DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

University of Padova
Master Degree in Computer Engineering

Solution strategies for the Traveling Salesman Problem

Denis Deronjic
1231829

Stefano Ivancich
1227846

Academic year 2020/2021
August 23rd, 2021

Contents

Chapter 1

Introduction

Many decision problems in industry, logistics, and telecommunications can be seen as satisfiability or optimization problems. The main paradigms used to solve these problems are Constraint Programming (CP) and Mixed Integer Programming (MIP). The purpose of this paper is to present, analyze and compare different approaches to solve the Traveling Salesman Problem (TSP) [1] as a way to understand more deeply the various issues that arise when approaching those kind of problems. This was developed during the 2021 Operations Research 2 course held by Prof. Matteo Fischetti at University of Padua.

In the next chapters we are going to present all the work done, which includes mathematical formulations, implementation and testing phases. In particular, this report is structured as follows:

- in this chapter we present the TSP history and its formulations,
- in Chapter 2 we will present all compact models we have studied and implemented, showing the pros and cons of each model,
- in Chapter 3 we will present a set of algorithms that can solve the TSP, finding the shortest tour and the solution is proved to be the optimal,
- in Chapter 4 we present math heuristics algorithms that use mathematical programming at their core around which heuristics are built,
- in Chapter 5 we will see more heuristics algorithms that can find a solution to the TSP but this solution is not proved to be optimal,

- in Chapter 6 we discuss about m eta heuristics that are problem independent heuristics,
- in Chapter 7 we will present the conclusions of our work.

All the source code developed is available at

`https://github.com/deno750/TSP_Optimization`

In this project we solve the TSP, first by modifying its formulation, then by using techniques that add constraints iteratively, and then by adopting several heuristic methods. The MIP solver used in this work is CPLEX while Visual Studio Code is used for the Ansi C programming. The instances for the TSP were taken from the TSPLIB library [6].

1.1 Problem history

The origin of the TSP is not clear. The German handbook "Der Handlungsreisende—Von einem alten Commis-Voyageur" from 1832 reports an explicit description of the TSP, made by a traveling salesman himself. The first mathematical formulation dates back to the 1800s by William Rowan Hamilton and Thomas Kirkman. One famous real example of the TSP problem is the one of salesman H. M. Cleveland, who worked for the Page Seed Company in 1925, and had to travel over 350 cities in the Maine in 90 days. Later in the 19th century several guides such as L. P. Brockett's "Commercial Traveller's Guide Book" appeared describing well-chosen routes in different countries. In 1972 Richard Manning Karp proved that the Hamiltonian cycle problem was NPcomplete, that implies the NP-hardness of the TSP.

1.2 Problem formulation

The Traveling Salesman Problem (TSP) consists in finding a Hamiltonian circuit of minimum cost on a given directed graph $G = (V, A)$. This problem arises naturally when it is necessary to distribute a given product to a set of locations, or when we need to optimally sequence a set of jobs. In some cases, the problem can be analogously defined on a undirected graph; this happens when the cost associated with an arc does not depend on its orientation.

In this document we are mainly using the Symmetric Traveling Salesman problem (STSP), that is defined as follows: consider an undirected weighted complete graph $G = (V, E)$ where $V = \{v_1, ..., v_n\}$ is the set of the n nodes and E is the set of the $n(n-1)$ edges. Also, let $c : E \to \mathbb{R}^+$ be a function that assigns to each edge $e = \{i, j\} \in E$ the cost $c(e) = c_e$. This function represent an arbitrary distances or weight.

The TSP problem ask to sequence of edges (or nodes) that forms the tour of minimum cost, also called optimal tour. To address this task, we need to formulate the problem as a mathematical model. The symmetric TSP can be expressed in the following conventional form, also known as Dantzig-Fulkerson-Johnson formulation (DFJ):

$$\min \sum_{e \in E} c_e x_e \quad \text{SUBJECT TO} \tag{1.1}$$

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \tag{1.2}$$

$$x_e \in \{0, 1\} \quad \forall e \in E \tag{1.3}$$

This model uses a polynomial number of constraints, so it's compact, but it's not complete, since it produces subtours.

So the Subtour Elimination Condition (SEC) is introduced, and it can be formulated as follows:

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V : |S| \geq 3 \tag{1.4}$$

The problem with this formulation is that it introduces 2^{n-1} constraints and $n(n-1)$ binary variables. The exponential number of constraints makes it impractical to solve directly the model.

Chapter 2

Compact models

2.1 Naive model

We first present a model that is not a compact model, but it will be used as basis for other models that will just add constraints on it. It is composed of in degree and out degree constraints (eq. 2.2 and eq. 2.3), that are imposing that for a node the total number of ingoing edges is 1 and the total number of outgoing edges is also 1.

$$\min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \quad \text{SUBJECT TO} \tag{2.1}$$

$$\sum_{i \in V} x_{ih} = 1 \quad \forall h \in V \tag{2.2}$$

$$\sum_{j \in V} x_{jh} = 1 \quad \forall h \in V \tag{2.3}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in V \tag{2.4}$$

An example of solution provided by this model can be seen in Fig. 2.1. As you can notice there are subtours. One way to remove them is to add the SEC constraints 1.4, but the number of such constraints is exponential in the number of nodes. While in fig. 2.2 the subtour elimination constraints are used. So, now we are going to overview 2 models that are both introducing just $O(n^2)$ new constraints.

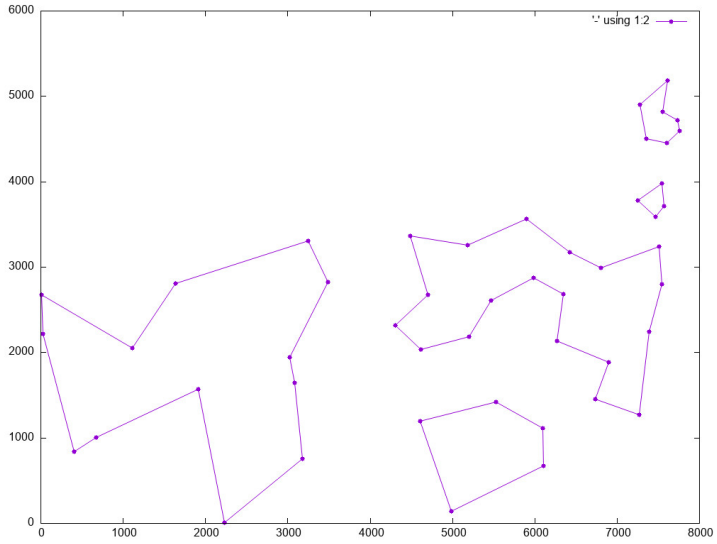

Figure 2.1: An example of solution with no subtour elimination constraints.

2.2 Miller-Tucker-Zemlin

The Miller-Tucker-Zemlin [2] model introduces the variables u_i that represent the position of node i in the optimal tour, from a first (arbitrary) node. The idea is that the values of u_i increase during the tour and are assigned as follows: the first node has no u variable associated to it, the second node has $u_i = 0$, the third node $u_i = 1$, and so on until the last node of the tour, which has $u_i = n - 2$. This mechanism avoids the generation of subtours since, in order to be closed, each tour requires a special node that does not have the sequence constraint on u: such property allows to connect the last node (with the largest value of u) to the first node (with the smallest).

In fig. 2.3 you can see a representation of this process.
So, the following constraints are added to the naive model:

$$u_j \geq u_i + 1 - M(1 - x_{ij}) \quad \forall i, j \in V \quad i, j \neq 1 \tag{2.5}$$

$$x_{ij} + x_{ji} \leq 1 \tag{2.6}$$

$$0 \leq u_i \leq n - 2 \quad \text{Integer} \quad \forall i \in V \setminus \{1\} \tag{2.7}$$

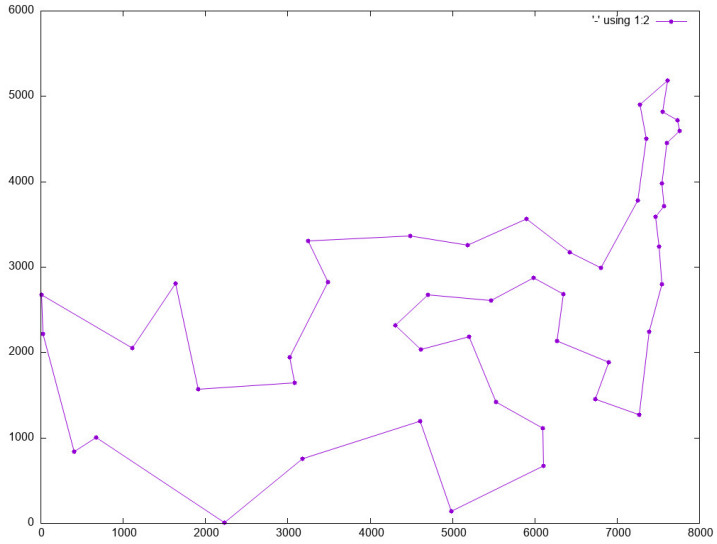

Figure 2.2: An example of solution with the MTZ subtour elimination constraints.

$$u_1 = 0 \tag{2.8}$$

The constraint (2.6) is not actually necessary to find the optimum, but in practice it could improve the convergence speed for some instances.

The MTZ formulation adds $O(n)$ variables (u_i) and $O(n^2)$ constraints (2.5).

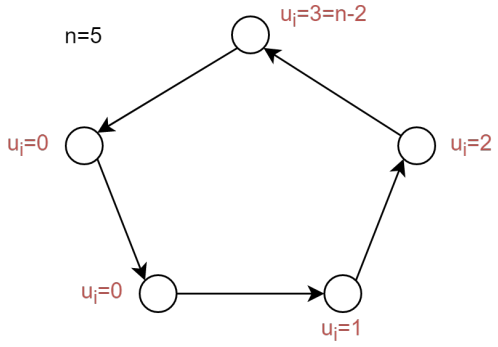

Figure 2.3: Example of the MTZ formulation.

