UNIVERSITY OF PADUA

INFORMATION ENGINEERING DEPARTMENT (DEI)

MASTER'S DEGREE IN COMPUTER ENGINEERING

Study on algorithms to solve the Travelling Salesman Problem

Operations Research 2 course

Prof.
Matteo Fischetti

Students:
Daniela Cuza (2054562)
Giovanni Faldani (2054141)

Abstract

Among all the problems in the NP-hard set, the travelling salesman problem is the most popular one. The goal of the TSP is to find the minimum itinerant distance for a salesman that starts from a location and returns in the precise same location, by visiting all the cities exactly once. Innumerable applications in computer science and engineering fields can be expressed as TSPs.
In this study, we present various algorithms for solving the TSP, some provide an optimal solution (exact algorithms) and others give a near-optimal solution (stochastic algorithms). Furthermore, an extensive comparison of algorithms is provided, by discussing their quality and computing time.

Contents

Chapter 1

Introduction

Developing new ideas for the travelling salesman problem (TSP) and solving TSP's has always been one of the main focus of mathematical optimization. The TSP has not only captivated mathematicians, and economists, but also physicists, engineers, biologists, and chemists are fascinated with this problem[18]. It originates from the seven bridges problem of Königsberg, in which Euler, the famous mathematician, was asked to find a way for a parade in town to pass through every bridge no more than one time each.
The expression 'travelling salesman' was employed for the first time in a 1932 German book by a veteran travelling salesman called Karl Menger.[31] The TSP was then presented and formulated in 1948 by the RAND Corporation. Thanks to the Corporation's fame, to the novel discipline of linear programming and also to the trials to resolve combinatorial problems, the TSP became a widespread and famous problem.
The travelling salesman problem can be applied to various real-world applications, such as computer wiring, vehicle routing, clustering a data array and job-shop scheduling with no intermediate storage [33]. The aforementioned applications are special cases of TSP, although apparently they can be considered unrelated. Furthermore, formulating these problems as TSP is the most trivial way to deal with them.
The TSP is one of the most common examples of NP-hard graph theory problems, in fact there is no polynomial time algorithm to solve it. Formally, the TSP is given as an optimization problem in which we must visit every node in a complete graph only once, with exactly two edges connected to each node (commonly called a Hamiltonian circuit). Every edge is defined by the two nodes it is connected to, e.g. the edge (i, j) is the one connecting node i to node j.
There is a numerical "cost" (geometric distance in our case) associated with every single edge, and the objective of the optimization is to find the Hamiltonian circuit which minimizes the overall sum of the costs of all selected edges.
Finding the solution to a TSP is a very difficult task, mainly due to the large number of possible tours: $(n-1)!/2$, where n is the number of nodes [31]. On the other hand, formalizing and describing a TSP is trivial, which is why it has been a central focus of researchers over the years.
In this project, we will be describing and comparing various known methods and algorithms used to compute solutions to solving the symmetric TSP, a variant of the problem focused only on undirected graphs.
The rest of the thesis is organized as follows: in section 2 we formalize our problem, setup details are described in section 3, in section 4 all the details of our methods and

algorithms are provided, in section 5 we compare the algorithms implemented in terms of computation time and quality and finally section 6 includes all the conclusions.

Chapter 2

Formalization of the Problem

2.1 Graph Model

Let $G = (V, E)$ be a given undirected graph, where V is a finite set of vertices and the elements of E are edges. Consider the cost function $c : E \to \mathbb{R}$, we will indicate by c_e the cost of edge $e = (i, j)$ and we will consider all the costs as non negative.

`Istances:`	$< G = (V, E), c >$
`Solutions:`	$\{G' = (V, E') : E' \subseteq E \wedge E'$describes a tour in $G\}$
`Objective:`	Find $G' = (V, E')$ that minimizes $\sum_{e \in E'} c_e$

2.2 Linear Programming Model

To represent the TSP in terms of a linear programming model, we need to formulate the expressions above in terms of variables and linear constraints. Regarding the variables, $0 - 1$ variables are used and they are represented as x_e. Given an edge $e = (i, j) \in E$, x_e indicates whatever the edge e is selected or not in $E' \subseteq E$.
Formally, the decision variables are defined as:

$$x_e = \begin{cases} 1, & \text{if } e \in E' \\ 0, & \text{otherwise} \end{cases}$$

To better understand the formulation of the ILP model, is useful to provide some definitions:

$$E(S) = \{\{i, j\} \ \in E : i \in S \wedge j \in S\}$$

$$\delta(S) = \{\{i, j\} \ \in E : i \in S \veebar j \in S\}$$

$$\delta(v) = \{\{i, j\} \ \in E : v \in \{i, j\}\}$$

The subsequent model is obtained:

$$\begin{cases} \min \sum_{e \in E} c_e x_e \\ \sum_{e \in \delta(v)} x_e = 2, \forall v \in V \\ \sum_{e \in E(S)} x_e \leq |S| - 1, \forall S \subset V : |S| \geq 3 \\ x_e \in \{0, 1\}, \forall e \in E \end{cases}$$

The model is composed by the objective function and by three families of constraints, where the first one indicates that the number of edges incident on a vertex v must be equal to 2, e.g. the degree of each vertex needs to be 2. The second one guarantees the absences of subtours, in fact the solution must contain a single tour. Finally, we need to consider the integrality constraints.

Chapter 3

Setup Description

3.1 Introduction

For the purposes of our study, we will be writing C code to strike the best compromise between computational efficiency and code readability. We will be needing to use functions from the IBM CPLEX [1] optimizer library, so our development environment of choice will be Microsoft Visual Studio 2022, which allows for simple inclusion of the CPLEX header files via the project settings. For the exact details of the setup, we followed the same steps as presented in the tutorial [2] on professor Fischetti's website. To plot the solutions we used the latest gnuplot [3] distribution available at the moment.
After setting up the environment, we followed a template [4] for a program meant to solve the Vehicular Routing Problem, which we used as a base to build our own program, mostly importing basic functions like command line argument parsing or file reading and writing.

3.2 CPLEX

In order to solve the travelling salesman problem, we can make use of an extraordinary optimization program: IBM ILOG CPLEX. Its effectiveness and advantage have been shown extensively through varied applications in thousands of commercial installations. This program, introduced and distributed by IBM, exploits advanced mathematical programming technology. In the beginning it was created only for the C language, however today it is also possible to use it with other programming languages, such as Python and Java. Moreover, IBM ILOG CPLEX can be used with different OS: Windows, Linux, UNIX and MAC OS. Notice that it is available for free for students: you need to download the academic version and login with an institutional account.
We report below the installation details.

3.2.1 Installation on Windows

Firstly, you need to start the IBM ILOG CPLEX Optimization Studio installer by double clicking the executable in Windows Explorer. Make sure that you have the Administrator privileges to install in the default location. The installer conducts you through the options for installation. On Windows, the default installation location of IBM ILOG

CPLEX Optimization Studio is:
$C : \backslash ProgramFiles \backslash IBM \backslash ILOG \backslash CPLEX\_Studio[edition][version]$
Note that, it may happen that during the installation you have chosen a distinct installation directory ($[install\_dir]$).
You can start the IDE using a Start Menu entry: $Start > AllPrograms > IBMILOG > CPLEXOptimizationStudio[edition][version] > CPLEXStudioIDE$.

3.2.2 Installation on MAC OS

You need to:

1. Double-click to unzip the $cos\_installer < version > -osx.zip$
2. Double-click the $cplex\_studio < version > -osp.app$ and the installer conducts you through the installation options.

On Mac, the default installation location of IBM ILOG CPLEX Optimization Studio is:
$/Applications/CPLEX\_Studio[edition][]version]$
You have two options to start the IDE:

- opening launchpad and clicking the oplide icon in the "$CPLEX\_Studio < version >$ " app.
- launching the oplide.app from the $/Applications/CPLEX\_Studio < version > /opl/oplidefolder$.

3.2.3 Installation on UNIX, Linux

The UNIX/Linux installer is an executable file. To start the installation make sure that you have the Administrator privileges. You can execute the file from the command line. You can give the command $./ < installname > .bin$, where $< installname >$ refers to the installer name. In order to issue a file execute permission, give the command $chmodu + x < installname > .bin$.
On UNIX, the default installation location of IBM ILOG CPLEX Optimization Studio is:
$/opt/IBM/ILOG/CPLEX\_Studio[edition][version]$
On Linux, the default installation location of IBM ILOG CPLEX Optimization Studio is:
$/opt/ibm/ILOG/CPLEX\_Studio[edition][version]$
Note that, it may happen that during the installation you have chosen a distinct installation directory ($[install\_dir]$).

To start the IDE you can use the LINUX terminal.

1. Start a terminal Window.
2. Modify the path as $[install\_dir]/opl/oplide$.
3. Issue the command ./oplide.

Or, from any terminal location, insert the absolute path:
$[install\_dir]/opl/oplide/oplide$.

Chapter 4

Optimization Algorithms

The optimization algorithms can mainly be divided into two groups: deterministic algorithms and stochastic algorithms [19].

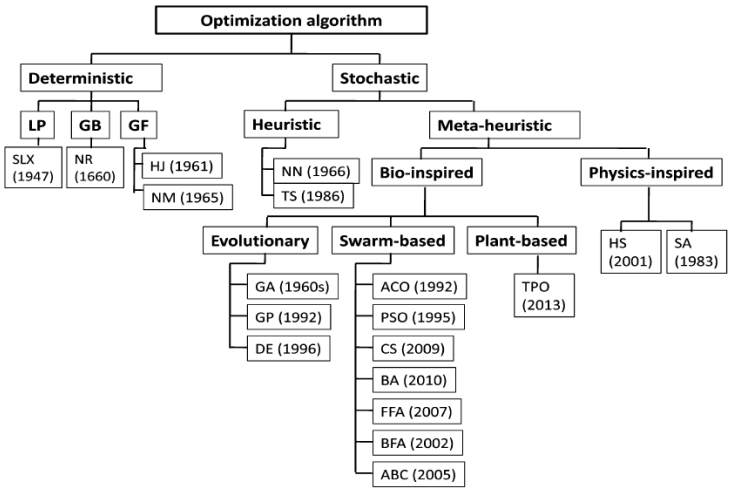


Figure 4.1: Taxonomy of optimization algorithm.

LP	Linear programming	NM	Nelder–Mead	NN	Nearest neighbor	ACO	Ant colony optimization
GB	Gradient-based	HS	Harmony search	HJ	Hooke–Jeeves	ABC	Artificial bee colony
GF	Gradient free	CS	Cuckoo search	TS	Tabu search	TPO	Tree physiology optimization
SLX	Simplex algorithm	BA	Bat algorithm	GA	Genetic algorithm	PSO	Particle swarm optimization
NR	Newton–Raphson	FFA	Firefly algorithm	GP	Genetic programming	SA	Simulated annealing
				DE	Differential evolution	BFA	Bacterial foraging algorithm

Figure 4.2: List of the name of each algorithm.

