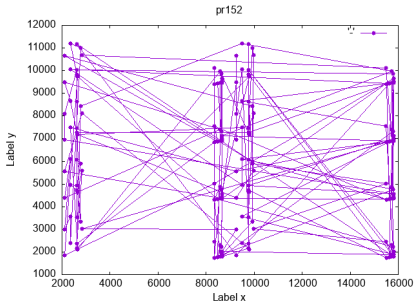
(c) With 2-OPT - 6^{th} epoch.



(d) Without 2-OPT - 6^{th} epoch.



(e) With 2-OPT - 20^{th} epoch.



(f) Without 2-OPT - 400^{th} epoch.

Figure 11: Genetic Algorithm with and without the 2-OPT refinement. The algorithm with 2-OPT spent 6749.69 ms, while that without 2-OPT spent, 40654.78 ms.

5 Experiments

On the following sections we will show all our performance results with an explanation and an interpretation of such those.

5.1 Optimal solutions algorithms: Compact models

Between MTZ and GG, we can see a definitively better behaviour about the second over the first. For all seeds and all time ratios, the two lines are clearly separated, and they never cross one each other. However, when the MTZ is updated with Lazy Constraints technique and/or Subtour Elimination Constraints, the method obtains a great improvement. Both MTZ with Lazy Constraints and MTZ with also SEC get better results with respect to GG. This behaviour isn't seed dependent, i.e. also changing the seed of the performance lines doesn't change too much to change the interpretation.

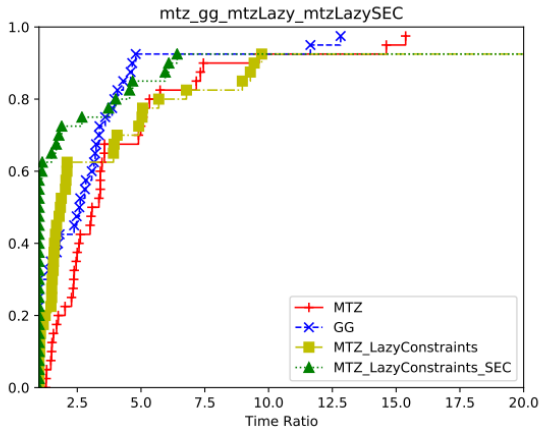


Figure 12: MTZ, GG, MTZ Lazy, MTZ Lazy SEC with random seed: 121324

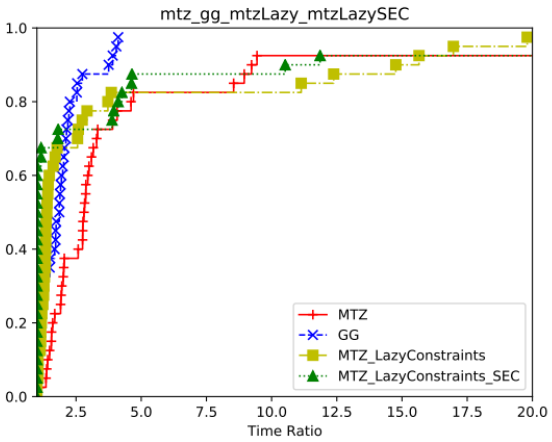


Figure 13: MTZ, GG, MTZ Lazy, MTZ Lazy SEC with random seed: 7485

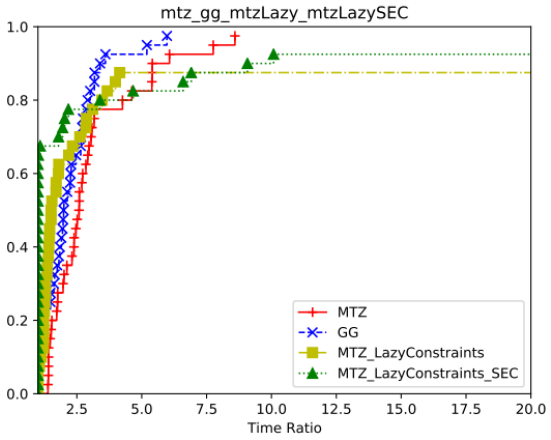


Figure 14: MTZ, GG, MTZ Lazy, MTZ Lazy SEC with random seed: 89

5.2 Optimal solutions algorithms: Dantzig, Fulkerson and Johnson based techniques

Also, on these charts, we can see a clear trend about the performance of the three methods. Concorde is clearly better with respect to Callback and Benders, while Callback win almost always against Benders. This behaviour remains still with high time ratio, this show the high and marked differences of the performances between the algorithms.