2.3 GG - Flow 1 model (Gavish and Graves)

The Gavish and Graves model also known as the flow commodity model [3] its a modified version of the MTZ in which the concept of flow through the edges is introduced. The basic idea is that the salesman is carrying $n-1$ units of some commodity and after leaving the first node he drops 1 unit at each node that he visits. The integer variables y_{ij} are introduced to represent the amount that pass from node i to node j. In fig. 2.4 you can see a representation of this process.
So, the following constraints are added to the naive model:

$$\sum_{i \in V} \sum_{i \neq h} y_{ih} - \sum_{j \in V} \sum_{j \neq h} y_{hj} = 1 \quad \forall h \in V \setminus \{1\} \tag{2.9}$$

$$y_{ij} \leq (n-2)x_{ij} \quad \forall i, j \in V \setminus \{1\} \tag{2.10}$$

$$y_{1j} = (n-1)x_{1j} \quad \forall j \in V \setminus \{1\} \tag{2.11}$$

$$0 \leq y_{ij} \leq n-2 \quad \text{Integer} \quad \forall i, j \in V \setminus \{1\} \tag{2.12}$$

$$0 \leq y_{1j} \leq n-1 \quad \text{Integer} \quad \forall j \in V \setminus \{1\} \tag{2.13}$$

$$y_{i1} = 0 \quad \forall i \in V \setminus \{1\} \tag{2.14}$$

The constraint (2.9) indicates that the in Inflow - outflow must be equal to 1. The constraint (2.10) links the new variables y_{ij} to x_{ij}.
This formulation adds $O(n^2)$ new variables (y_{ij}) and $O(n^2)$ constraints (2.10).

2.4 Comparison between Compact Models

We compared different implementation of MTZ with GG on a bunch of datasets that have less than 60 node, with a timelimit of 3600 seconds imposed with the cplex function `CPXsetdblparam(env, CPXPARAM_TimeLimit, time_limit)`. In particular, the implementations tested are:

- MTZ with Static constraints

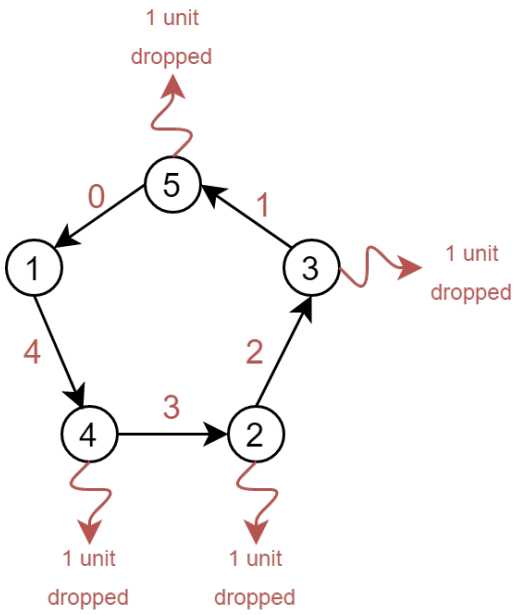

Figure 2.4: Single Commodity Flow (GG)

- MTZ with Lazy Constraints
- MTZ with Static constraints + subtour elimination of degree 2
- MTZ with Lazy Constraints + subtour elimination of degree 2
- GG

As mentioned before these compact models uses $O(n^2)$ variables and constraints, that is why they were tested on small instances. However, as you can clearly see in fig. 2.5 the GG model is much faster than the different implementations of the MTZ model, since the its performance profile is always above the MTZ profile. While MTZ with Lazy Constraints and subtour elimination of degree 2 seems to be the second fastest method.

In any case the solutions reached by these methods are always the optimum.

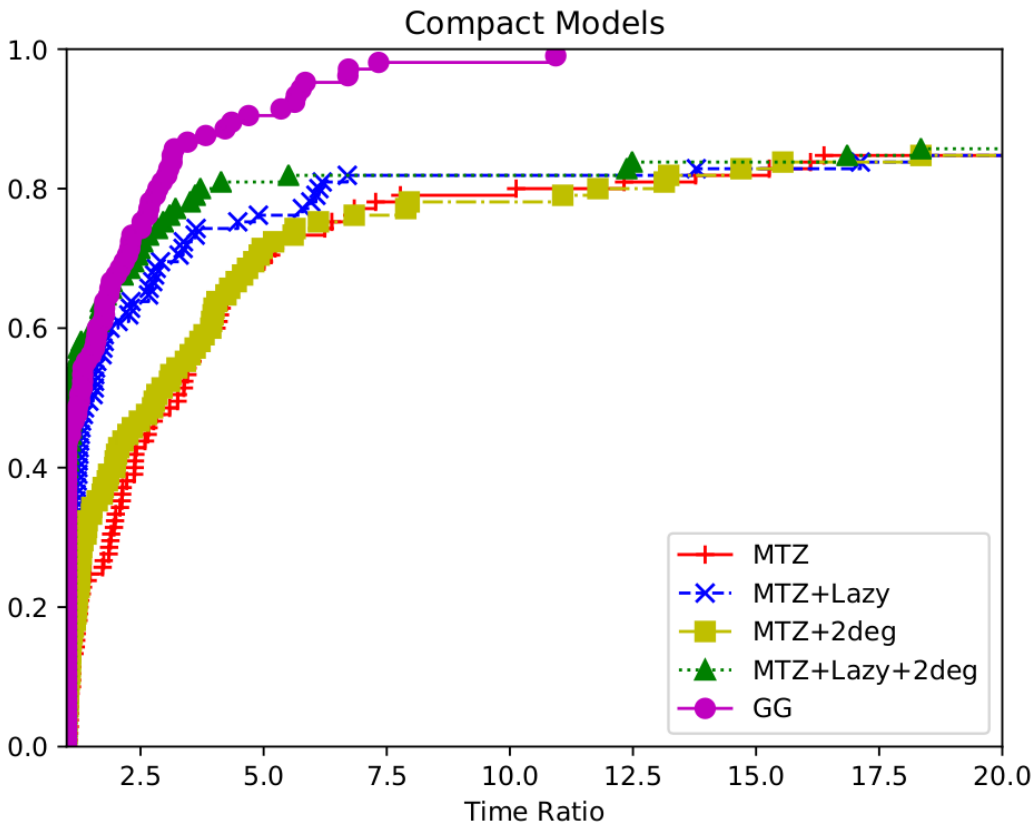

Figure 2.5: Compact models comparison: MTZ vs GG

Chapter 3

Exact Models

This section introduces methods that solve to optimality a MIP problem using the basic CPLEX MIP solver. The first approach iteratively adds the SEC constraints and solves the problem until the optimal solution without subtours is found, while the second uses a callback function that is called periodically by the MIP optimizer (CPLEX) in order to allow the user to query or modify the state of the optimization.

3.1 Loop Method

As said before, the Dantzig-Fulkerson-Johnson model just adds the (3.1) subtour constraints in the naive model, but the number of such constraints is exponential.

In the previous chapter we discussed some compact models to solve this issue. Here we present another way called "the loop method".

The Loop method is based on the idea that the great majority of all possible subtours is made by very bad edge's combinations, corresponding to branches of the branching tree that the MIP solver would quickly discard while minimizing the cost of the solution. So, the method focuses only on those subtours which are selected by the solver, and forbid it to choose them.

Bender's implementation solves itereatively the DFJ model. It starts from the degree constraints and solves the problem with Cplex's MIP solver. When the solution is found, it checks whether subtours are present; if so, for each subtour adds the SECs and solves the new model until a solution is found or the time limit is reached.

In particular, if the solution has m subtours and S_k is a subtour, then the Loop method adds the following constraints:

$$\sum_{e \in E(S_k)} x_e \leq |S_k| - 1 \quad k = 1, ..., m \tag{3.1}$$

The pseudo-code of Bender's implementation of the LOOP method is shown in (Algorithm 1).

Algorithm 1: Bender implementation of the DFJ model

Input : TSP instance.
Output: a valid tour (CPLEX solution).

```
model ⟵ naive model
solution ⟵ solve model
while solution has subtours do
    foreach subtour in solution do
        sec_constraints ⟵ generate SECs constraints of subtour
        add sec_constraints to model
    end
    solution ⟵ solve model
end
```

This procedure avoids the generation of an exponential number of constraints, however rebuilding and reoptimizing the model from scratch at each iteration is the major drawback.

3.2 Incumbent callback

Another way to add SEC constraints is to exploit the branch-and-cut technique used (in this case by CPLEX) to solve the problem. The branch-and-cut algorithm provided by CPLEX at the root node applies some preprocessing steps. For each node of the branching tree, it applies dozens of cut separation families (Gomory, Clique, 0-1/2 cuts, ...) and then, after the relaxation is calculated, some primal heuristics are applied to find better and better incumbent solutions. They get as input the fractional solution, applies those heuristics to transform it in integer and than, if the cost is better than the incumbent solution, this solution is updated. This integer solution

probably contains subtours. Between the heuristic step and the update of the incumbent, is possible to instruct CPLEX to call an our custom callback function. In this function we check in the same way of the bender's implementation if the instance contains subtours. If so, we add the SECs for the connected components. In Fig.3.1 this mechanism is explained.

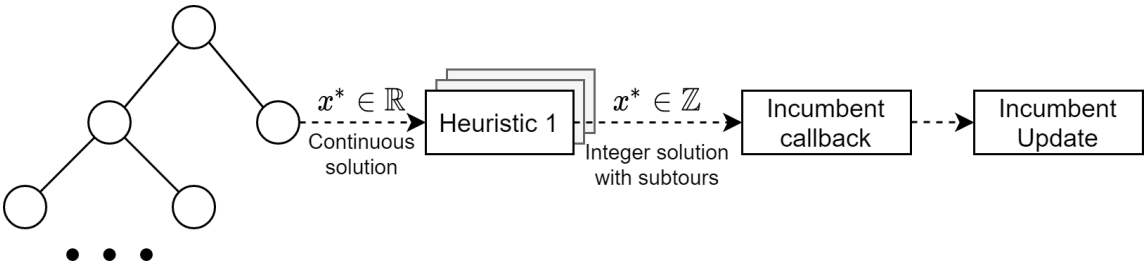

Figure 3.1: CPLEX Branch and Cut Incumbent Callback mechanism

With this method only a single decision tree is generated avoiding to generate multiple branching trees as in Loop method. This hopefully leads in a faster computation.

The pseudo-code of Incumbent Callback is shown in (Algorithm 2).

Algorithm 2: Incumbent Callback implementation of the DFJ model

```
Function IncumbentCallback(model, solution):
if solution has subtours then
    foreach subtour in solution do
        sec_constraints ⟵ generate SECs constraints of subtour
        add sec_constraints to model
    end
end
```

In the incumbent callback is possible to apply custom heuristics which can help CPLEX finding quickly a better solution. From the integer solution that CPLEX returns in this callback, if there's only one connected component the 2-opt heuristic which will be described in 5.2.1 can be applied in order to remove crossing edges from the solution and find a better one hopefully reducing the computation time. The solution found by 2-opt is added to CPLEX with the help of `CPXcallbackpostheursoln` function.