4.1 Stochastic Algorithms

The stochastic algorithms, given a certain input, can provide different outputs, because of the random component. Thanks to this peculiar property, stochastic algorithms are a widespread and crucial tool in optimization problems. Stochastic algorithms are additionally divided in two groups: heuristic and meta-heuristic algorithms[19].

4.1.1 Heuristics

Solving a travelling salesman problem optimally can be a very long procedure. Alternatively, the heuristic algorithms, also known as approximation algorithms, are a good choice in terms of computational time.
Heuristic algorithms consist in obtaining the solution by 'trial and error'[19]
With this procedure we are not certain to find the global optimal solution, in fact we obtain a near-optimal solution and a guarantee of how bad our solution can be [36].
Here, we discuss mainly three heuristic algorithms. For instance, nearest neighborhood, extra mileage and the 2-optimality move.

Nearest Neighborhood Algorithm

The Nearest Neighborhood algorithm represents one of the easiest and most basic ways to solve the TSP by using heuristics. It was first introduced by Karg and Thompson [27]. The algorithm, based on a greedy procedure, goes as follows:

1. Select the starting node.
2. Find the nearest uncovered node and visit it.
3. If there are other nodes still to be visited repeat step 2.
4. Return to the first node [36].

An alternative version of the nearest neighborhood is the GRASP one, it differs from the greedy approach by a degree of randomization in the choice of nodes. In particular, in the GRASP method we consider the second best choice with a given probability p. When p is set to 0, the grasp approach degrades into the greedy one.
There are mainly three possibilities to choose the starting node:

- select node 0
- select a random node
- try all nodes and choose the one that provides the better solution (multistart approach)

The pseudocode in 1, represents the grasp approach of the nearest neighborhood algorithm. The solution is stored in the successor array and the starting node is chosen by the user. In output the algorithm provides the cost of the solution. The computational complexity is $O(n^2)$ in the number of nodes n.

Algorithm 1 Nearest Neighborhood Algorithm

Input: successor array; startnode; prob

Output: cost of the solution found by the algorithm

```
1: n_conv ← nnodes;
2: for i=0 to n_uncov do
3:     uncov[i] ← i;
4: end for
5: current_node ← startnode
6: update n_uncov and uncov
7: for i=1 to nnodes-1 do
8:     min_dist ← INF;
9:     index_best ← −1;
10:    min_dist2 ← INF;
11:    index_best2 ← −1;
12:    for j=0 to n_uncov do
13:        new_dist ← cost (current_node, uncov [j]) ;
14:        if new_dist < min_dist and new_dist < min_dist2 then
15:            update min_dist, index_best, min_dist2, index_best2
16:        else if new_dist > min_dist and new_dist < min_dist2 then
17:            update min_dist2, index_best2
18:        end if
19:    end for
20:    if random01() < prob then
21:        update successor and uncov with the second best choice
22:    else
23:        update successor and uncov with the best choice
24:    end if
25:    update current_node
26: end for
27: compute the cost of the solution
28: return cost;
```

Insertion Heuristic

Insertion heuristics were first studied and presented by Rosenkrantz et al. for the travelling salesman problem [38]. It has been demonstrated that they are efficient and popular methods also for different applications such as vehicle routing and scheduling problems [10]. Insertion heuristics are mainly famous because they are easy to implement, considerably fast and provide good solutions. Furthermore, they can be easily expanded to deal with complex constraints. Different versions of insertion heuristics were introduced, in this thesis we present the extra mileage one. The steps of the greedy approach are:

1. Design an initial sub-tour.
2. For each node h not in the solution yet, search the node i such that $[i, h] + c[h, succ[i]] - c[i, succ[i]]$ is minimal
3. Insert node h in the sub-tour between nodes i and $succ[i]$.

4. If there are other nodes still to be added to the tour repeat step 2.

In our code, the initial sub-tour is composed by two nodes and the edges that connect them. There are principally three alternatives to select the initial nodes:

- select the farthest nodes A and B
- select the nodes A and B at random
- compute the solution for every possible starting node and the farthest one from it and choose the one that provides the better solution (multistart approach)

The pseudocode 2 corresponds to the grasp approach of extra mileage, given a partial tour. The solution is stored in the successor array. In output the algorithm gives the cost of the solution. The computational complexity is $O(n^3)$ in the number of nodes n.

Algorithm 2 Insertion Heuristics

Input: successor array; uncov array; n_uncov; prob
Output: cost of the solution found by the algorithm

```
1: while n_uncov > 0 do
2:     delta ← INF;
3:     delta2 ← INF;
4:     h_best ← −1;
5:     i_best ← −1;
6:     h_best2 ← −1;
7:     i_best2 ← −1;
8:     for h=1 to uncov do
9:         for i=1 to nnodes do
10:            if succ[i] isuncovered then
11:                continue;
12:            else
13:                new_delta ← cost[i, h] + cost[h, succ[i]] − cost[i, succ[i]];
14:                if new_delta < delta and new_delta < delta2 then
15:                    update delta, h_best, i_best, delta2, h_best2, i_best2
16:                else if new_delta > delta and new_delta < delta2 then
17:                    update delta2, h_best2, i_best2
18:                end if
19:            end if
20:        end for
21:    end for
22:    if random01() < prob then
23:        update successor and uncov with the second best choice
24:    else
25:        update successor and uncov with the best choice
26:    end if
27: end while
28: compute the cost of the solution
29: return cost;
```

2-optimality move

One way to improve the quality of the solution provided by a certain algorithm, is to apply the 2-optimality move. The idea behind the 2-optimality move is very simple, 2 edges are removed from the tour and then the two paths are reconnected, in order to get a valid tour. This is a clever approach only if the difference between the new cost and the old cost is negative, so when the current solution can be improved. Generally, 2-optimality move is applied until no other improvements can be obtained, in which case the tour is defined as 2-optimal. In our code, for each pair (a, b) we compute $\Delta cost(a, b) = new\_cost - old\_cost$, and then we choose the pair (a, b) associated with the minimum $\Delta cost(a, b)$. Let b' be the successor of b and a' the successor of a in the old tour. The updating of the tour consists of three steps:

1. invert the path from a' to b
2. set the successor of a as b
3. set the successor of a' as b'

Another possibility to enhance a tour consists in computing a 3-optimality move. Such an approach is very close to the 2-optimality move, but in this case three edges are eliminated and there exist two possibilities to reconnect the 3 paths to obtain a valid tour. In general, a 3-opt tour is as well 2-opt [22].

4.1.2 Metaheuristics

A Meta-heuristic algorithm can be seen as an evolution of an heuristic algorithm, it consists in the combination of the random component with local search[19].

Tabu Search

Tabu Search was first presented in 1986 at the University of Colorodo by Dr Fred Glover [14]. In the years since, researchers have spent a lot of energy better formalizing this approach [15] [42] [16]. Various computational experiments have shown that Tabu Search is a metaheuristic approach that can challenge virtually all popular methods and that, due to its adaptability, can prevail over many known procedures.[23].

It consists in an iterative search to obtain better tours and avoid local minima. [29]. We implement a distinct tabu search version which employs 2-edge exchange. A 2-edge exchange eliminates from the current solution 2 edges and substitutes them with 2 different ones, achieving a new feasible solution. It is the same method used also by 2-opt, but in this scenario we have a more elaborate supervisory control process.
Tabu search makes use of a tabu list, consisting of all the nodes that can not be used to move to a near solution. In our code, the parameter tenure controls the length of the tabu list. The tenure is not constant, but oscillates between two values. When the value of tenure is the minimum one (in our code corresponds to 10), we are in the intensification phase. The intensification phase consists in a refinement of the solution via 2-optimality. On the other hand, when the value of the tenure is the maximum one, we are in the diversification phase. As the name suggests, in the diversification phase the solution is

Algorithm 3 2-optimality move

Input: successor array

Output: return 1 if a move was performed successfully

```
1: delta ← INF;
2: best_a ← −1;
3: best_b ← −1;
4: for a=1 to nnodes do
5:     for b=a+1 to nnodes do
6:         temp_cost ← (cost[a, b] + cost[succ[a], succ[b]]) − (cost[a, succ[a]] + cost[b, succ[b]]);
7:         if temp_cost < delta and temp_cost < 0 then
8:             delta ← temp_cost;
9:             best_a ← a;
10:            best_b ← b;
11:        end if
12:    end for
13: end for
14: a_prime ← succ[best_a];
15: b_prime ← succ[best_b];
16: if delta > INF/2 then
17:    return 0; // no crossing pairs to remove
18: end if
    //invert the path from a_prime to b
19: current ← a_prime;
20: next ← succ[a_prime];
21: while current ≠ best_b do
22:    temp ← succ[next[];
23:    succ[next] ← current;
24:    current ← next;
25:    next ← temp;
26: end while
    //change the successor of a and a_prime
27: succ[best_a] ← best_b;
28: succ[a_prime] ← b_prime;
    return 1; // two optimality move executed successfully
```

modified as much as possible to escape any local minimum. The concept of intensification and diversification is blended with the analogy of natural processes such as biology, chemistry, physics or environment.

Variable Neighborhood Search

The Variable Neighborhood Serch metaheuristic was first introduced in 1997 by Mladenović and Hansen [35]. The main purpose of VNS is to avoid local optimal solutions by using the concept of neighborhood change [20]. In literature, various applications in which

Algorithm 4 Tabu Search

Input: initial solution stored in *succ*, *tenure_shift_frequency*
Output: cost of the solution found by the algorithm

```
1: //initialization of tenure
2: min_tenure ← 10;
3: if 100 < nnodes/10 then
4:     max_tenure ← 100;
5: else
6:     max_tenure ← nnodes/10;
7: end if
8: //inizialization of tabu list
9: for i=0 to nnodes-1 do
10:    tabu_list[i] ← INF;
11: end for
12: curr_iter ← 1;
13: tenure ← max_tenure;
14: while current_time < time_limit do
15:    if curr_iter%tenure_shift_frequency = 0 then
16:        if tenure = max_tenure then
17:            tenure ← min_tenure;
18:        else
19:            tenure ← max_tenure;
20:        end if
21:    end if
22:    2_optimality_move(succ,tenure,tabu_list, curr_iter)
23:    compute the cost of the solution
24:    update the solution if improved
25:    curr_iter ← curr_iter + 1
26: end while
27: compute the cost of the solution
28: return cost;
```

VNS demostrate great results are described. For instance, exact solution of large-scale location problems by primal–dual VNS; creation of feasible solutions to large mixed integer linear programs by hybridization of VNS and local branching; creation of excellent feasible solutions to continuous nonlinear programs; creation of feasible solutions and/or improved local optima for mixed integer nonlinear programs by hybridization of sequential quadratic programming and branch and bound within a VNS framework; investigation of graph theory to discover conjectures[21]. Similarly to Tabu Search, VNS is composed of two phases, an intensification one and a diversification one. In our project, the intensification phase is based on the application of 2-optimality move several times, until no more improvements can be obtained. Regarding the diversification phase, the solution is perturbed by using a 5-opt kick. In particular, five edges at random are removed, the path is then reconnected obtaining still a valid solution.