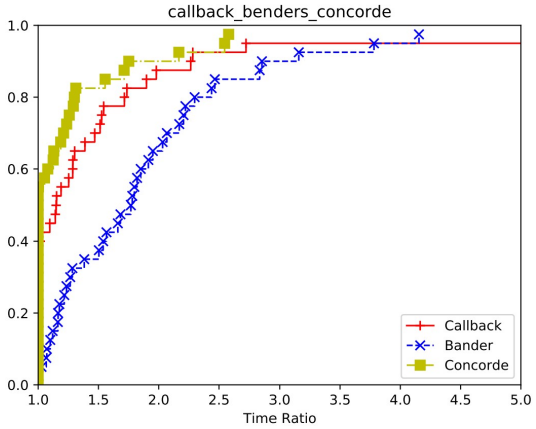


Figure 15: Callback, Benders and Concorde with random seed: 121324

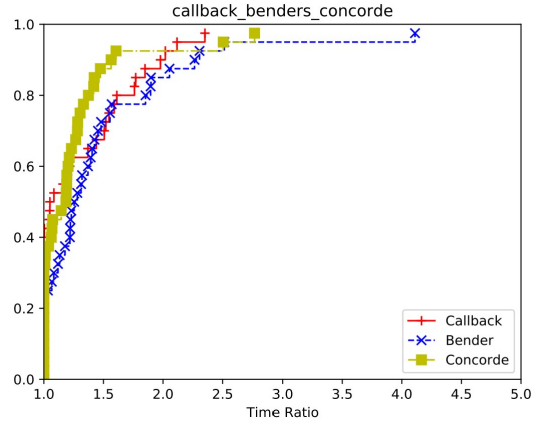


Figure 16: Callback, Benders and Concorde with random seed: 7485

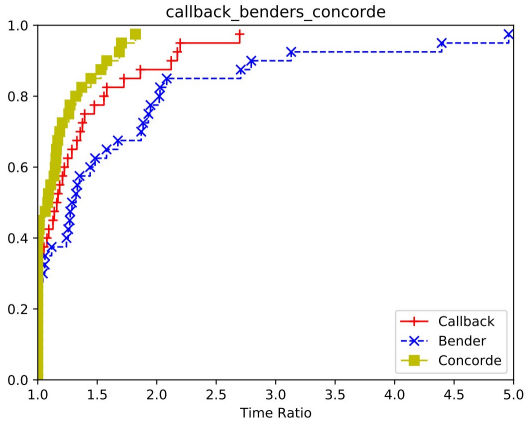


Figure 17: Callback, Benders and Concorde with random seed: 89

5.3 Refinement Heuristics

Hard and Soft fixing have a really similar performance. We can see that at time ratio equal to 1.0 both are excellent and also when it increases the two method are pretty similar. However, the soft method has a bit better results with respect to the hard one.

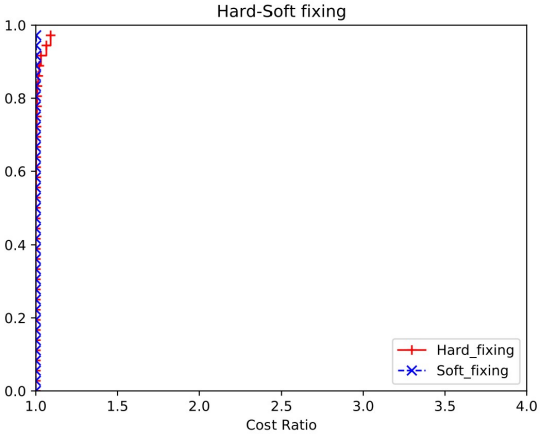


Figure 18: Hard vs Soft fixing

5.4 Constructive Heuristics

We can see that the three methods are pretty similar. Greedy seems to perform a bit better than the GRASP ones. In fact, accounting the Greedy method as a particular case of GRASP (i.e. Grasp(0-1) = Greedy), it's easy to bring that less neighbours the GRASP keeps and less probability to take a random neighbours instead of the nearest one there is, then better are the performance of the GRASP method.