3.3 User-cut callback

In this section we're going to describe an advanced usage of CPLEX's callback. At each node of the branching tree, CPLEX allows to call a user-cut callback which is our custom function called in the relaxed solution of the problem. The relaxed solution does not take into considerations the integer constraints of the variables. In this way whoever needs to create custom cuts for the relaxed problem, can do that easily by using this type of callbacks. In the TSP though, the solution must be integer. Unfortunately the algorithms used in the previous models cannot be used in this case. An example is the Union find algorithm which only works with integer solution. For that reason some functions from the concorde [7] library are used. For instance the function that counts the components in a fractional solution is `CCcut_connect_components` and the function which calculates the min-cut of a flow problem is `CCcut_violated_cuts`. The implemented callback works like the Incumbent callback in the sense that the number of components in the relaxed solution returned by CPLEX is computed using the aforementioned function, and for each component, SEC is applied as global constraint. The difference with the incumbent callback comes when the number of components found is one. If in the incumbent callback the number of components is one, the solution is treated as feasible; in user-cut instead we need to calculate the min-cut on the graph.

For instance in TSP for each cut $(S, V \setminus S)$, we need to satisfy

$$\sum_{(i,j)\in\delta(S)} {x_{ij}}^* \geq 2 \quad \forall S \in V, S \neq \emptyset \tag{3.2}$$

where $i \in S$, $j \in V \setminus S$, ${x_{ij}}^* \geq 0$ is the value of the edge (i, j) in the relaxed solution and $\delta(S)$ is the cut-set of a cut S.

The concorde function `CCcut_violated_cuts` returns the cuts which violate 3.2; then we apply on each cut the SECs.

Unfortunately it is not possible to apply this callback at every node of the branching tree because it would create a huge overhead due to the time complexity of the concorde's algorithms. As a consequence we apply the cuts with a probability of 10%. An alternative method which could be implemented is applying the cuts when the depth of the node in the branching tree is less than a threshold (for example 5).

One important notice is that User-Cuts callbacks are a subroutine of the Incumbent callback method. For instance, the callback, which was defined in

incumbent callback method is called when an integer solution is found while the user cuts are called when a fractional solution is found. This generally helps CPLEX to apply some important constraints before the update of the incumbent giving the chance to reduce the size of the branching tree and so the computing time.

3.4 Comparison between Exact Models

In this section we report the comparison between the exact methods we implemented, in particular we tested the models on 34 instances from 130 to 700 nodes. The plot in fig. 3.2 clearly shows that the user-cut callback gives the best results, with its profile being above both Bender and Incumbent Callback, even if it employs time expensive routines, based on concorde API. Incumbent with 2-opt does not perform way better than Incumbent only callback. Their results are pretty the same Bender's implementation is the worst because it spends a lot of time rebuilding the model from scratch at each iteration, while the callbacks methods don't do that.

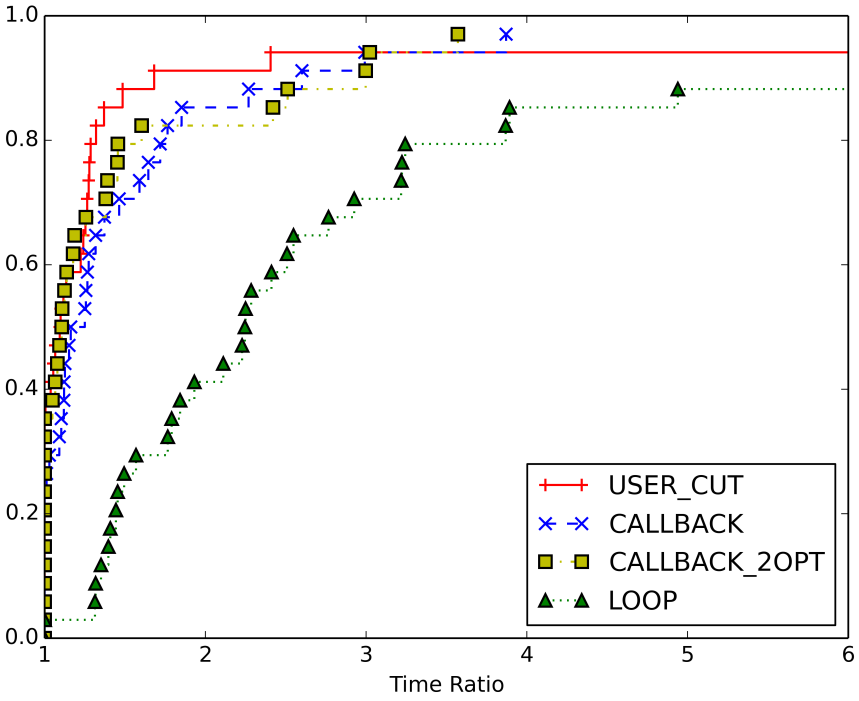

Figure 3.2: Exact models comparison: Loop vs Callback vs Callback with 2-opt vs Usercut

Chapter 4

Matheuristics Models

An Heuristic according to [4], is defined as "any approach to problem solving or self-discovery that employs a practical method, not guaranteed to be optimal, perfect, logical, or rational, but instead sufficient for reaching an immediate goal. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution". According to this definition, Heuristic algorithms are designed to search for good solutions in a reasonable time by sacrificing optimality. In this chapter we are going to talk about Matheuristics. Matheuristics algorithms combine the use of heuristic methods and Mathematical Programming (MP). The techniques that we are going to present are hard fixing and local branching.

4.1 Hard Fixing

The idea behind the hard fixing Heuristic is to iteratively fix some variables (edges) of the reference solution computed by CPLEX and then try to solve the new simplified problem with the MIP solver. When the solution is found, the fixed edges are unfixed and then the loop restarts until a time limit is reached. Since at each iteration the solution doesn't violate the SEC constraints, this accelerates the time for optimizing but the solution it's not guaranteed to be optimal. Furthermore at each iteration the solution can just be equal or better than the previous one. In particular it starts with an initialization step in which it instantiate the TSP model without SECs constraints but only with the degree constraints and compute quickly an

incumbent solution with the TSP solver. This solution is probably very far from the optimal. So, hard fixing iteratively fixes a given percentage of edges, the fixing of variable x_{ij} can be done by setting its lower bound to 1 in the model held by CPLEX. Then our best TSP solver, the user-cut callback (chapter 3.3), is used to solve that model. Finally all the edges are unfixed by setting the lower bound back to 0. This process continue until the time limit is reached. Since the model has some fixed edges, it has a fewer number of variables so an exact solver can easily handle such reduced model. Furthermore, the algorithm keeps iterating using the same percentage until it is not able to find a better solution for 5 consecutive times (or timelimit occurs); then this percentage increases (or decreases), more specifically in our implementation the sequence of percentages we used is 90%, 80%, 50% and 30%. The algorithm after some iterations tends to make very small improvements making the algorithm never change the fixing percentage. To overcome this issue we forced the algorithm to change the fixing percentage when the number of consecutive small improvements reaches the maximum limit allowed. In this way the algorithm tends to explore more widely new neighborhoods by the end of the time limit.

An example of the solution space exploration made by this technique can be viewed in Fig. 4.1, as you can see smaller fixing-probabilities result in a faster and more feasible computation but the solution won't be much different from the current one due to the smaller neighborhood. The pseudo-code of the Hard Fixing is shown in (Algorithm 3).

4.2 Local Branching

The Local Branching also known as soft fixing, proposed by Fischetti [5], has the idea to add a constraint to the mathematical model called local branching constraint, which forces to fix a certain number of variables without choosing them explicitly.

The current solution x^h can be seen has a binary vector (eg. $[0, 1, 0, ...]$) of length $|E|$, where an element is set to 1 if the corresponding edge in the solution is used, otherwise is set to 0. From this concept, we can use a notion of distance from the two arrays called Hamming Distance, that is defined as the number of positions at which the corresponding symbols are different. In

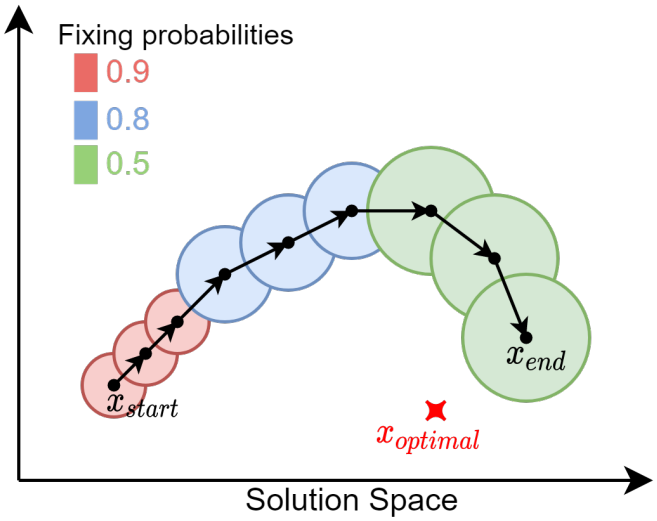

Figure 4.1: Example of hard fixing neighborhood search

particular it can be calculated as:

$$H(x, x^h) = \sum_{e \in E:\ x_e^h = 1} (1 - x_e) + \sum_{e \in E:\ x_e^h = 0} x_e \tag{4.1}$$

where the first sum can be seen as the number of flips from 1 to 0, and the second sum as the number of flips from 0 to 1.

Looking again at the solution space, the hamming distance of a solution can be seen as the radius r distancing it from other solutions, similarly as the fixing-probability of the hard-fixing method. So now we can impose a limit to this distance:

$$\sum_{e \in E:\ x_e^h = 1} (1 - x_e) + \sum_{e \in E:\ x_e^h = 0} x_e \leq r \tag{4.2}$$

For the TSP problem, the second summation of eq. 4.2 can be removed because the number of flips from 1 to 0 is equal to the number of flips from 0 to 1. For instance

$$\sum_{e \in E:\ x_e^h = 1} (1 - x_e) = \sum_{e \in E:\ x_e^h = 0} x_e \tag{4.3}$$

Then removing the second summation from 4.2 it becomes

$$\sum_{e \in E:\ x_e^h = 1} (1 - x_e) \leq k \tag{4.4}$$

where $k = \frac{r}{2}$. With some algebraic passages we led to the final formulation of the soft-fixing constraints:

$$\sum_{e:\ x_e^h = 1} x_e \geq n - k \tag{4.5}$$

where n is the number of edges equals to 1 in a TSP tour (i.e. $n = |V|$) and k is a parameter that represents the number of edges of the incumbent solution that CPLEX is free to reconsider in the upcoming re-optimization of the problem. High values of k may allow too much variations and do not respect the idea of a local constraint. Since empirically has been proved that values up to 15 are effective, in our implementation we chose a list of different values, in particular $k = [3, 5, 7, 9]$. The algorithm is similar to the hard-fix, here we just change the neighborhood by controlling the radius parameter k instead of the fixing-probability. The pseudo-code of the Soft Fixing is shown in (Algorithm 4).