Algorithm 5 VNS

Input: initial solution stored in *succ*, *nnodes*
Output: cost of the solution found by the algorithm

```
1: while current_time < time_limit do
2:     //intensification phase
3:     apply various times 2-optimality move, until no other improvements are possible
4:     update incumbent
5:     //diversification phase
6:     // convert succ into path
7:     k ← 0;
8:     path[0] ← 0
9:     for i=1 to nnodes-1 do
10:        path[i] ← succ[k];
11:        k ← succ[k];
12:    end for
13:    // chose at random the index of five nodes
14:    i ← 0;
15:    while i < 5 do
16:        next = random01() * (nnodes-1)
17:        if next is not equal to a node already chosen then
18:            node[i] ← next;
19:            i ← i + 1;
20:        end if
21:    end while
22:    sort the array node
23:    //substitute the index with the node
24:    for j=0 to 4 do
25:        node[j] ← path[node[j]];
26:    end for
27:    //reconnect the tour
28:    temp ← succ[node[0]]
29:    for j=0 to 3 do
30:        succ[node[j]] ← succ[node[j + 1]];
31:    end for
32:    succ[node[4]] ← temp;
33: end while
34: compute the cost of the solution
35: return cost;
```

Simulated Annealing

The notion of annealing was developed for the first time in combinatorial optimization by Kirkpatrick et al. and Cerny. [7].
Simulated Annealing Algorithm is known as the oldest metaheuristics and is characterized by an uncommon explicit technique to escape from local minima [43]. Its main idea is to adopt a worse solution than the current one, with the aim of avoiding local minima. The

SA algorithm begins with an original solution x and creates a possible solution y from the neighborhood of x. In order to establish if the solution y is approved, the metropolis acceptance criterion is employed [34]. The candidate solution y is adopted in place of the current x based on the subsequent probability:

$$p = \begin{cases} 1, & \text{if } f(y) \leq f(x) \\ e^{-\frac{(f(y)-f(x))/scaler}{t}}, & \text{otherwise} \end{cases}$$

where t is the temperature (it starts at maximum value and decreases with increasing iterations), and in our project we use as *scaler* the average edge cost of a TSP heuristic solution with 1000 nodes.
In an attempt of using the SA algorithm on a real problem, we need to detail the neighborhood structure and cooling schedule. The cooling schedule includes the initial temperature, temperature decrement function, Markon chain lenght and termination condition. Unluckily, there is no choice and relative parameter that will work well for every application.
The SA algorithm was extensively and profoundly inspected on TSP problems. A convenient approach to generate the candidate solution y in TSP problems, also used in this work, is to employ 2-opt between two random nodes.
The study conducted by Skiscim et al. [40] shows that the performance obtained by using Simulated Annealing for travelling salesman problem is quite bad with respect to other well-known heuristics.

Genetic Search

Genetic Algorithms were initially developed by Holland in 1975 [24]. A genetic algorithm derives from the theory of natural evolution, first presented by Charles Darwin. In particular, it is based on the process of natural selection, in which the most qualified individual is chosen for reproduction, in order to create children of the next generation. Basically, there are five steps that we need to examine in a genetic algorithm:

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

Initial population

The algorithm starts with various solutions of our problem, this form the initial population. A solution can be associated with a chromosome, composed of different genes.
In our code, the size of the initial population is 1000 and we create the individuals by using the nearest neighborhood algorithm.

Algorithm 6 Simulated Annealing

Input: initial solution stored in succ, Tmax
Output: cost of the solution found by the algorithm

```
1: scaler ← 25;
2: T ← Tmax;
3: best_cost ← INF;
4: // create the initial solution by using multistart Grasp Nearest Neighbor
5: while current_time < time_limit do
6:     // update temperature
7:     T ← Tmax * (time_limit − current_time);
8:     //execute 2opt between 2 random nodes
9:     a = random();
10:    b = random();
11:    while a == b do
12:        b = random();
13:    end while
14:    delta_cost ← (cost[a, b] + cost[succ[a], succ[b]]) − (cost[a, succ[a]] + cost[b, succ[b]]);
15:    a_prime = succ[a];
16:    b_prime = succ[b];
17:    if delta_cost < 0 then
18:        invert the path from a prime to b
19:        succ[a] ← b;
20:        succ[a_prime] ← b_prime;
21:    else
22:        p ← exp(−delta_cost/(scaler * T))
23:        if random() < p then
24:            invert the path from a prime to b
25:            succ[a] ← b;
26:            succ[a_prime] ← b_prime;
27:        end if
28:    end if
29:    compute current_cost
30:    if current_cost < best_cost then
31:        update solution and the cost
32:    end if
33: end while
34: return best_cost;
```

Fitness function

A fitness score is associated with each individual, it reflects the probability that an individual will be chosen for reproduction. Therefore, the fitness score gives an idea on how good a solution is.
In this project, the fitness is related to a penalty cost. The penalty cost of a child is given by the sum between the cost of the solution and a big penalty for infeasibility.

Selection

In the selection phase, the goal is to choose the more suitable individuals, in order to transfer their genes to the next generation. The selection of individuals is determined by the fitness score, the higher an individual's fitness score, the higher his likelihood of being selected.

Crossover

The crossover phase is the most important one in the genetic algorithm. In our code, we take two existing solutions (parents), select a crossover point at random and splice together their path through the nodes, to obtain the child. After that, we remove double visits and repair the child into a feasible solution with the Extra Mileage heuristic. We can then improve our new population by using 2-opt move. In this project, we create 400 children by using crossover.

Algorithm 7 Crossover

Input: parent_1, parent_2, nnodes
Output: child

```
1: position = random()
2: //create child : merge the two parents at the chosen random position
3: for i=0 to position_1 do
4:     child[i] ← parent_1[i]
5: end for
6: for i=pos to nnodes-1 do
7:     child[i] ← parent_2[i]
8: end for
9: //repair the solution by using grasp extra mileage heuristic
10: return child;
```

Mutation

The mutation is useful to obtain diversity in the population and avoid premature convergence. We take only one existing solution and swap around nodes in different positions of the array representing the path through all the nodes. This method doesn't need to be repaired as the resulting solution is always feasible. In this project, we create 100 children by using mutation.

In [25] the authors propose a parallel genetic algorithm (GA) with edge assembly crossover (EAX) for TSP. GA-EAX is composed of two stages, namely GA-EAX/Stage1 and GA-EAX/Stage2. GA-EAX/Stage1 uses a crossover operator known as EAX-Single with the aim of enhancing population members by preserving the population diversity as large as possible. EAX-Single exchanges edges locally among two solutions. Alternatively, GA-EAX/Stage2 uses a crossover operator known as EAX-Block2 with the aim of enhancing the population members achieved after invoking GAEAX/ Stage1. EAX-Block2

exchanges edges globally among two solutions. GA-EAX approach shows very promising results but the computational complexity is quite high.

Algorithm 8 Mutation

Input: parent, n (number of nodes to mutate)
Output: child

```
1: child ← parent
2: for i=0 to n-1 do
3:     h = random()
4:     k = random()
5:     temp ← child[k]
6:     child[k] ← child[h]
7:     child[h] ← temp
8: end for
9: return child;
```

Algorithm 9 Genetic Search

Input: size of the initial population *num_sol*
Output: cost of the champion found by the algorithm

```
1: create the initial population
2: search the champion of the initial population
3: while current_time < time_limit do
4:     //crossover phase
5:     num_child ← 0;
6:     while num_child < num_sol * 0.4 do
7:         choose parent_1 at random;
8:         choose parent_2 at random;
9:         if parent_1 ≠ parent_2 then
10:            children[num_child] = crossover(parent_1, parent_2, nnodes);
11:            num_child + +;
12:        else
13:            continue;
14:        end if
15:    end while
16:    // mutation phase
17:    while num_child < num_sol * 0.5 do
18:        choose parent at random
19:        choose the number of nodes to mutate n at random
20:        children[num_child] = mutate(parent, n);
21:        apply various times 2-opt move, until no other improvements are possible
22:        num_child + +;
23:    end while
```

```
24:     sort the population in order of cost
25:     substitute bad solutions with children and randomly skip a bad solution
26:     update the champion if necessary
27: end while
28: return cost;
```

4.2 Deterministic Algorithms

Deterministic algorithms, given a certain input, always provide the same output, by executing the same sequence of steps. As an example, hill-climbing algorithm is a deterministic algorithm[19].

4.2.1 Benders decomposition

Benders decomposition was first introduced in 1962 [8]. It is a widely used method in mathematical programming and is designed to provide a solution to even particularly complicated linear programming problems that have a partitioned matrix.
Benders decomposition is based on a divide-and-conquer technique. In fact, the variables of the starting problem are separated into two subsets, in this way a first-stage master problem is solved considering only the first subset of variables. On the other hand, the second-stage subproblem is solved considering the second subset of variables. If the first-stage solution is infeasible, we need to create and include Benders cuts to the master problem. It is necessary to solve the master problem until it's not possible to generate more cuts. We can also name this method "row generation" because of the new constraints added to the master problem [32].
In our project, we employ IBM ILOG CPLEX[1] library of functions and build a model for the current TSP instance we're trying to solve, without implementing all the possible Subtour Elimination Constraints (SECs) as they are exponential in quantity and would slow down the program dramatically.
What we do instead is using Bender's idea and solve the instance first with no SECs, obtaining a rough solution made only of a collection of subtours, and we then improve the solution by adding a SEC for each connected component in our current solution and running the CPLEX optimizer again until only one connected component of the graph remains.
In 13 we provide the pseudo code of our approach. First of all, we need to build the model by considering only the degree constraints. Then we generate an initial integer solution of our problem by using CPLEX. To avoid investing excessive time in computation, we set a time limit when calling CPLEX.
The next step consists in calculating the number of connected components and if the solution is not characterized by one cycle, SEC is defined and added for each excessive component.
Basically, the computational complexity depends on two factors:

1. The speed of CPLEX to solve the TSP model. We can consider this step efficient, in fact CPLEX is well optimized and our model is not so complex.

2. The complexity to find the components, it is equal to $O(n^2)$

Algorithm 10 Benders Approach

Input: A TSP instance

Output: A valid tour (CPLEX solution)

```
model ← TSP model with only degree constraints
solution ← solve model
n_components ← calculate number of components
while n_components > 1 do
    for each comp ∈ components do
        add SEC constraint for comp
    end for
    solution ← solve model
    n_components ← calculate number of components
end while
```

Algorithm 11 Compute Connected Components

Input: A solution of TSP

Output: the successors array *succ*,the array of components *comp*, the number of components *nComp*

```
succ ← initialize successor array to -1;
comp ← initialize components array to -1
n_comp ← 0
for start=0 to n-1 do
    if comp[start] > 0 then
        // the node is already in a component
        continue;
    end if
    // A new component has been found
    n_comp ← n_comp +1;
    i ← start;
    done ← false
    while !done do
        comp[i] = nComp;
        done ← true
        for j=0 to nnodes-1 do
            if i ≠ j and x_ij > 0.5 and comp[j] == −1 then
                succ[i] ← j;
                i ← j;
                done ← false
                break;
            end if
        end for
    end while
    succ[i] ← start;
end for
```

4.2.2 Improvement of Benders Approach: Patching Heuristic

One major drawback of the Benders approach is that it only reaches the optimal solution in the last iteration. However, it can happen that the algorithm stops before reaching the last iteration, due to the imposition of a time limit. In this situation, it's best to get at least one feasible solution, so we need to repair our tour. To get a feasible solution, we use patching heuristics.