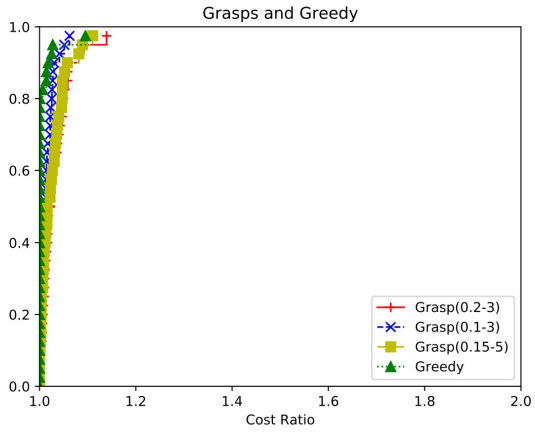


Figure 19: GRASP and Greedy (Nearest Neighbors).

5.5 Metaheuristics

Variable Neighbors Search, Tabu Search and Genetic Algorithm have pretty similar performances. VNS achieves a slightly better results with respect to the others, but the differences are minimal.

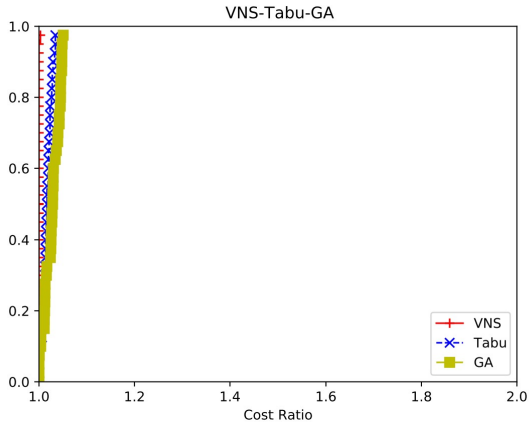


Figure 20: Variable Neighbors Search, Tabu Search and Genetic Algorithm.

6 Conclusions

As we can see on the previous sections, Travelling Salesman Problem is solvable with really several techniques. We could focus our attention on the time spent to find the optimal solutions or on the goodness of the solution found given certain amount of time. This duality can lead to a large field of applicability aimed to satisfy all client's requirements: how time is spent and how much good the solution is. Clearly, all these parameters are slightly machine-dependent, i.e. they could improve if the algorithms run on machines with better hardware specifications. Given the analysis made on each technique, we can confirm that Concorde and Callback algorithms find the optimal solution with definitively better performance with respect to all others, while when we run the constructive heuristic algorithms for a finite time the Greedy results to be slightly better compared to GRASP implementations. As we expected for Metaheuristics the VNS implementation find a better solution comparing with GA and Tabu search, however also the Soft-fixing algorithm produces outcome that are notable, and it performs better comparing with Hard-fixing.

References

[1] Wikipedia's contributors. (2021, July 25). Travelling salesman problem. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1035459634

[2] Local branching. (2002, May 30). Travelling salesman problem. from http://www.dei.unipd.it/~fisch/ricop/RO2/2003-Mathematical_Programming%20%5blocal%20branching%5d.pdf

[3] Heuristics for travelling salesman problem. from http://www.dei.unipd.it/~fisch/ricop/RO2/Heuristics%20for%20the%20Traveling%20Salesman%20Problem%20By%20Christian%20Nillson.pdf

[4] Variable Neighborhood Search for travelling salesman problem (2003, July) from http://www.dei.unipd.it/~fisch/ricop/RO2/VNS_tutorial.pdf

[5] A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem. *A.J. Orman and H.P. Williams.*

[6] Tabu Search for travelling salesman problem. from http://www.dei.unipd.it/~fisch/ricop/RO2/tabu_search_tutorial.pdf

[7] C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer Programming Formulation of Traveling Salesman Problems. from http://doi.acm.org/10.1145/321043.321046.

[8] Wikipedia's contributors. (2021, July 25). Matheuristics. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Matheuristic

[9] Wikipedia's contributors. Metaheuristic. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Metaheuristic

[10] Wikipedia's contributors. (2021, July 5). Genetic algorithm. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Genetic_algorithm&oldid=1032041921