4.3 Comparison between Matheuristics

The matheuristics test-set contains 18 instances from the TSPLIB with 300 to 1000 nodes, over a time limit of 20 minutes (1200 seconds). The performance profiling for matheuristics is shown in Fig. 4.2. We compared 3 different models, an hard-fixing (HARD-FIX in the plot) that uses always the same fixing-probability of 0.7, an advanced hard-fix (HARD-FIX2 in the plot) that changes fixing-probability over time $[0.9, 0.8, 0.5]$ as described in (Algorithm 3) and soft-fixing with changing radius $[3, 5, 7, 9]$ as described in (Algorithm 4). The plot clearly shows that soft fixing is the worst algorithm, but is still competitive for instances with less than 600 nodes. While the advanced hard-fixing is by far the best. Data in table 4.1 shows that the cost of solutions found by HARD-FIX2 is within 1% in all instances except for dsj1000 (30%) and pr1002 (3%) that are the largest. Furthermore, in Fig. 4.3 we plotted the cost at each iteration over the dfj1000 tsp instance for Hard Fixing with single fixing-probabilities of 70% and 90%, the advanced hard-fixing and soft-fixing. As you can see the advanced hard-fixing outperform the other methods and is much more smooth.

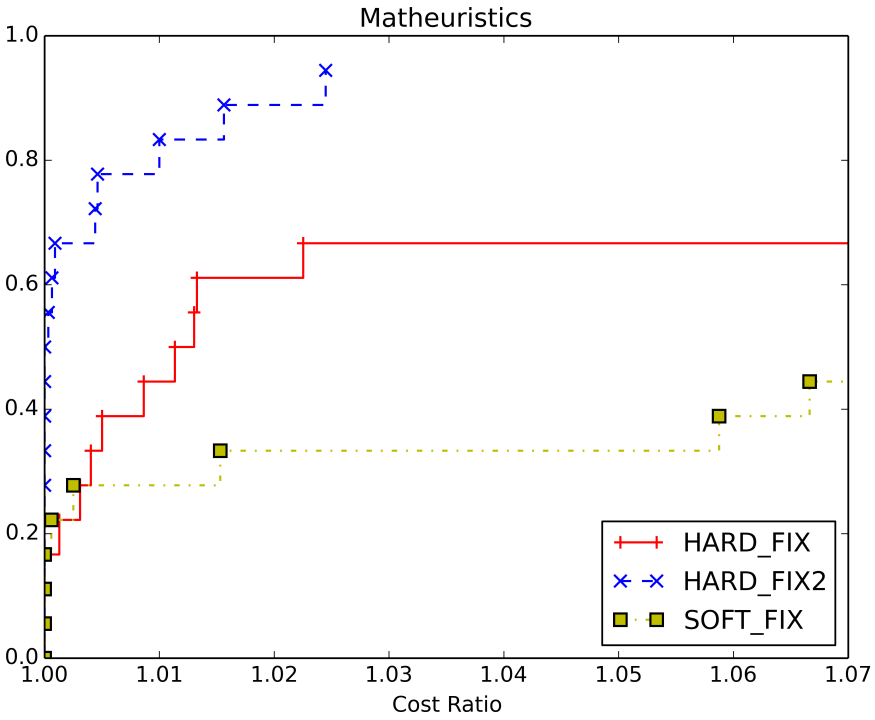

Figure 4.2: Matheuristics comparison: Hard-Fix vs Advanced Hard-Fix vs Soft-Fix

Algorithm 3: Hard fixing implementation

Input : TSP instance.
Output: a valid tour.

```
prob_array ⟵ array of probabilities /* eg. [0.9, 0.8, 0.5] */
prob_idx ⟵ 0
number_small_improv ⟵ 0

model ⟵ naive_model (with degree constraints)
solution ⟵ get a feasible solution for model quickly

/* While we are within the time limit and prob array */
while time_elapsed < time_limit AND prob_idx < len(prob_array) do
    time_remain ⟵ time_limit − time_elapsed
    model ⟵ fix some edges according to prob_array[prob_idx]
    solution ⟵ get a solution of model within time_remain

    current_improv ⟵ 1 − cost(solution)/cost(best_solution)

    /* If it's just a small improvement */
    if current_improv < minimum_improv then
        number_small_improv ++

        /* If we had a lot of small improvements */
        if number_small_improv = max_small_improv then
            prob_idx ++ /* Use next probability */
            number_small_improv ⟵ 0
        end
    else
        number_small_improv ⟵ 0
    end

    best_solution ⟵ solution
    model ⟵ unfix edges
end
```

Algorithm 4: Soft fixing implementation

Input : TSP instance.
Output: a valid tour.

```
radius_array ⟵ array of radius /* eg.  [3, 5, 7, 9]                    */
radius_idx ⟵ 0
number_small_improv ⟵ 0

model ⟵ naive_model (with degree constraints)
solution ⟵ get a feasible solution for model quickly

/* While we are within the time limit and radius array                  */
while time_elaps < time_limit AND radius_idx < len(radius_array) do
    time_remain ⟵ time_limit − time_elaps
    model ⟵ fix some edges according to radius_array[radius_idx]
    solution ⟵ get a solution of model within time_remain

    current_improv ⟵ 1 − cost(solution)/cost(best_solution)

    /* If it's just a small improvement                                 */
    if current_improv < minimum_improv then
        number_small_improv ++

        /* If we had a lot of small improvements                        */
        if number_small_improv = max_small_improv then
            radius_idx ++ /* Use next radius                            */
            number_small_improv ⟵ 0
        end
    else
        number_small_improv ⟵ 0
    end

    best_solution ⟵ solution
    model ⟵ unfix edges
end
```

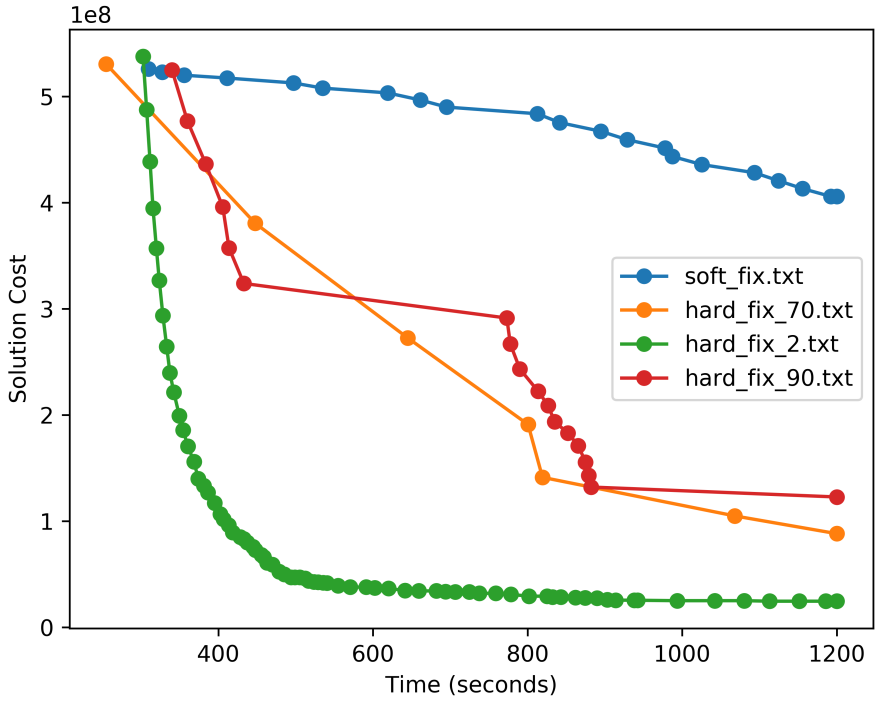

Figure 4.3: Solution vs Timestamp comparison between Matheuristics over the dsj1000 tsp instance

	Models			
Instance	**Real Cost**	**HARD_FIX**	**HARD_FIX2**	**SOFT_FIX**
ali535	202339	207566	206164	202994
p654	34643	1185640	34805	510224
rat783	8806	9014	8896	57132
gr666	294358	300963	299465	633564
d657	48912	50117	50053	104542
pr439	107217	107511	110144	134285
rat575	6773	6804	6834	6821
u724	41910	65876	42770	439548
rd400	15281	15480	15286	15281
u574	36905	37452	37476	38025
vm1084	239297	2177905	388384	7173041
att532	27686	28076	27761	29392
d493	35002	35227	35086	37424
lin318	42029	42159	42449	42029
pr1002	259045	1551030	266639	4579930
gr431	171414	173407	172714	171923
pcb442	50778	50778	50824	50808
dsj1000	18659688	371793030	26526532	332813825

Table 4.1: Results of Hard Fixing and Local Branching algorithms

Chapter 5

Heuristics

The execution of exact algorithms becomes more and more computational demanding with respect to the number of nodes of the problem, until it became unfeasible to solve exactly an instance for the today (2021) technology. For that reason, we can use Heuristics algorithms which don't find the optimal solution but a good one with a reasonable computational cost. We are going to introduce 2 families of heuristics:

- **Constructive heuristics** that generate from scratch an approximate solution for the problem.
- **Refinement heuristics** that improve an existing solution.

5.1 Constructive Heuristics

Constructive heuristics build a solution from scratch in feasible amount of time, usually getting within 10-15% of optimality. This type of heuristics are often use as a starting point for other heuristics, since the quality of the final solution of an heuristic is highly dependent on the starting instance. We are going to introduce the heuristics: Nearest Neighbors and Insertion.

5.1.1 Nearest Neighbors

This is a greedy 2-approximation algorithm (meaning that the solution is at most 100% far from the optimum), that generates in $O(n^2)$ a solution for the TSP instance starting from one node and at each iteration selecting the

next node in the tour that is closest to the current. The pseudocode for the computation of a single starting solution from an arbitrary node is shown in Algorithm 5.

Algorithm 5: Nearest Neighbour implementation

Input : Starting node
Output: a valid tour.

```
tour ⟵ empty tour
solution ⟵ 0
visited ⟵ {start_node} /* visited nodes                    */
curr_node ⟵ start_node

/* While there are nodes to visit                          */
while |visited| ≠ N do
    closest_node ⟵ closest node to curr_node ∉ visited
    solution ⟵ solution + dist(curr_node,closest_node)
    curr_node ⟵ closest_node
    add curr_node to visited
    add edge (curr_node,closest_node) to tour
end
add edge (curr_node,start_node) to tour
solution ⟵ solution + dist(curr_node,start_node)
```

5.1.2 Nearest/Farthest Insertion

This algorithm initialize the tour with 2 farthest (or nearest) nodes (initialization step). Then, at each iteration, the node k not visited whose distance to the tour is minimal (maximal for farthest insertion) is selected (selection step). This node is inserted in the tour in a way that it insertion causes the smallest increase in the tour length (insertion step). This step is done by finding an edge (i, j) in the tour that minimizes $\Delta(i, j, k) = c_{ik} + c_{kj} - c_{ij}$ that is referred as extra-mileage. It works in $O(n^3)$. The bottleneck of this algorithm is the time spent computing k. The pseudocode for the computation of a single starting solution from an arbitrary node is shown in Algorithm 6.