Algorithm 12 Improvement of Benders Approach

Input: A TSP instance
Output: the solution as an array *succ*, **UB**

```
1: model ← TSP model with only degree constraints
2: LB ← 0;
3: UB ← INF;
4: while LB < 0.9999 * UB do
5:     if current_time > time_limit then
6:         break;
7:     end if
8:     update cplex time limit;
9:     set cutoff limit
10:    set node limit
11:    solution ← solve model
12:    get the best objective value obj
13:    if obj > LB then
14:        LB ← obj;
15:    end if
16:    n_components ← calculate number of components
17:    if ncomp > 1 then
18:        for each comp ∈ components do
19:            add SEC constraint for comp
20:        end for
21:        apply patching Heuristics
22:    end if
23:    compute the current cost
24:    if current_cost < UB then
25:        UB = current_cost;
26:    end if
27: end while
```

A patching operation consists in the elimination of two edges and in the subsequent addition of two new edges in order to reduce the number of cycles by one [41]. Among all the candidates, the edges selected to be added or eliminated are related to the cost of the solution, in particular we want the solution corresponding to the minimum cost. In general, given k cycles, the requirement to form a single tour is $k-1$ patching operations[28]. Note that the patching heuristics and the 2-opt move are very similar, the main difference between the two approaches is the fact that in the patching heuristics we don't act in a single tour. In the example 4.3, in order to generate a single tour, two patching

operations are employed, namely first arcs (2,4), (5,6) are eliminated and (2,6) and (5,4) are added, and then we eliminate (12,9) and (2,6) and add (2,9) and (12,6). Typically, the final tour is feasible but not optimal.

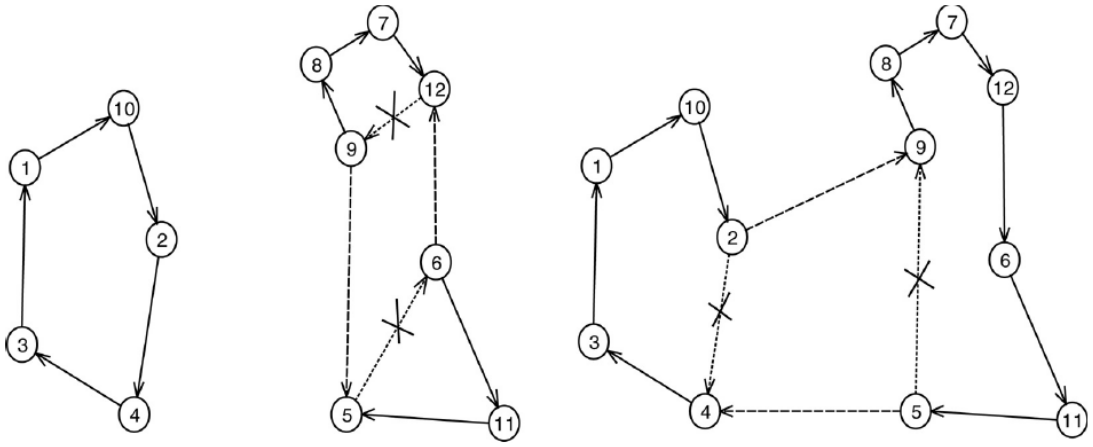


Figure 4.3: Creating a single tour by applying two patching heuristics.

4.2.3 Branch and cut with a callback

An alternative way to solve TSP is using branch and cut with callback function. The Branch and Cut approach was introduced in the 90s to solve and optimize Mixed-Integer Linear Programs [37]. This method is composed of two famous optimization tools, namely Branch and Bound[30] and Cutting Planes[17]. This way, the advantages of both algorithms are exploited, in order to obtain the optimal solution. The Branch and Cut approach achieves an optimal solution by relaxing the problem to generate the upper bound. Through relaxing the problem, it becomes easier to solve. Moreover, the upper bound indicates the highest value of the objective to be feasible. The optimal solution is achieved if the objective is equivalent to the upper bound [26].
The branch and bound approach is implemented by CPLEX, together with a strategy to compute cuts and some heuristics. The callback function is used to interact with the user. CPLEX will call the callback when it has a candidate solution and at this point we need to check whatever the solution is feasible. In particular, if the number of components are more than one, the solution is infeasible and SEC's must be added. In our code, we also obtain a solution by using patching heuristics and 2-opt. We post the solution obtained in this manner and CPLEX will eventually use it to update the incumbent. The pseudocode of the callback function 14 is very similar to the Bender's approach already discussed, the difference is that with branch and cut method, the SEC's are generated on the fly, during the execution of CPLEX. Alternatively, in Bender's approach, different branching trees are generated from scratch.

Algorithm 13 Patching Heuristics

Input: succ, comp, ncomp
Output: A valid tour

```
1: localcomp ← ncomp;
2: while localcomp > 1 do
3:     delta ← INF;
4:     best_a ← −1;
5:     best_b ← −1;
6:     for a=0 to nnodes-1 do
7:         for b=0 to nnodes-1 do
8:             if comp[a] < comp[b] then
9:                 temp_cost ← (cost[a, succ[b]] + cost[b, succ[a]]) − (cost[a, succ[a]] +
   cost[b, succ[b]]);
10:                if temp_cost < delta then
11:                    delta ← temp_cost;
12:                    best_a ← a;
13:                    best_b ← b;
14:                end if
15:            end if
16:        end for
17:    end for
18:    update the solution
19:    a_prime ← succ[best_a];
20:    b_prime ← succ[best_b];
21:    COMPA ← comp[best_a];
22:    h ← b_prime;
23:    fusion of two components
24:    while h ≠ best_b do
25:        comp[h] ← COMPA;
26:        h ← succ[h];
27:    end while
28:    comp[best1_b] = COMPA;
29:    succ[best_a] = b_prime;
30:    succ[best_b] = a_prime;
31:    localcomp − −;
32:    if localcomp ≠ 1 then
33:        add SEC associated to one component
34:    end if
35:    compute 2-optimality move;
36: end while
```

Algorithm 14 CallBack function

Input: Cplex heuristics solution x^h

```
1: compute succ, comp, ncomp
2: if ncomp > 1 then
3:     for i=0 to ncomp-1 do
4:         reject candidate solution and add one cut
5:     end for
6: end if
```

4.3 Matheuristics

Matheuristic, as the name reflects, is a combination of mathematical programming and metaheuristics[12]. In particular, the idea of matheuristic consists in the use of the solver as a "black-box" and in the modification of the model provided in input to the solver. This approach is a general design, so it can be employed in various applications, with a small adjustment to their structure [9].
Matheuristics produce a good solution at early stages, this is more convenient with respect to a optimal solution at the late steps of the algorithm. In fact, due to the use of a time limit, perhaps the optimal solution will never be reached [13].
In this section, we provide some of the most famous approaches.

4.3.1 Hard Fixing

In order to decrease the primal integral and aggressively reduce the upper bound, one idea is to keep CPLEX unaltered and modify the model provided to CPLEX using Hard Fixing heuristic. Given an initial solution, in Hard Fixing some edges are fixed $x_{ij} = 1$ and others are maintained free $x_{ij} \in [0, 1]$. This approach is iteratively re-used on the smaller problem obtained from fixing: CPLEX solver is called another time, a new target solution is obtained, some of its variables are fixed, and on and on. In such manner, the problem size decreases after each iteration, consequently, the CPLEX solver can focus on smaller and smaller problems with increasing probability to find the optimal solution.
A crucial point, is the way to choose the variables to be fixed.
In our code, the edges to be fixed are chosen randomly, given a certain probability $pfix$.
Here we provide the pseudocode of this approach 15.

Algorithm 15 Hard Fixing

Input: instance, pfix
Output: a valid tour

```
1: get an initial solution
2: while current_time < time_limit do
3:     for each edge do
4:         if random01() < pfix then
5:             fix edge in the model;
6:         end if
7:     end for
8:     tour ← solve model;
9:     unfix all edges of the model;
10: end while
11: return tour;
```

4.3.2 Local Branching

Local Branching, a local search heuristic, was introduced in 2003 by M. Fischetti and A. Lodi [13].

Local Branching begins with an incumbent feasible solution x^h and describes neighborhoods by providing which variables to fix [9]. To better get the idea of local branching, consider x^h and x, two feasible solution, where x^h is the incumbent feasible solution and x is a general solution of the neighborhood. Consider the variable index set $\mathcal{N} = \{1, ..., n\}$, it is divided into $(\mathcal{B}, \mathcal{G}, \mathcal{C})$, where $\mathcal{B} \neq \emptyset$ contains the indices of the 0-1 variables. The hamming distance is $\Delta(x^h, x) = |\{j \in B : |x_j^h - x_j| = 1\}|$, and the subset of binary variables which have the value of 1 is expressed by $S^h = \{j \in B : x_j^h = 1\}$. Local Branching delineates a restricted neighborhood of x^h that will be investigate by the sub-MIP, including only the solutions satisfying the extra constraint $\Delta(x^h, x) \leq k$,where k is a radius hyperparameter. The local branching sub-MIP is composed by all cutting planes and variable bounds originating from valid inequalities discovered during the investigation of the global branch-and-cut tree and excludes variable bounds established by branching, which are accepted only on a subtree. The restriction on the dimension of the neighborhood is encouraged by including to the formulation the known also as local branching constraints. Let k be the value of the hyperparameter, the k-opt neighborhood $\mathcal{N}(x^h, k)$ of x^h of an incumbent solution is described as the set of feasible solutions of the initial MIP satisfying the extra local branching constraint:
$\Delta(x^h, x) = \sum_{j \in S^h}(1 - x_j) + \sum_{j \in B \setminus S^h} x_j \leq k$,
where the two terms in left-hand side represent the number of variables converting their value from 1 to 0 and from 0 to 1, with respect to x^h.
An equivalent expression of the above formula is given by:
$\sum_{j \in S^h}(1 - x_j) \leq k' = (k/2)$,

Algorithm 16 Local Branching

Input: instance
Output: a valid tour

```
1: get an initial solution;
2: while current_time < time_limit do
3:     add local branching constraint to the model
4:     tour ← solve model;
5:     remove last constraint;
6: end while
7: return tour;
```

4.3.3 Matheuristics inside CPLEX

RINS

Relaxation induced neighborhood search (RINS) approach, proposed in 2005 by Danna, Rothberg and Le Pape [11], has some ideas in common with local branching, described in section 4.3.2. In fact, both methods present the neighborhood as a sub-MIP and investigate it employing the MIP solver.
Commonly, in the branch-and-cut tree two different solutions are produced. The current solution is feasible, considering the integrality constraints, however, it is not optimal until reaching the final step. On the other hand, the solution obtained by relaxing the problem,

in the majority of cases is not integral, but the value of its objective function is regularly better with respect to the incumbent solution.
Therefore, the incumbent solution and the continuous relaxation solution, reach one of the two purposes, integrality and optimization of the objective value respectively.
A crucial observation is that the values of the incumbent and continuous relaxation are not all different, but some are in common. The basic idea of RINS consists in the use of these variables as partial solution, with the intention of expanding it, to form a complete solution, which accomplishes both desires, namely, integrality and optimality. Clearly, the intention of RINS is to give more attention to the variables that are not congruent in the incumbent and continuous relaxation.
RINS algorithm goes as follows, at a node of the global branch-and-cut tree, the subsequent phases are executed:

1. Fix the elements in common between the incumbent and the optimal solution at the current node;

2. Place an objective cutoff by looking at the value of the objective of the current incumbent;

3. Resolve a sub-MIP on the residual variables.

In various cases, since the RINS algorithm is hard and extensive, we need to stop it, for example by using a node limit nl.