Algorithm 6: Nearest/Farthest Insertion implementation

Input : nothing.
Output: a valid tour.

```
solution ⟵ 0
visited ⟵ {the 2 nearest/farthest nodes} /* visited nodes    */
tour ⟵ edge between the 2 nearest/farthest nodes

/* While there are nodes to visit                          */
while |visited| ≠ N do
    /* find the edge (i,j) nearest/farthest to the tour  */
    for node k ∉ visited do
        for edge (i, j) ∈ tour do
            compute extra mileage
        end
    end
    Select node k with minimum extra-mileage
    Replace edge (i, j) with edges (i, k) and (k, j)
end
```

5.1.3 Implementation choices, Comparisons and Results

We also used two techniques:

- **GRASP** (greedy randomized adaptive search procedure): used in the nearest neighbor search, introducing randomization in the choice of the next node of the tour by choosing the second nearest node with probability of 10%:
- **Multistart**: initializes nearest neighbour search for each possible node and then returns the best solution.

For the Extra-Mileage algorithm we pick the 2 farthest nodes and than, at each iteration, the node that is nearest to the tour.

Table 5.1 and Fig. 5.1 show the comparison between different construction algorithms in terms of time spent to create the tour and the final cost of the solution. As you can see the best constructive heuristic is Extra Mileage. Greedy multistart algorithm is the following. Iterative grasp and the basic

greedy (nearest neighborhood) are equivalent in terms of solution quality. Basic grasp is the worst one. In terms of time computation, basic greedy and basic grasp are the fastest algorithms. Iterative Grasp works until the time limit is reached, while the other algorithms can stop when a good solution is found. In principle for all the methods but Iterative Grasp, the computation usually takes fraction of time of the time limit dedicated (apart obviously for small time limits or very large instances).

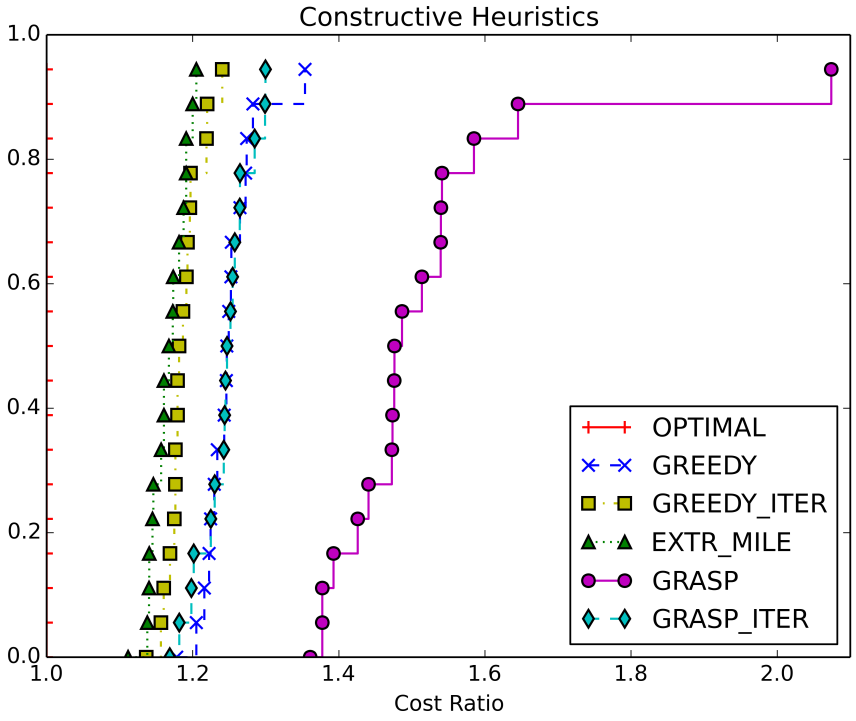

Figure 5.1: Constructive Heuristics comparison

5.2 Refinement Heuristics

This kind of heuristics start from a given solution and improves it making small changes. Their performances are strongly dependent by the construction heuristic used.

		GREEDY		GREEDY_ITER		EXTR_MILE		GRASP		GRASP_ITER	
Instance	**Real Cost**	**Cost**	**Time (s)**	**Cost**	**Time (s)**	**Cost**	**Time (s)**	**Cost**	**Time (s)**	**Cost**	**Time (s)**
ali535	202339	253362	0.03	**241072**	16.76	243778	15.99	298675	0.03	255853	300
p654	34643	43457	0.01	43027	5.24	**40123**	4.30	71935	0.01	45052	300
rat783	8806	11054	0.01	10540	10.45	**10399**	7.50	12910	0.01	11156	300
gr666	294358	366843	0.05	351041	31.11	**341997**	29.31	406054	0.05	365454	300
d657	48912	61627	0.01	60175	5.99	**57879**	4.73	75956	0.01	61885	300
pr439	107217	131281	0.005	**127230**	1.63	127715	1.28	176401	0.005	128470	300
rat575	6773	8605	0.01	**7993**	3.67	8079	2.88	9372	0.01	8483	300
u724	41910	52943	0.01	50802	7.62	**49944**	5.97	59319	0.01	53868	300
rd400	15281	19183	0.005	18431	1.29	**18187**	0.97	22671	0.005	19150	300
u574	36905	50459	0.01	45440	3.57	**43261**	2.88	59085	0.01	46661	300
vm1084	239297	301477	0.02	290806	26.14	**274081**	19.37	363967	0.02	320469	300
att532	27686	35516	0.01	33387	5.52	**33250**	4.45	42259	0.01	34341	300
d493	35002	41665	0.005	**40189**	2.34	40330	1.80	50435	0.005	41332	300
lin318	42029	54019	0.005	**49201**	0.61	49497	0.48	63037	0.005	50104	300
pr1002	259045	331103	0.02	313745	19.27	**302240**	15.18	391413	0.02	334032	300
gr431	171414	208932	0.02	204473	8.36	**197761**	8.07	238814	0.02	208304	300
pcb442	50778	61979	0.005	**58950**	1.70	61170	1.32	78570	0.005	63335	300
dsj1000	18659688	24631468	0.02	22450178	21.31	**21991699**	16.05	29570791	0.02	24677521	300

Table 5.1: Results of constructive heuristics

5.2.1 2-OPT

This heuristic proposed in 1958 by [8], first initializes the tour randomly or using some construction algorithm and then iteratively improves this tour by resolving crossing edges. This is done by selecting 2 edges that are crossing each other and then exchange them. More specifically, at each step it selects 2 edges (a, b) and (c, d) from the tour and crosses them obtaining the edges (a, c) and (b, d) exploiting the triangle inequality. Thanks to the triangle inequality it's possible to determine, when a solution contains two or more crossing edges, that this solution is not optimal.

Given two edges (a, b) and (c, d) where $a \neq c$, $a \neq d$ and $b \neq c$ the condition for exchanging two edges is the following:

$$\Delta(a, c) = c_{ac} + c_{bd} - c_{ab} - c_{cd} < 0 \tag{5.1}$$

where c_{ij} is the cost between node i and node j. When the condition 5.1 is satisfied, the edges (a, b) and (c, d) are replaced with (a, c) and (b, d) respectively, decreasing the total cost of the tour. The 2-opt move can be seen in fig. 5.2.

The 2-opt algorithm continues until there are no more crossing edges, i.e. when the condition 5.1 is not satisfied for all couple of edges, reaching a local optimum. The 2-OPT algorithm is shown in (Algorithm 7).

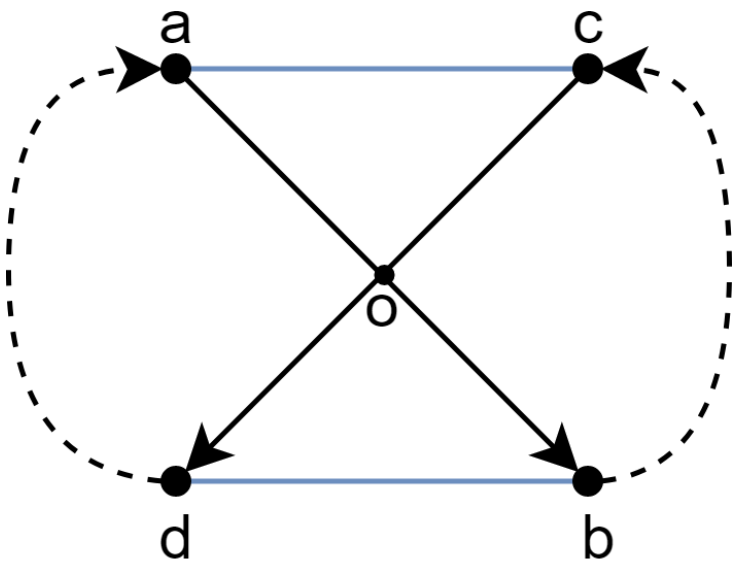

Figure 5.2: 2-Opt move: (a, b) and (c, d) are crossing, so replacing them with (a, c) and (b, d) decreases the tour length.

5.2.2 Comparison

We compared different versions of 2-opt refinement each with a different constructive heuristic as the initial solution. The results are shown in table 5.2 and Fig. 5.3. The best is greedy with multistart. This could be surprising since in comparison made with constructive heuristics in Fig. 5.1 the Extra Mileage was the winner. A reason on why applying 2-opt refinement the advantage of extra mileage is lost is due to the fact that extra mileage generates a solution with fewer crossing edges compared with greedy and grasp. In terms of quality of the solution and running time, the best one is 2-opt in combination with basic greedy. As you can see in table 5.2, the solving time is in the order of fractions of second for the majority of instances tested, making this algorithm suitable for larger instances.

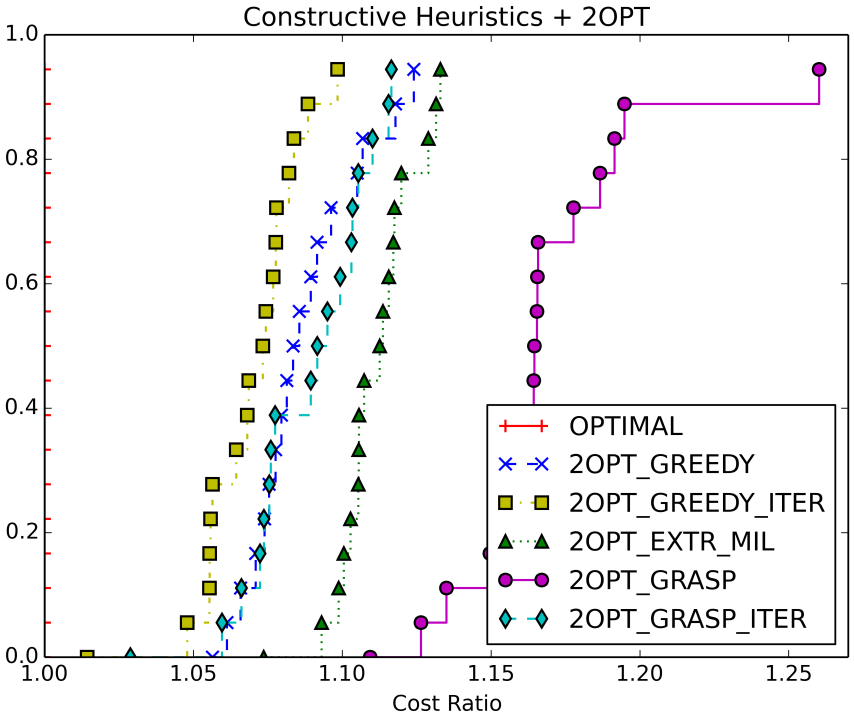

Figure 5.3: Constructive Heuristics + 2OPT comparison

Algorithm 7: 2-opt refinement

Input : a valid tour.
Output: a valid tour possibly of smaller length.

best_cost $\longleftarrow$ $cost$(tour)

`/* While we are within the time limit */`
while time_elaps <time_limit **do**
  `/* for each pair of subsequent nodes */`
  for *node* $a \in [0, n-2]$ **do**
    for *node* $c \in [a+1, n-1]$ **do**
      $b \longleftarrow succ(a)$ `/* successor of a */`
      $d \longleftarrow succ(c)$ `/* successor of c */`
      `/* skip non valid configurations */`
      if $b == d$ *or* $a == d$ *or* $b == c$ **then**
        continue
      end
      $\Delta(a, c) \longleftarrow (c_{ac} + c_{db}) - (c_{ab} + c_{dc})$
      if $\Delta(a, c) < 0$ **then**
        swap (a, b) with (a, c) and (c, d) with (b, d)
      end
    end
  end
  `/* If we couldn't improve the tour, stop */`
  if best_cost <= $cost$(tour) **then**
    stop
  end
  best_cost $\longleftarrow$ $cost$(tour)
end

		2OPT_GREEDY		2OPT_GREEDY_ITER		2OPT_EXTR_MIL		2OPT_GRASP		2OPT_GRASP_ITER	
Instance	**Real Cost**	**Cost**	**Time (s)**	**Cost**	**Time (s)**	**Cost**	**Time (s)**	**Cost**	**Time (s)**	**Cost**	**Time (s)**
ali535	202339	227440	0.90	**222260**	17.56	226579	15.84	238297	0.81	225908	120.75
p654	34643	36765	0.53	**35139**	5.70	38203	4.48	43658	0.29	35641	12.56
rat783	8806	9470	0.30	**9410**	10.45	9841	7.60	10166	0.36	9717	120.34
gr666	294358	325816	1.25	**318530**	31.85	328849	30.25	343098	1.03	328372	121.18
d657	48912	53098	0.28	**52241**	5.88	55216	4.61	56960	0.32	53284	120.25
pr439	107217	115748	0.14	114123	1.72	118724	1.39	124308	0.11	**113609**	120.11
rat575	6773	7324	0.22	**7155**	3.91	7543	3.01	7865	0.19	7263	120.19
u724	41910	45409	0.34	**45128**	7.83	47423	6.32	48312	0.31	44999	120.24
rd400	15281	16647	0.06	**16401**	1.35	16891	1.02	17814	0.10	16442	120.08
u574	36905	40282	0.19	**39647**	3.77	41171	3.00	43971	0.21	40285	120.22
vm1084	239297	**257862**	0.77	260470	27.51	262928	20.40	278616	0.78	263958	120.67
att532	27686	30594	0.28	**29836**	5.84	31368	4.76	31424	0.37	30317	120.19
d493	35002	**36975**	0.12	37728	2.50	38695	1.96	39428	0.16	37643	120.18
lin318	42029	45009	0.07	**44358**	0.68	45940	0.54	49872	0.07	45283	120.05
pr1002	259045	283967	0.59	**273491**	19.86	285094	16.41	301900	0.66	287580	120.68
gr431	171414	184075	0.59	**180915**	8.91	184026	9.29	204805	0.51	182742	120.48
pcb442	50778	54119	0.11	**53211**	1.86	56141	1.46	56331	0.13	55818	120.11
dsj1000	18659688	20858850	0.87	**20222830**	22.16	20760775	17.20	21450916	0.81	20625547	120.68

Table 5.2: Results of constructive heuristics + 2opt

Chapter 6

Metaheuristics

Metaheuristics are a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic optimization algorithms. Those algorithms can solve any type of optimization problem with only few adaptations. Even with a naive adaptation for a specific problem these algorithms can obtain a good solution for certain instances. So far, Metaheuristics for TSP should hopefully find better solutions than any 2-opt variation giving a good time limit. In general, the main idea of the majority of metaheuristic algorithms is to escape from a local minimum in order to find hopefully a better one.

In the following sections we are going to present three different metaheuristic algorithms:

- **Variable Neighborhood Search (VNS)** that randomly changes k different edges from the current solution and improves it by using 2-opt.
- **Tabu Search** that adds some edges in a tabu list which become forbidden for an improving move.
- **Genetic algorithm** that simulates Darwin's evolution theory which the objective is to generate the fittest population which may contain a good solution.

6.1 Variable Neighborhood Search (VNS)

Instead of running greedy + 2opt for multiple initial solutions, the VNS heuristic proposed by [9] is based on local search; it starts from an arbitrary solution, optimizes it reaching a local minimum and then, escapes this local minimum by moving to a neighbour solution in the hope to find a better solution. This process is shown in Fig. 6.1. VNS is based on 3 facts:

- A local minimum for a neighborhood structure is not necessarily the same for other neighborhood structures.
- A global minimum is also a local minimum for all possible neighborhood structures.
- In many problems a local minimum for one neighborhood structure is very close a local minimum of another neighborhood structure.

This means that with a different neighborhood structure we can reach a different local minimum; this comes the name of this algorithm, Variable Neighborhood Search. For the TSP we used Basic VNS that uses a predefined set of neighborhood structures. These neighborhoods, are usually of different sizes starting from the smallest one to a larger one when a better solution is not found.

VNS iteratively optimizes the current solution till a local minimum is found (intensification phase) that is done by 2-opt. Then the algorithm chooses randomly a solution in the neighborhood (diversification phase). If the current solution is better than the best found so far, it gets updated and in the next iteration the smallest neighborhood will be used otherwise a incrementally larger neighborhood will be used in the next iteration. The pseudocode for the VNS is shown in Algorithm 8.

We implemented just the 3-opt neighborhood, that means that during the kick operation we remove randomly 3 edges and reconnect them in a predefined way because we saw that the 2-opt neighborhood never lead to a better solution without increasing the neighborhood size.

In Fig. 6.2 we plotted the solutions cost through the iterations selecting a random solution in the 3-opt neighborhood has. As you can see the diversification phase worsen suddenly the current solution while the intensification phase improves it.

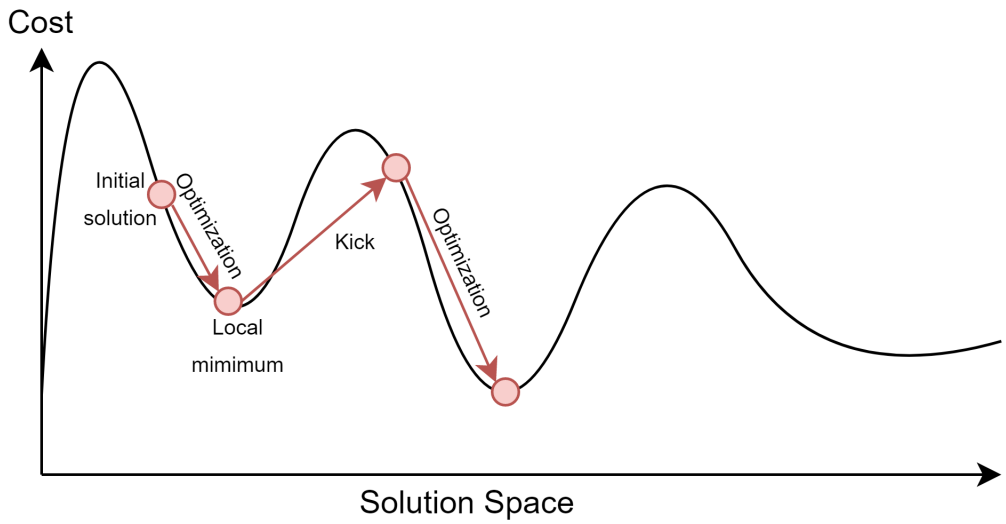

Figure 6.1: VNS starts from an initial solution and optimizes it reaching a local minimum. To escape this local minimum it applies a random kick and optimizes again in the hope to find a better solution.

6.2 Tabu search

As we know, local search algorithms can be stuck in a local minimum. Tabu search tries to avoid this by allowing a move that worsen the current solution when a move that can improve it cannot be found. However by doing this we can still be trapped in cycles because, when an improving move is applied, the solution goes back to the same local minimum. To avoid this situation, Tabu Search keeps a list called **tabu-list**, that contains moves that cannot be performed again. More specifically, when a move is performed is added to the tabu-list which is a FIFO queue with a specific size. The tabu-list size is called **tenure**. In this way a move won't remain illegal forever but just for a certain amount of iterations. The performance of this algorithm depends dramatically by the tenure of the tabu-list. If it is too small, the algorithm gets stuck in the local minimum because there is a sequence of moves that brings the solution back to the local minimum. Instead if the tenure is too large, the search is not effective because the neighborhood is too small. To overcome this, we can change the tenure dynamically during the execution.