Polishing Heuristics

Polishing Heuristics, introduced by Rothberg [39], is based on an evolutionary MIP heuristic which choose nodes of a branch-and-bound tree by employing the classical components of the genetic algorithm, for instance [12]:

- Population: It is composed of feasible solutions collected by employing the polishing approach itself or by distinct heuristics. Moreover, the dimension of the population, in terms of number of solutions, is preserved during the algorithm.

- Combination: Two or more elements of the population (the parents) are associated with the purpose of producing a new element of the population (the offspring) with better features. The RINS idea is used, namely, all variables of the parents which are the same are fixed, and the smaller MIP is heuristically resolved by a black-box MIP solver, not over a finite number of branch-and-bound nodes. Notice that the described procedure is significantly more laborious with respect to a conventional step in evolutionary approaches, although it ensures feasibility of the child solution.

- Mutation: In order to acquire diversification a conventional mutation step is computed, which (i) randomly chooses an element in the population, (ii) randomly fixes a few of its variables, and (iii) heuristically resolves the smallest MIP originated.

- Selection: The two parents to reproduce are selected at random, where the second one is chosen between the solutions with a good objective function.

In this project, we combine RINS and polishing heuristics into a unique matheuristic. Their implementation is already present in CPLEX, hence we only set some parameters, namely, CPXPARAM_MIP_Strategy_RINSHeur and CPXPARAM_MIP_PolishAfter_Time. The first parameter represents the node frequency of the RINS heuristic and the second one the time after which CPLEX starts polishing. In our experiments, we set the node frequency to 30 and the CPXPARAM_MIP_PolishAfter_Time to 15 seconds.

Chapter 5

Experimental Results

In this section, we will outline the methodology and results adopted for testing and collecting data from all the algorithms we described previously. The focus this year was very much on heuristic approaches, with only 2 exact methods being implemented, while the heuristics implemented by us totaled to 8. In order to compare everything appropriately, we needed to standardize our testing methods to collect huge amounts of data in an automated fashion.
To perform the experiments we built a specialized set of `.bat` files to run the program a large number of times with different values for the random seed, having every relevant result prefaced by a `$STAT` marker and then collected via the `findstr` batch command. Afterwards we dumped the data to a `.csv` file that was then fed to a python script [5] , that would then transform our data into performance plots of the tests, as shown later in this section.
The program we developed is also capable of reading files from the TSPLIB [6], but due to the relative low number of instances for the 2D Symmetric TSP, we used randomly generated instances for our test instead.
Since heuristics are unlikely to ever reach optimality, we also implemented a time limit function to abort the program and provide the best available solution once the time limit is reached. Some of the algorithms can potentially run forever in search of a global optimal solution, so the use of a time limit is very important to a fair comparison.
To test the exact methods we instead made sure to omit the time limit and instead benchmark them on how fast they reach optimality, since they both get to the global optimal solution eventually. For these programs we didn't care about collecting the cost of the solution, but just the time taken to reach the optimal final solution.

5.1 Dataset

The dataset used for our experiments is a collection of randomly generated TSP instances with a variable number of nodes. Before each run of the program, we can set the random seed of the instance to generate and how many nodes it will be composed of. The instance is randomly generated with each point having a random X and Y coordinate between 0 and 1000. Depending on the experiment we used a different number of nodes or instances (modifiable via a command line parameter), as will be detailed, due to technological constraints.
Tests using heuristic algorithms were done on instances of 1000 nodes, while to benchmark

the exact methods we settled on smaller instances of 300 nodes, since CPLEX takes quite a long time to solve larger instances.

5.2 Algorithm tuning

Before illustrating the results, we tuned some of the parameters to find how to best run certain algorithms, as illustrated in this section.

5.2.1 Tuning Tabu Search

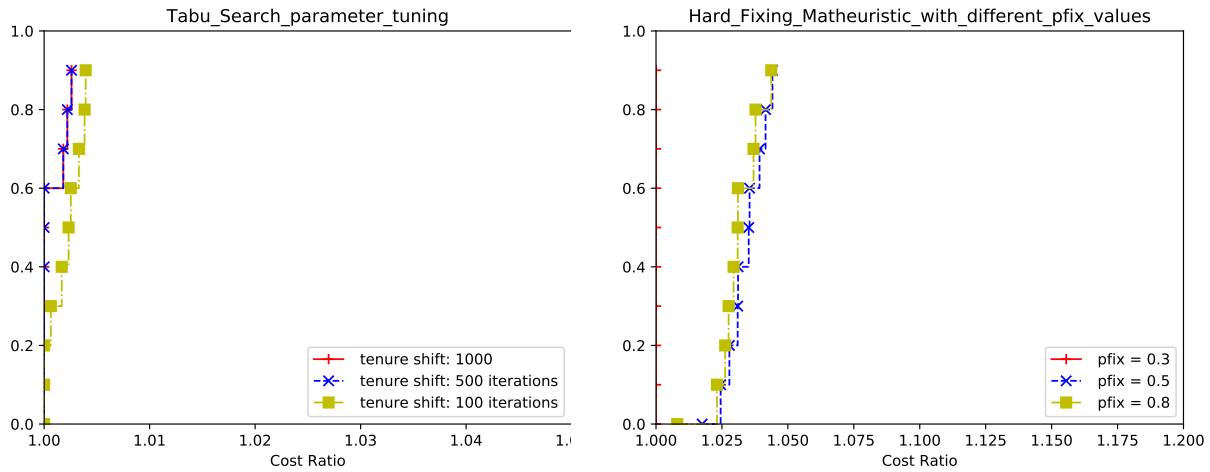
We implemented Tabu Search with a variable Tenure value that alternates between intensification (Tabu List size =10) and diversification (Tabu List size = 100) every I iterations. Such I, or tenure shift as we called it, is a hyperparameter we tuned over some possible different values, shown in the performance profile in Figure 5.1a. Since we needed to get the tuning done fairly quickly, we only used 10 random instances of 1000 nodes with a runtime of 300 seconds.

Surprisingly, the performance with $I = 1000$ and $I = 500$ is identical, while $I = 100$ is slightly worse in terms of the cost of the solutions obtained, so we opted to keep $I = 1000$.

5.2.2 Tuning Hard Fixing

The Hard Fixing Matheuristic has a random chance of fixing an edge to 1 in a heuristic solution provided before the run of CPLEX callback. This probability is a tunable parameter we call $pfix$, and the algorithm was tuned on 10 random instances of 1000 nodes with a time limit of 300 seconds. The results are shown in Fig. 5.1b.

The clear winner here is $pfix = 0.3$, so a 30% probability of fixing and edge in the starting heuristic solution. Fixing more edges equates to searching a much smaller neighborhood of solutions, so this limitation probably doesn't allow to improve the edge configuration as much.



(a) Performance profile of the variants of Tabu Search with different values of I

(b) Performance profile of the variants of Hard Fixing with different values of $pfix$

Figure 5.1: tuning plots

5.3 Comparing Heuristics

We compared three different heuristic algorithms: Multi start Nearest Neighbor, Multi start Extra Mileage and Farthest start Extra Mileage, all implemented with the GRASP approach and a second choice probability of 10%. What these three consist of is:

1. Multi start Nearest Neighbor: Nearest Neighbor heuristic applied to every possible starting node in the graph.

2. Multi start Extra Mileage: Extra Mileage heuristic applied to every possible starting subtour in the graph (very intensive).

3. Farthest start Extra Mileage: Extra Mileage heuristic applied to the starting subtour made of the two farthest nodes in the graph. This is a particular case of multi start, so if we let both programs run until the end, it will always be worse, but with a stringent time limit of 5 minutes, it might find a better solution.

The dataset for this comparison was composed of 100 randomly generated instances with seed from 1 to 100, all of which had 1000 nodes. The time limit was set to 5 minutes to avoid having Multi start Extra Mileage running for too long.
All results have then been refined with the 2-opt algorithm, which has been set to run until no more moves are possible, or the time limit runs out. For the multi start Extra Mileage algorithm, since it's likely it won't reach the end by the time 5 minutes pass, we left 5 seconds to perform 2-opt on the best solution found to even out the results with the algorithms that are able to terminate within the 5 minute time limit.
The results can be seen in the performance profile graph at Fig.5.2 In general the fastest algorithm is undoubtedly Multistart Nearest Neighbor, as it only requires runtime that is $O(n^3)$ in the number of nodes n. Farthest Start Extra Mileage is also $O(n^3)$ but requires a preliminary pass of all nodes to determine the farthest possible couple, while