We implemented 3 different policies to change dynamically the Tabu

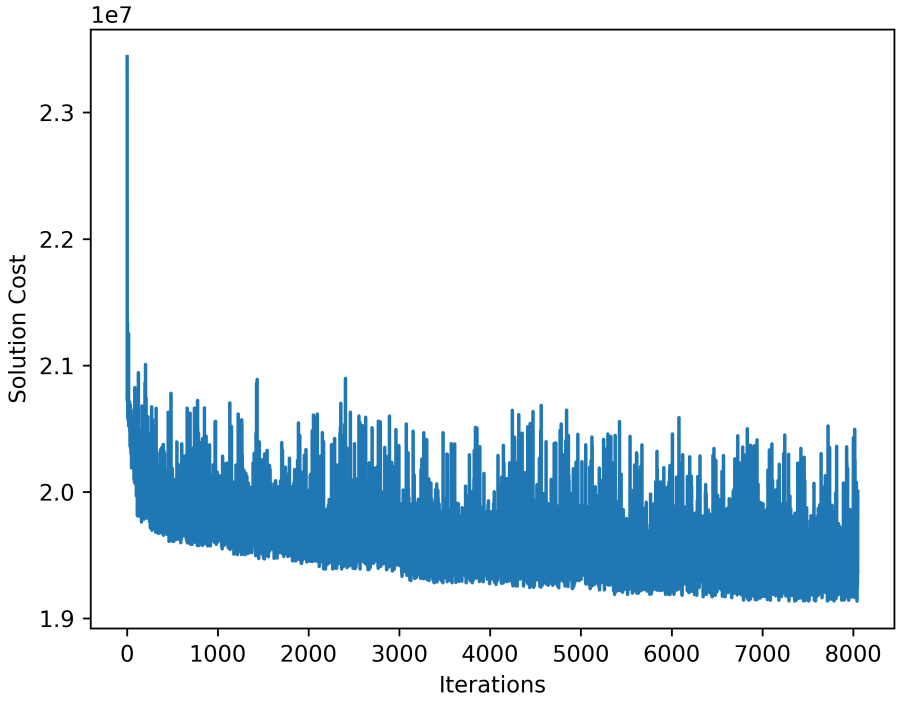

Figure 6.2: VNS solutions cost trough iterations over DSJ1000 tsp instance. A random solution in the 3-opt neighborhood is chosen and then optimized with 2-opt at each iteration.

tenure:

- **Step policy**: changes the tabu-list size every 100 iterations to a minimum or a maximum
- **Linear policy**: the tabu-list size grows of 1 unit for each iteration 'till reaching the maximum size, then decreases of 1 unit for each iteration 'till reaching the minimum size, and so on...
- **Random policy**: changes the tabu-list size every 100 iterations to a random size chosen in a range.

The tabu list is implemented in order to contain forbidden edges. At every iteration a 2-opt algorithm is computed taking in consideration the tabu

Algorithm 8: Basic VNS for TSP implementation

```
Input  : a tsp instance.
Output: a valid tour.

best_solution ⟵ greedy_2opt(tour)
k ⟵ 1

while time_remain < time_limit do
    curr_solution ⟵ best_solution
    curr_solution ⟵ kick(curr_solution,k)
    curr_solution ⟵ 2opt(curr_solution)
    if curr_solution cost <best_solution cost then
        best_solution ⟵ curr_solution
        k ⟵ 1
    end
    k ⟵ k + 1
end
return best_solution
```

list. When 2-opt encounters an edge which is tabu, it skips that edge. This ensures 2opt to explore other solutions rather than going back to the previous one. After the 2-opt optimization step, a worsening 2-opt move is done by taking two random edges that are not in tabu-list and are crossed. After that, these two edges are added in the tabu list. From time to time some edges are removed from tabu based on the current value of the tenure. For instance, given an edge $e \in E$ currently in tabu-list (i.e. $tabu[e] > 0$), this edge is removed from tabu-list if the following condition is satisfied:

$$curr\_iter - tabu[e] > tenure \tag{6.1}$$

where for each element in tabu-list, the iteration of when the edge e is added in tabu-list is saved. The tenure value changes during the execution alternating diversification and intensification phases. For example in the step policy, the tenure is updated every 100 iterations between min_tenure and max_tenure which are parameters chosen by design which are $num\_nodes * 0.02$ and $num\_nodes * 0.1$ respectively.

The Tabu-Search pseudocode is shown in (Algorithm 9).

Algorithm 9: Tabu Search for TSP implementation

Input : a tsp instance.
Output: a valid tour.

```
best_solution ⟵ greedy_2opt(tour)
curr_iter ⟵ 1
while time_remain < time_limit do
    curr_solution ⟵ best_solution
    curr_solution ⟵ 2opt(curr_solution, tabu_list, curr_iter, curr_tenure)
    if curr_solution cost <best_solution cost then
        best_solution ⟵ curr_solution
    end
    e1 ⟵ rand(E)
    e2 ⟵ rand(E)
    curr_solution ⟵ 2opt_move(e1, e2)
    tabu_list [e1] ⟵ curr_iter
    tabu_list [e2] ⟵ curr_iter
    curr_tenure ⟵ update_tenure(curr_tenure, curr_iter).
    curr_iter ⟵ curr_iter + 1
end
return best_solution
```

6.3 Genetic

In computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). 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 [10]. We represent a individual as a list of nodes that is the order in which the tour is visited. The steps executed by a GA are:

1. Create the initial population of N individuals (generation 0).

2. Compute the fitness value of each individual in the current population. For the TSP problem the fitness is the tour cost.

3. Selection: select 60% of individuals (parents). We used the Rank based roulette wheel selection proposed by [11].

4. Crossover: create a new individual (child) by merging 2 random parents from the selected part of the population. The merge operation works by pick randomly a subtour from the first parent and fill the empty cells with the nodes from the second one in order of appearance, without adding duplicate nodes. You can see this process in Fig. 6.4.

5. Mutation: mutate each child in order to avoid local convergence. In our case we apply a mutationt to an individual with 5% probability.

6. keep just the 1000 best individuals among the previous population and offsprings exploiting rank based roulette wheel selection.

7. Repeat steps 3-6 for G generations or until time limit is reached

In our implementation the initial population can be chosen to be initialized by completely random tours or by some constructive heuristics. For instance grasp is the most suitable for this since it exploits some randomness in selecting the nearest node letting the initial population have with high probability, different individuals in the population compared with extra mileage and greedy. Another design choice is the mutation probability for which an individual gets a mutation by inverting a subtour and the 2-opt mutation. We have tested three different versions of the genetic implementation:

- **Pure genetic**: Only genetic algorithm is involved without any other type of heuristics such as constructive or 2-opt.
- **Genetic with grasp initial population**: Applies genetic algorithm with an initial population generated by grasp heuristic.
- **Genetic with 2-opt mutation**: Genetic with initial population generated randomly with a low probability 2-opt mutation after crossover operation.

The results can be viewed in Fig. 6.3. As you can see, 2-opt mutation and grasp initial population lead to same performances. There's no clearly one method in advantage on the other while, the pure genetic obtains the worst results.

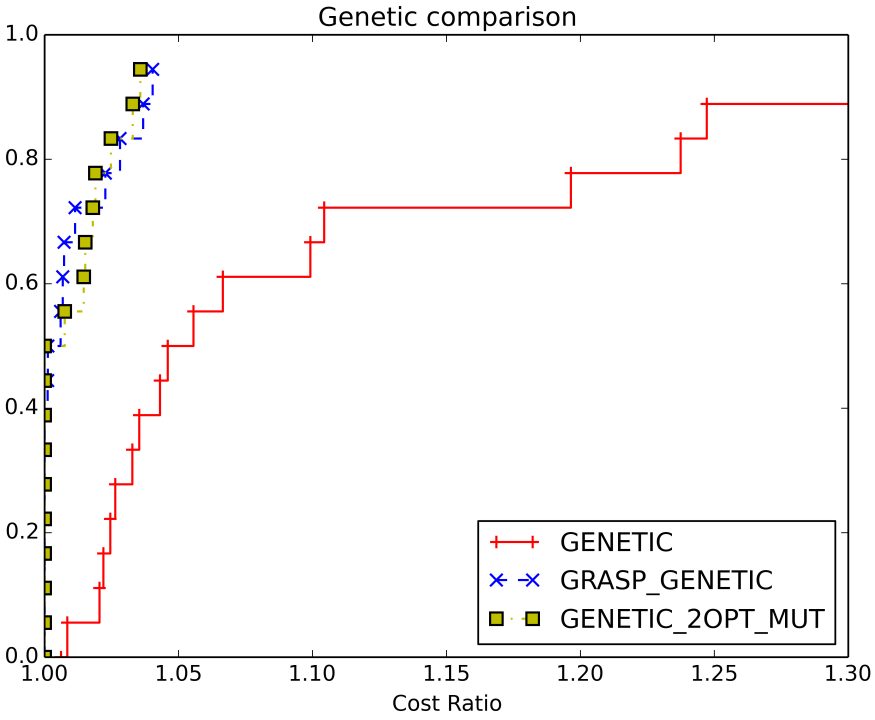

Figure 6.3: Comparison between pure genetic, genetic with grasp initial population and genetic with 2-opt mutation

The Genetic Algorithm pseudocode is shown in (Algorithm 10).
In Fig. 6.5 you can see the best solution cost over the generations.

6.4 Comparison between metaheuristics

The results of the Meta heuristics comparison are shown in Table 6.1 and in Fig. 6.6. As you can see the best is the VNS having the best results for every instance tested. The TABU variants almost perform the same to each other, while Genetic is the worst. The genetic algorithm tested here is the one with 2-opt mutation.