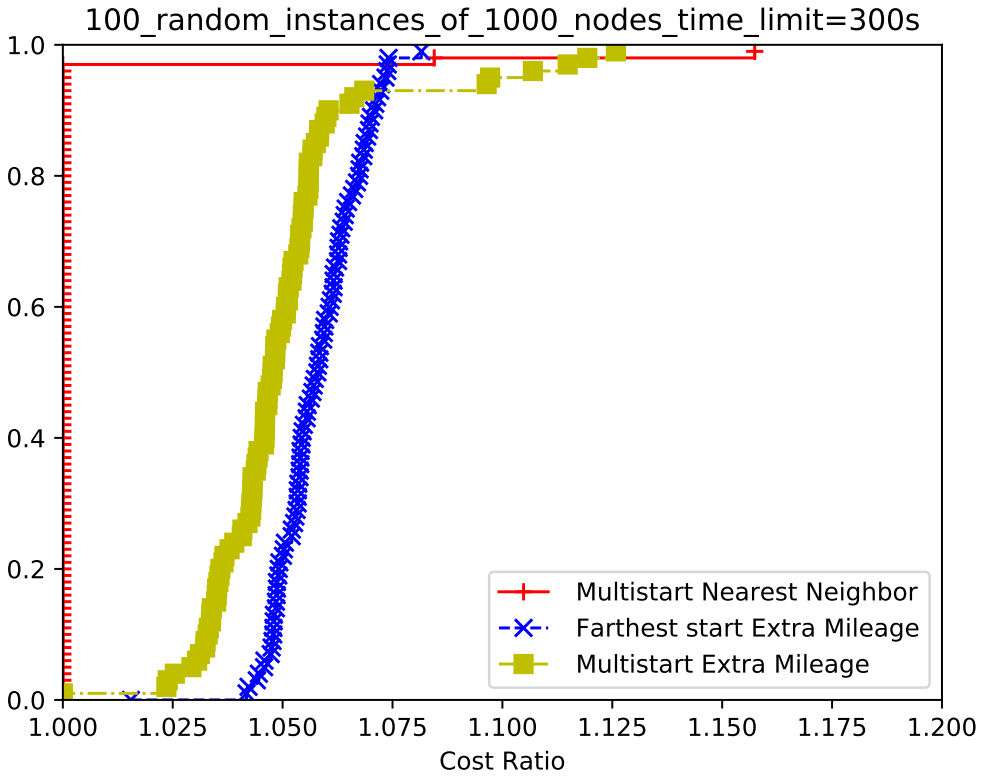


Figure 5.2: Performance profile of the three heuristic algorithms

Multi Start Extra Mileage is very intensive at $O(n^5)$. When it comes to the cost of the results, the graph seems to indicate that Multi start Nearest Neighbor provides the best solutions, although it seems to fall off in some cases. Still, even if the other two algorithms can outdo it in some scenarios, we chose to mostly use the Nearest Neighbor heuristic due to its speed, meaning it can be applied to obtain a fast starting solution for other metaheuristc algorithms.

5.4 Comparing Metaheuristics

We developed 4 different metaheuristic algorithms for the project, so to compare them all we had to scale back the number of instances to get it done in a reasonable amount of time. All the metaheuristics we implemented use a GRASP Nearest Neighbor solution as the beginning of their exploration of the solution space. We used a testbed of 50 random instances of 1000 nodes to test the 4 algorithms on: Tabu Search, VNS, Genetic Search and Simulated Annealing, all with a time limit of 300 seconds.
In specific each algorithm was implemented as such:

1. Tabu Search: the algorithm begins from a multi start GRASP Nearest Neighbor solution that isn't 2-optimized, and starts working from there alternating a tenure of 10 for intensification and 100 for diversification every 1000 iterations of the modified 2-opt algorithm.

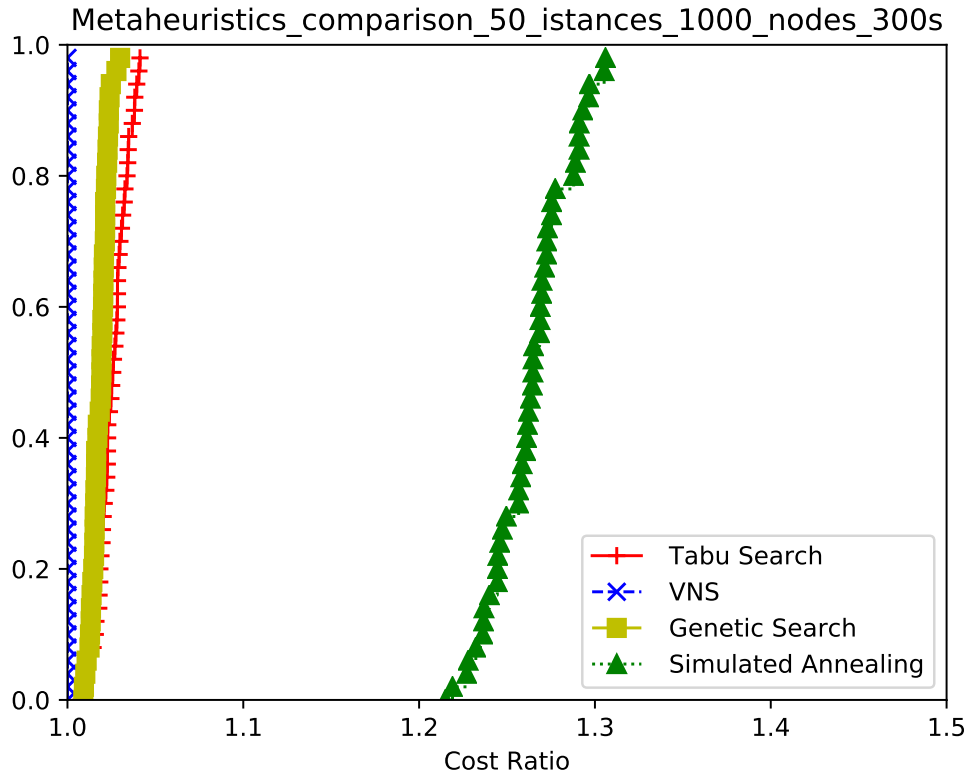


Figure 5.3: Performance profile of the four metaheuristic algorithms

2. Variable Neighborhood Search: from a multi start GRASP Nearest Neighbor solution we run 2-opt until no more moves can be made, then use a 5-opt kick to begin re-exploring the solution space.

3. Genetic Search: from a population of 1000 GRASP Nearest Neighbor Solutions with a random node start and with no 2-opt, we create 500 children, 400 of which are obtained via merging and 100 of which via mutation each generation. The children are then repaired and 2-opted to make them feasible solutions and the best one each generation is made the champion and compared to find the best overall solution.

4. Simulated Annealing: implemented from a multistart GRASP Nearest Neighbor solution with no 2-opt, the probability of making a non-optimal 2-opt move was calculated with the formula $p = e^{\frac{-\Delta cost}{scaler \cdot T}}$, with the delta cost parameter being the difference in cost before and after the move, the scaler being roughly the average cost of an edge in a reference heuristic solution, and the temperature being a parameter starting from 10 that gradually decreases as the algorithm reaches the time limit.

The results are shown in Figure 5.3. As expected the Simulated Annealing approach works very badly for the TSP, while the others are a lot more evenly matched. In our tests, VNS comes out as the clear victor in terms of getting the best solutions, while Tabu Search and Genetic Search obtain similar performance.

5.5 Comparing Matheuristics

The two Matheuristics algorithms we developed were compared on a testbed of 100 random instances of 1000 nodes, with a time limit of 300 seconds, so that neither approach can easily reach optimality during the 30 seconds in which CPLEX Callback is employed. Both algorithms start from a GRASP Nearest Neighbor heuristic solution and begin refining it from there with CPLEX callback runs on a short time limit. For Hard Fixing the probability to fix a random edge was set to 30%, while in Local Branching the neighborhood size k was set to 10. The results obtained are in Figure 5.4.

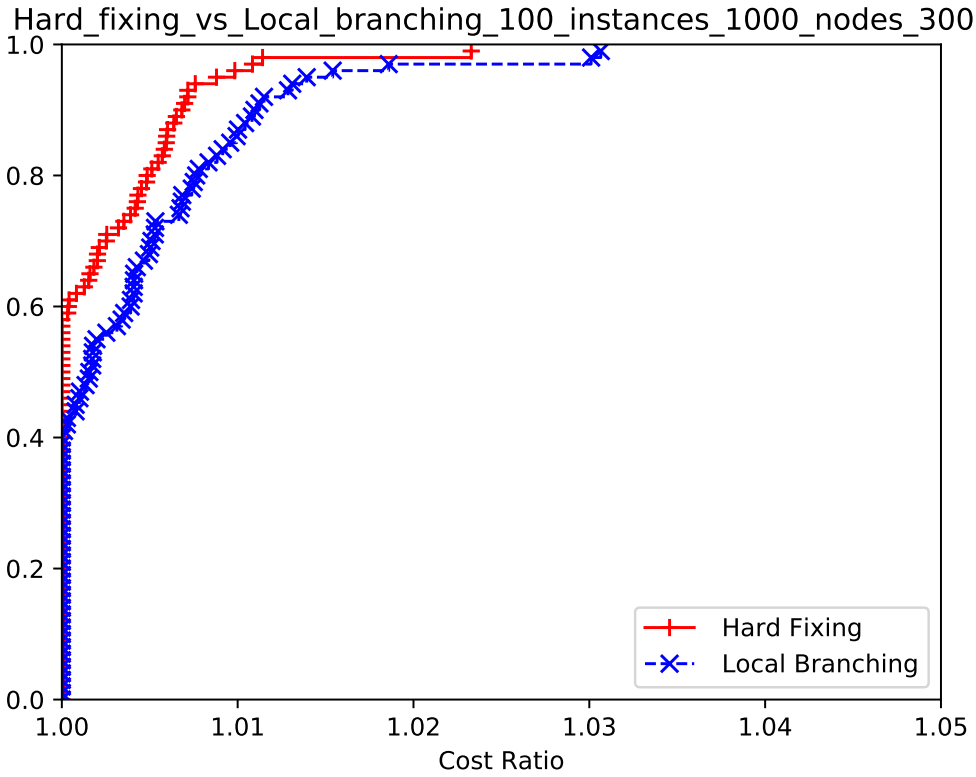


Figure 5.4: Performance profile of the two matheuristic algorithms

The Hard Fixing matheuristic comes out on top every time, making it clearly better for obtaining solutions with low cost in the same timeframe. The two approaches are however very close, more so than any other heuristic comparison.

5.6 Comparing exact methods

To compare the efficiency of the exact solver methods we implemented, those being the Benders' Loop and CPLEX Callback methods, we ran tests on 100 random instances of 300 nodes and kept track of the time it took for each approach to reach optimality.
We then compared the runtimes of the two algorithms in the performance plot at Figure 5.5.

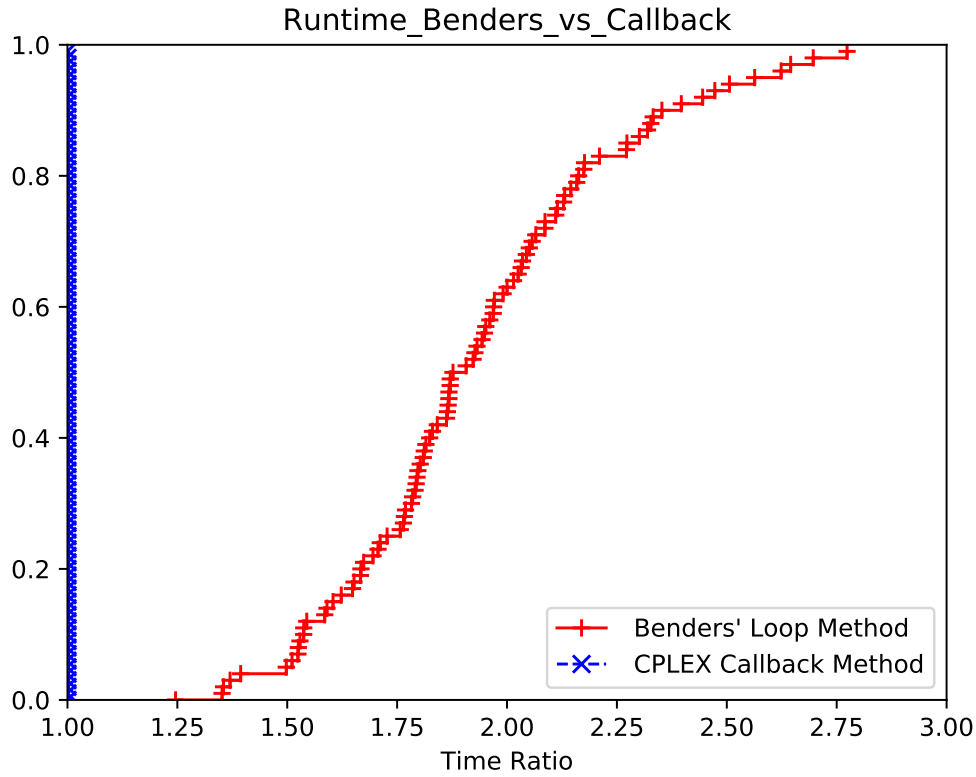


Figure 5.5: Performance profile of the runtime of the two exact algorithms

Both algorithms use a multi start GRASP Nearest Neighbor solution as an initial MIP start, and the implementations of the two methods were as follows:

- Benders' Loop: Implemented in such a way to add SECs to the CPLEX model every time an optimal infeasible solution is obtained composed by multiple subtours, one for each component. The infeasible solution is then repaired via a Patching Heuristic and 2-opted to obtain a feasible solution in case the time limit is exceeded. A cutup on the Upper Bound of the objective function and a node limit of 100 were also specified as CPLEX parameters.

- CPLEX Callback: like the Benders' method, we also use the Patching Heuristic and 2-opt to have a feasible solution ready in case the program exceeds the time limit, all in a thread-safe manner since it makes use of multiple CPUs. We also post these solutions to CPLEX in case they happen to be better than the current incumbent, to potentially speed up the solving process.

Indisputably, the CPLEX Callback algorithm always reaches optimality faster in every single test, making it preferable for use in Matheuristic algorithms too.

5.7 Comparison of all the best heuristic methods

For the final comparison we decided to take all the best performing heuristics and compare them all in one final test. We picked multi start GRASP Nearest Neighbor as it's the fastest and most reliable standard heuristic, VNS since it's the best metaheuristic, and decided to benchmark both the regular Hard Fixing heuristic with $pfix = 0.3$ and two combined matheuristics, already present in CPLEX, namely RINS and Polishing heuristic. The testbed for this was 50 random instances of 1000 nodes with a time limit of 300 seconds.
Multi start GRASP Nearest Neighbor was used as a starting template solution in all other meta and matheuristic methods, so it's maybe a bit pointless to include in the graph, but the results are nonetheless shown in Fig. 5.6

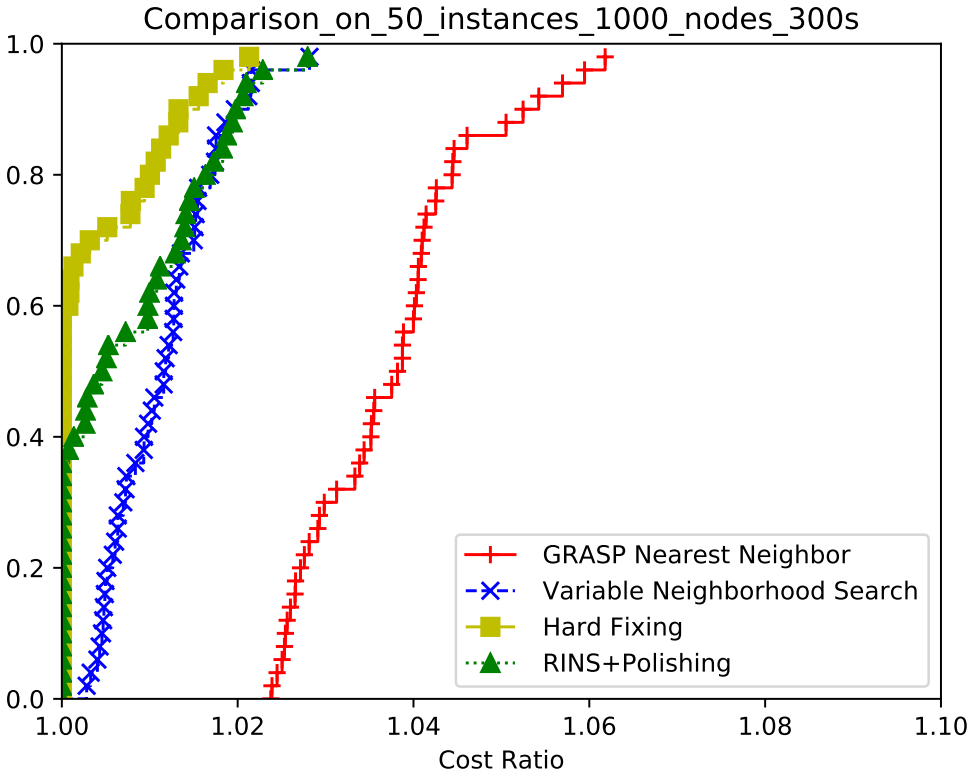


Figure 5.6: Performance profile of the best heuristic algorithms

In the end, of all heuristic methods presented in this thesis, Hard Fixing with $pfix = 0.3$ ends up being the best of them all in terms of being executed on the specific testbed we used within a time limit of five minutes.

Chapter 6

Conclusion

The Traveling Salesman Problem (TSP) is one of the best known mathematical optimization problems, which consists of a set of nodes and costs associated with each edge. The aim is to find the shortest possible tour that visits each node exactly once and then returns to the starting node.
The Travelling Salesman Problem is crucial for practical applications and being employed as a standard problem for performance investigation of numerous algorithms in the discrete optimization field.
In this work, the search for optimum of a TSP problem is compared using 2 heuristics, 4 metaheuristics, 2 exact methods and 3 matheuristics. The heuristics are Nearest Neighbor (NN) and Insertion Heuristic (IH). The metaheuristics are Tabu Search (TS), Variable Neighborhood Search(VNS), Simulated Annealing (SA) and Genetic Algorithm (GA). The exact methods are Benders Decomposition (basic version and an improved version using patching heuristic) and Branch and Bound with a callback. Finally, the matheuristics are Hard Fixing (HF), Local Branching (LB) and RINS combined with a polishing heuristic. The winners respectively for each category are: Nearest Neighbor (NN), Variable Neighborhood Search(VNS), Branch and Bound with a callback and Hard Fixing.
We would like to indicate some notions that might boost the algorithms presented here. We are sure that by tuning other parameters of our methods, the results can improve dramatically. Anyway, the best approach presented here is clearly Branch and Bound with callback. It is also important to consider that an exact method needs long computing time also for instances with a small number of nodes.
If the size of the problem is over 1000 nodes, the exact methods might not obtain the optimal solution very quickly, at which point heuristics might be preferable. It all comes down to one crucial element, speed.

Chapter 7

Appendix

7.1 Input File Parsing

Our program can also take as input and solve files from the TSPLIB website [6] standard format file for Symmetric TSP.
These are text files with the `.tsp` extension formatted in the following way:

- `NAME`: the name of the specific instance
- `COMMENT`: any further details about the specific instance
- `TYPE`: the type of problem, TSP in our case
- `DIMENSION`: the number of nodes in the graph
- `EDGE_WEIGHT_TYPE`: the type of edge the graph uses, mostly related to the distance metric between the two nodes. We will use two-dimensional euclidian distance. To toggle between integer rounding for the distances we use the command line argument -i_c (integer costs). We only implemented the $\texttt{EUC}_2D$ and `ATT` types in our program. `NODE_COORD_SE` the final section of the file, detailing in every line the number of the node and its `X` and `Y` coordinate va
- `EOF`: End Of File flag.

The command line arguments we implemented are as follows:

- `-file <filename>` or `-input <filename>` or `-f <filename>`: the name of the input file if one is being used.
- `-seed <randomseed>`: sets the random seed for the program run.
- `-mode <X>`: the mode of the solver which will be used for the program. The modes are as follows:
  1. GRASP Nearest neighbor heuristic + 2opt until no move is possible
  2. farthest couple start GRASP Extra mileage heuristic + 2opt until no move is possible
  3. multi start GRASP Extra mileage heuristic + 2opt until no move is possible
  4. Tabu search metaheuristic starting from GRASP Nearest neighbor solution

5. VNS metaheuristic starting from GRASP Nearest neighbor solution
6. Genetic metaheuristic from a population of 1000 GRASP Nearest neighbor solutions
7. Cplex exact solution implementation with Benders' loop
8. Cplex exact solution implementation with Branch Cut via Callback
9. Hard Fixing Matheuristic
10. Local Branching Matheuristic
11. Simulated Annealing metaheuristic

By default the mode is set to 0.

- `-tl <timelimit>`: the time limit set for the program's total runtime.
- `-n <nodes>`: number of nodes of a randomly generated instance if one is used. Defaults to 1000.
- `-i_c`: integer costs flag, sets all distances between points to be rounded to integer values.
- `-p`: plot flag, when specified it plots the instance before solving it.

7.2 Plotting with GNUplot

To make plots of the solutions at the end of every run of the program we used the GNUplot utility.
Once the sequence of points of the solution is saved in the `succ` array and subsequently printed into the `data.dat` file, we use these commands to plot the corresponding figure:

```
set style line 1 \
linecolor rgb '#FF0000' \
linetype 1 linewidth 2 \
pointtype 7 pointsize 2
plot "data.dat" with linespoints linestyle 1
pause mouse close
```

The first 4 lines of code are just stylistic typesetting for the lines, points and colors we will use in the plots, while the last line reads data from a `data.dat` text file where the coordinates of the lines and points are saved.
The `data.dat` file is structured with the X and Y coordinates of the points in each line, all separated by an empty line to plot just the points; while to plot the edges, we have to put the X and Y coordinates of the two points that form edge in contiguous lines.
An example:

```
2.5 3.5

4 6.5
```

plots the two points separately.

```
2.5 3.5
4 6.5
```

plots the two points and an edge connecting them.

The results end up looking like Fig. 7.1 for just plotting the points, or Fig. 7.2 for a solution.