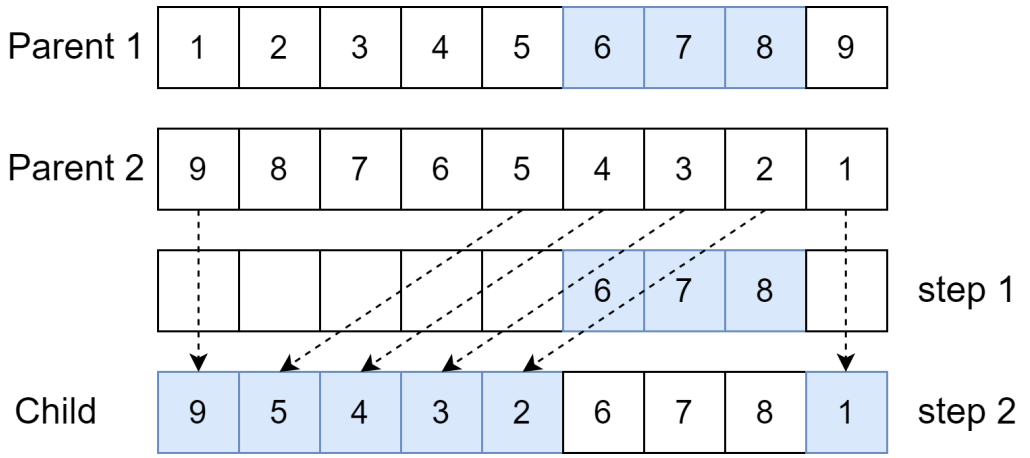

Figure 6.4: Crossover operation: in step 1 we pick a subtour of the first parent, and in step 2 we fill the blank cells with the nodes from the second parent in order of appearance without duplicates

Algorithm 10: Genetic algorithm for TSP implementation

Input : tsp instance.
Output: a valid tour.

population ⟵ {N random TSP tours}
compute the tour cost for each tour in population

while time_remain < time_limit **do**
  for *K times* **do**
    par1, par2 ⟵ select 2 tours from population with probability over their cost
    child ⟵ crossover between par1 and par2
    child ⟵ randomly swap some nodes of child
    child.cost ⟵ compute the tour cost for child
    add child to population
  end
  population ⟵ keep the N best tours
end
return best tour in population

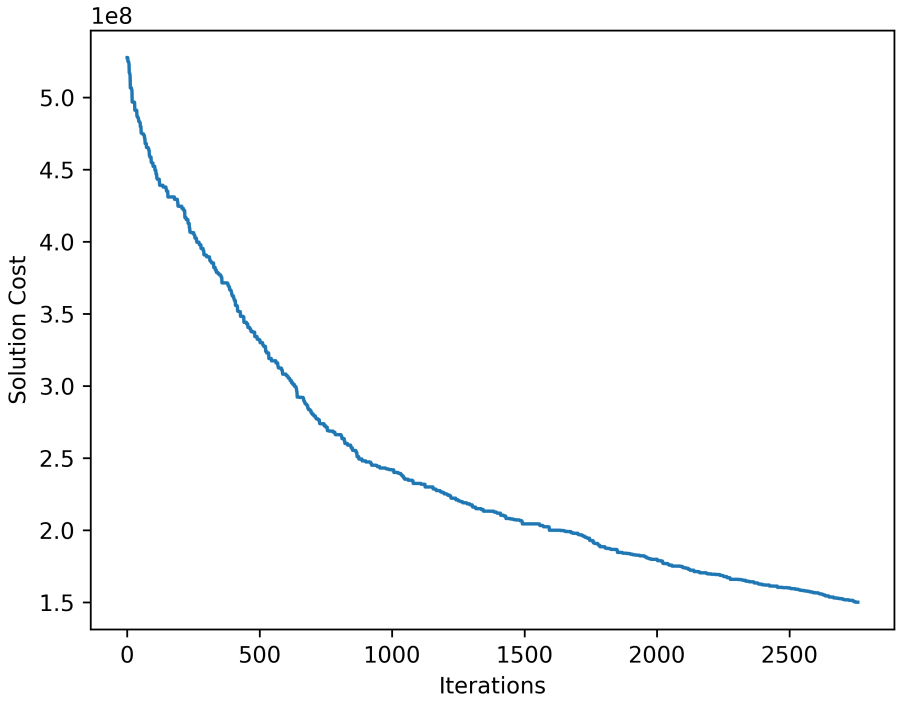

Figure 6.5: Genetic solutions cost trough iterations over DSJ1000 tsp instance

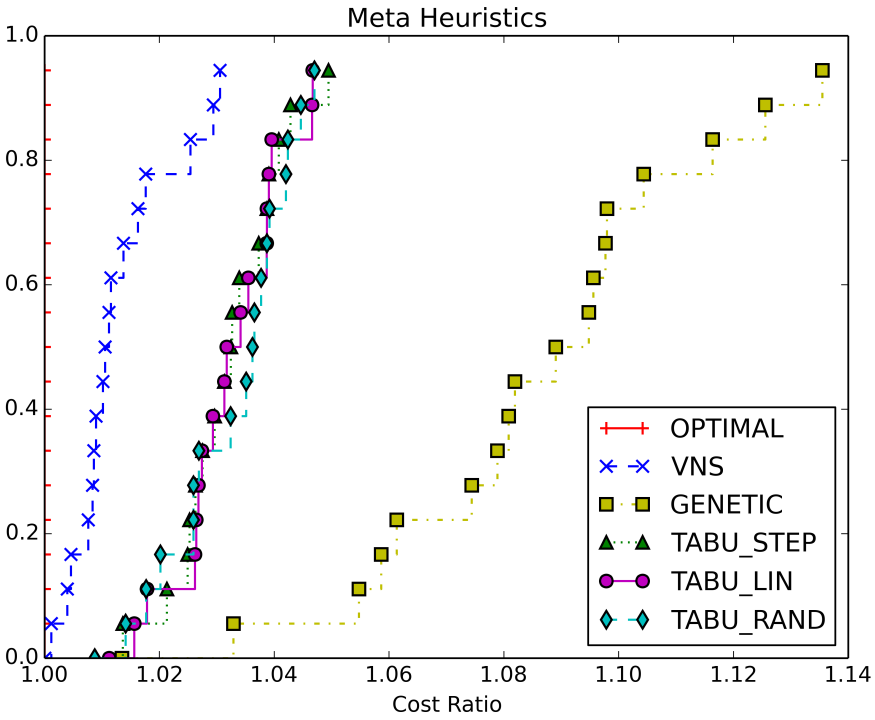

Figure 6.6: Comparison between meta heuristics methods each ran for 20 mins on each instance

		Models				
Instance	**Real Cost**	**VNS**	**GENETIC**	**TABU_STEP**	**TABU_LIN**	**TABU_RAND**
ali535	202339	**204605**	227742	207656	207640	210263
p654	34643	**34684**	35110	34954	35033	34946
rat783	8806	**8961**	9725	9150	9150	9147
gr666	294358	**298407**	334235	308915	308109	308194
d657	48912	**49332**	52919	50444	50464	50497
pr439	107217	**107223**	113796	108676	108891	108732
rat575	6773	**6804**	7376	7063	7035	7060
u724	41910	**42591**	45917	43331	43397	43442
rd400	15281	**15397**	16418	15667	15729	15677
u574	36905	**37280**	39816	38110	38165	38297
vm1084	239297	**246610**	267146	248219	248759	247958
att532	27686	**27917**	29924	28506	28417	28658
d493	35002	**35371**	36919	35965	35940	35909
lin318	42029	**42405**	43412	42925	42779	42773
pr1002	259045	**265627**	284358	269073	269073	269933
gr431	171414	**173396**	187667	176975	176781	176019
pcb442	50778	**50979**	53756	52043	52169	51802
dsj1000	18659688	**19208075**	20487747	19421019	19529106	19492379

Table 6.1: Results of meta heuristics ran for 1200 seconds

Chapter 7

Conclusions

In this work we implemented tested and analysed 29 algorithms to solve the Traveling salesman problem. Some of them, called exact algorithms, return the optimal solution while others, called heuristic algorithms, return an approximate solution for larger instances in a feasible amount of time. For each algorithm we have highlighted strengths and weaknesses over several tsp instances.

7.1 Compact methods

Compact methods are capable on finding an optimal solution by using MIP solver adding a polynomial number of constraints to the naive mathematical formulation of the TSP. For instance $O(n^2)$ constraints are added. The best compact model we found is the Gavish and Graves model that adds $O(n^2)$ new variables and $O(n^2)$ constraints. Testing on instances with up to 70 nodes, we have figured out that the major drawback of these models is that they require a long computational time. This makes them impractical to solve larger instances.

7.2 Exact methods

Exact models use the MIP optimizer to iteratively solve sub-problems leading to an optimal solution for the given TSP instance. According to our test, those models can solve bigger instances than the compact ones but are still impractical for instances with more than 1000 nodes. The test was conducted

utilizing instances between 150 and 750 nodes. The best performing exact model we found is the user-cut which utilizes a callback function to apply SECs in incumbent and relaxed solutions.

7.3 Heuristic methods

Most combinatorial optimization problems like the TSP are not solvable to optimum for very large instances. For this reason heuristic algorithms are used. These algorithms does not guarantee to find an optimal solution but can compute a good one in a small amount of time which make them suitable for large instances.

We firstly presented Matheuristics. Matheuristics use a MIP solver in a combination with some heuristic. The best one we found is Hard-Fixing that fixes iteratively some edges on the original problem and solves the new simplified one using MIP. The size of the tested instances goes from 300 to 1000 nodes.

The second category of heuristics we presented are Constructive-Heuristics. Constructive-Heuristics are heuristic algorithms which obtain a solution in a very small amount of time. Due to their solution quality, these algorithms are utilized to construct a good feasible initial solution to feed to a more powerful heuristic (usually refinement-heuristic). The best constructive algorithm which returns good solutions is the Extra Mileage algorithm although the greedy one gives acceptable solutions with a fraction of time needed by Extra Mileage.

The third type of heuristics presented is refinement-heuristic. Different versions of 2-opt algorithm were tested showing that with combination of multi start greedy algorithm, the 2-opt gives the best results.

We finally presented Meta-Heuristic algorithms which are problem independent heuristics. These algorithms try to find better local minimum in the neighboring solutions by worsening the current one with the hope to reach after some re-optimizations a better solution. We have seen that the best performing algorithm is VNS being the best one for all the instances tested in comparison with Tabu and Genetic. These algorithms are scalable and parallelizable these qualities make them suitable for larger instances rather than matheuristics.

Bibliography

[1] Cormen T., Leiserson C., Rivest R. and Stein C. *Introduction to Algorithms.* The MIT Press, Ed. 3, pages 1096-1097, 2009.

[2] Miller, C. E. and Tucker, A. W. and Zemlin, R. A. *Integer Programming Formulation of Traveling Salesman Problems.* Association for Computing Machinery, 1960

[3] Gavish Bezalel, Graves Stephen C. *The Travelling Salesman Problem and Related Problems.* MIT, 1978.

[4] *https://en.wikipedia.org/wiki/Heuristic*

[5] Fischetti, M., Lodi, A. *Local branching.* Math. Program., Ser. B 98, 23–47 (2003). https://doi.org/10.1007/s10107-003-0395-5

[6] TSPLIB95. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

[7] Concorde. https://www.math.uwaterloo.ca/tsp/concorde.html

[8] G. A. Croes, *A method for solving traveling salesman problems.* Operations Res. 6 (1958) , pp., 791-812.

[9] N. Mladenović, P. Hansen, *Variable neighborhood search.* Computers & Operations Research, Volume 24, Issue 11, 1997, Pages 1097-1100

[10] Mitchell, Melanie, *An Introduction to Genetic Algorithms*, 1996, MIT Press,Cambridge, MA, USA

[11] Razali, Noraini & Geraghty, John. *Genetic Algorithm Performance with Different Selection Strategies in Solving TSP.* 2011