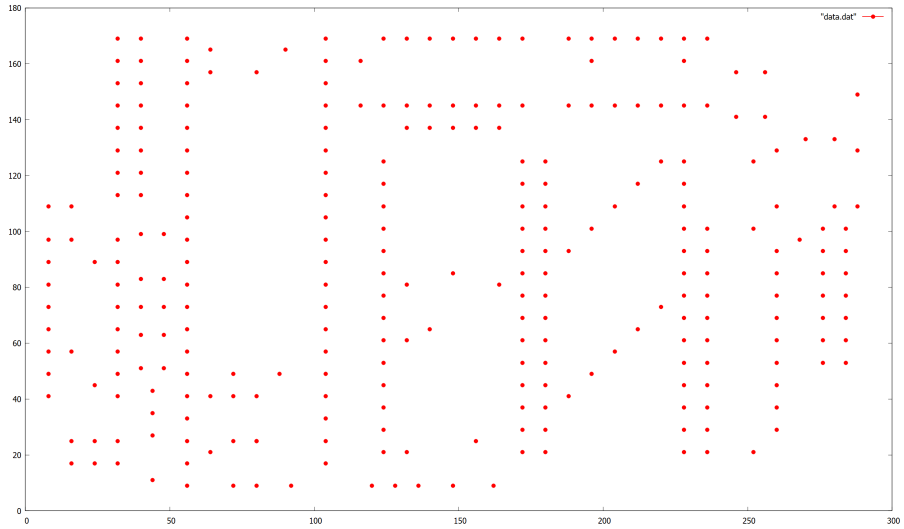


Figure 7.1: Plot of the a280.tsp TSPLIB instance

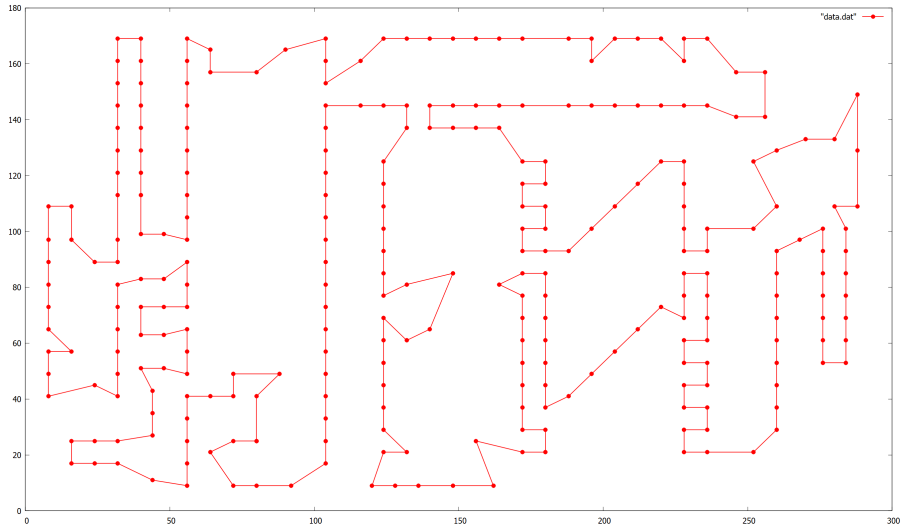


Figure 7.2: Plot of the optimal solution of the a280.tsp TSPLIB instance

7.3 Batch file structure

The `.bat` files we used are simple text files that contain lines of code to instruct the Windows OS to perform certain actions in an automated fashion. The main structure of our files was as follows:

```
@echo off
setlocal enableextensions enabledelayedexpansion
CALL :LOG >logfile.log
FINDSTR /i /s /c:"$STAT" logfile.log > results.txt

SET /A count=0
SET line =
FOR /F "tokens=2 delims= " %%G in (results.txt) DO (
  SET /A count+=1
  SET /A flag=!count!  %% 3
  IF !flag!  == 0 (
    SET line=!line!%%G
    ECHO.!line!  >> table.csv
    SET line=
  )
  IF NOT !flag!  == 0 (
    SET line=!line!%%G,
  )
)
pause
exit /B
```

Where then the `:LOG` subroutine contains our program calls as if they were being run from the terminal, like:

```
:LOG
echo $STAT seed_1
tsp.exe -n 1000 -mode 1 -i_c -tl 300 -seed 1
tsp.exe -n 1000 -mode 2 -i_c -tl 300 -seed 1
```

The only change that needs to be made is the red number that has to be adjusted with the number of `$STAT` items you want in each line of the final `.csv` file. The loop basically takes every line from the `logfile.log` with a `$STAT` flag, followed by the relevant value (usually the cost of the final solution) and counts up to the number of items specified before going to a new line.

List of Figures

List of Algorithms

Bibliography

[1] URL: https://www.ibm.com/uk-en/analytics/cplex-optimizer.

[2] URL: http://www.dei.unipd.it/~fisch/ricop/RO2/howto_WinCplexGnuplot.pdf.

[3] URL: http://www.gnuplot.info/.

[4] URL: http://www.dei.unipd.it/~fisch/ricop/RO2/RO2_Cplex_TEMPLATE_2020.zip.

[5] URL: http://www.dei.unipd.it/~fisch/ricop/RO2/PerfProf/.

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

[7] Emile HL Aarts, Jan HM Korst, and Peter JM van Laarhoven. "A quantitative analysis of the simulated annealing algorithm: A case study for the traveling salesman problem". In: *Journal of Statistical Physics* 50.1 (1988), pp. 187–206.

[8] J BnnoBRs. "Partitioning procedures for solving mixed-variables programming problems '". In: *Numerische mathematik* 4.1 (1962), pp. 238–252.

[9] Marco Antonio Boschetti and Vittorio Maniezzo. "Matheuristics: using mathematics for heuristic design". In: *4OR* 20.2 (2022), pp. 173–208.

[10] Ann Melissa Campbell and Martin Savelsbergh. "Efficient insertion heuristics for vehicle routing and scheduling problems". In: *Transportation science* 38.3 (2004), pp. 369–378.

[11] Emilie Danna, Edward Rothberg, and Claude Le Pape. "Exploring relaxation induced neighborhoods to improve MIP solutions". In: *Mathematical Programming* 102.1 (2005), pp. 71–90.

[12] Martina Fischetti and Matteo Fischetti. "Matheuristics". In: *Handbook of Heuristics*. Ed. by Rafael Martí, Pardalos Panos, and Mauricio G.C. Resende. Cham: Springer International Publishing, 2016, pp. 1–33.

[13] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical programming* 98.1 (2003), pp. 23–47.

[14] Fred Glover. "Future paths for integer programming and links to artificial intelligence". In: *Computers & operations research* 13.5 (1986), pp. 533–549.

[15] Fred Glover. "Tabu search—part I". In: *ORSA Journal on computing* 1.3 (1989), pp. 190–206.

[16] Fred Glover. "Tabu search—part II". In: *ORSA Journal on computing* 2.1 (1990), pp. 4–32.

[17] Ralph E. Gomory. “An algorithm for integer solutions to linear programs”. In: 1958.

[18] Martin Grötschel and Olaf Holland. “Solution of large-scale symmetric travelling salesman problems”. In: *Mathematical Programming* 51.1 (1991), pp. 141–202.

[19] A Hanif Halim and IJAoCMiE Ismail. “Combinatorial optimization: comparison of heuristic algorithms in travelling salesman problem”. In: *Archives of Computational Methods in Engineering* 26.2 (2019), pp. 367–380.

[20] Pierre Hansen and Nenad Mladenovic. “A tutorial on variable neighborhood search”. In: *Les Cahiers du GERAD ISSN* 711 (2003), p. 2440.

[21] Pierre Hansen et al. “Variable neighborhood search”. In: *Handbook of metaheuristics*. Springer, 2019, pp. 57–97.

[22] Keld Helsgaun. “An effective implementation of the Lin–Kernighan traveling salesman heuristic”. In: *European journal of operational research* 126.1 (2000), pp. 106–130.

[23] Alain Hertz, Eric Taillard, and Dominique De Werra. “A tutorial on tabu search”. In: *Proc. of Giornate di Lavoro AIRO*. Vol. 95. 1995, pp. 13–24.

[24] John H Holland. *Adaptation in natural and artificial systems*. MIT press, 1975.

[25] Kazuma Honda, Yuichi Nagata, and Isao Ono. “A parallel genetic algorithm with edge assembly crossover for 100,000-city scale TSPs”. In: *2013 IEEE congress on evolutionary computation*. IEEE. 2013, pp. 1278–1285.

[26] Miroslav Karamanov. *Branch and Cut: An Empirical Study*. Carnegie Mellon University, 2006.

[27] Robert L Karg and Gerald L Thompson. “A heuristic approach to solving travelling salesman problems”. In: *Management science* 10.2 (1964), pp. 225–248.

[28] Richard M Karp. “A patching algorithm for the nonsymmetric traveling-salesman problem”. In: *SIAM Journal on Computing* 8.4 (1979), pp. 561–573.

[29] John Knox. “Tabu search performance on the symmetric traveling salesman problem”. In: *Computers & Operations Research* 21.8 (1994), pp. 867–876.

[30] A. H. Land and A. G. Doig. “An Automatic Method of Solving Discrete Programming Problems”. In: *Econometrica* 28.3 (1960), pp. 497–520.

[31] Pedro Larranaga et al. “Genetic algorithms for the travelling salesman problem: A review of representations and operators”. In: *Artificial intelligence review* 13.2 (1999), pp. 129–170.

[32] Leon S Lasdon. *Optimization theory for large systems*. Courier Corporation, 2002.

[33] Jan Karel Lenstra and AHG Rinnooy Kan. “Some simple applications of the travelling salesman problem”. In: *Journal of the Operational Research Society* 26.4 (1975), pp. 717–733.

[34] Nicholas Metropolis et al. “Equation of state calculations by fast computing machines”. In: *The journal of chemical physics* 21.6 (1953), pp. 1087–1092.

[35] Nenad Mladenović and Pierre Hansen. “Variable neighborhood search”. In: *Computers & operations research* 24.11 (1997), pp. 1097–1100.

[36] Christian Nilsson. "Heuristics for the traveling salesman problem". In: *Linkoping University* 38 (2003), pp. 00085–9.

[37] M. Padberg and G. Rinaldi. "Optimization of a 532-city symmetric traveling salesman problem by branch and cut". In: *Operations Research Letters* 6.1 (1987), pp. 1–7. ISSN: 0167-6377. DOI: https://doi.org/10.1016/0167-6377(87)90002-2. URL: https://www.sciencedirect.com/science/article/pii/0167637787900022.

[38] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis II. "An analysis of several heuristics for the traveling salesman problem". In: *SIAM journal on computing* 6.3 (1977), pp. 563–581.

[39] Edward Rothberg. "An evolutionary algorithm for polishing mixed integer programming solutions". In: *INFORMS Journal on Computing* 19.4 (2007), pp. 534–541.

[40] Christopher C Skiscim and Bruce L Golden. *Optimization by simulated annealing: A preliminary computational study for the tsp*. Tech. rep. Institute of Electrical and Electronics Engineers (IEEE), 1983.

[41] Marcel Turkensteen et al. "Iterative patching and the asymmetric traveling salesman problem". In: *Discrete Optimization* 3.1 (2006), pp. 63–77.

[42] Dominique de Werra and Alain Hertz. "Tabu search techniques". In: *Operations-Research-Spektrum* 11.3 (1989), pp. 131–141.

[43] Shi-hua Zhan et al. "List-based simulated annealing algorithm for traveling salesman problem". In: *Computational intelligence and neuroscience* 2016 (2016).