Lab exercise - Part I & II

Methods and Models for Combinatorial Optimization

Furno Francesco

francesco.furno@studenti.unipd.it - 2139507

A.A. 2024 - 2025

Table of contents

Figures

Tables

1. Introduction

This project presents two solutions for the Travelling Salesman Problem (TSP), utilizing two distinct approaches:

1. **Mathematical Model**: This approach employs the CPLEX C API's to formulate and solve the TSP as an optimization problem.
2. **Heuristic Model**: This approach uses the Simulated Annealing algorithm to find an approximate solution to the TSP.

Both methods aim to minimize the total travel distance while visiting each point exactly once and returning to the starting point. The mathematical model guarantees an optimal solution, whereas the heuristic model provides a near-optimal solution in a shorter amount of time.

1.1. Problem description

A company produces boards with holes used to build electric panels. Boards are positioned over a machine and a drill moves over the board, stops at the desired positions and makes the holes. Once a board is drilled, a new board is positioned and the process is iterated many times. Given the position of the holes on the board, the company asks us to determine the hole sequence that minimizes the total drilling time, taking into account that the time needed for making an hole is the same and constant for all the holes.

1.2. Project structure

The project is organized in the `TSP-solver` folder as shown in Figure 1.

- `data/` : it contains the datasets generated by the script `data_generator.py`. In the subfolder `sol/` it stores all the path solutions for each dataset.
- `plots/` : it contains the plots generated by the script `data_generator.py`. In the subfolder `sol/` it stores all the plots with their path solution.
- `report.pdf` : the written report of the project.
- `scripts/` : it contains the python script used to generate and plot data. They are explained in section 2. Instances generator.
- `src/` : it contains the source files of the c++ project:
  - `main.cpp` : the main file of the project. It runs the CplexSolver and SASolver on two datasets.
  - `makefile` : the makefile used to compile the project.
  - `CplexSolver.cpp/.h` : a TSP solver which uses the CPLEX C API's. It implements the abstract class `TSPSolver` . It is explained in section Part I: Cplex Solver.
  - `cpxmacro.h` : useful macros of the CPLEX C API's.
  - `Point.cpp/.h` : a class representing a single point.
  - `SASolver.cpp/.h` : a TSP solver which uses the Simulated Annealing algorith. It implements the abstract class `TSPSolver` . It is explained in section Part II: Simulated Annealing.
  - `TSP.cpp/.h` : a TSP representation of the problem. It loads points from a dataset and calculates the matrix costs.

- `TSPSolution.cpp/.h` : a representation of a path solution to a TSP problem.
- `TSPSolver.cpp/.h` : an abstract class representing a solver for the TSP.

```
TSP-solver/
├── data/
│   ├── sol/
│   │   ├── Path_TSP_10_1.dat
│   │   └── ...
│   ├── TSP_10_1.dat
│   └── ...
├── plots/
│   ├── sol/
│   │   ├── Path_TSP_10_1.png
│   │   └── ...
│   ├── TSP_10_1.png
│   └── ...
├── report.pdf
├── scripts/
│   ├── data_generator.py
│   └── plot_solutions.py
└── src/
    ├── main.cpp
    ├── makefile
    ├── CplexSolver.cpp
    ├── CplexSolver.h
    ├── cpxmacro.h
    ├── point.cpp
    ├── point.h
    ├── SASolver.cpp
    ├── SASolver.h
    ├── TSP.cpp
    ├── TSP.h
    ├── TSPSolution.cpp
    ├── TSPSolution.h
    ├── TSPSolver.cpp
    └── TSPSolver.h
```

Figure 1: Project structure

1.3. Compilation

The project is written in C++ and compiles using the makefile provided in the `src` folder. To compile the project, use the command `make` in the `src/` folder. The makefile compiles the project using the `g++` compiler and produces an executable named `main`. The project uses the C++11 standard, so it is necessary to use a compiler that supports this standard.

It is possible to clean the project using the command `make clean`. This command removes the executable and all the object files.

The python scripts provided in the `scripts` folder and used to generate and plot data can be run using the command `python3 script_name.py`. The scripts require the `matplotlib` and `numpy` libraries to be installed. The libraries can be easily installed using the command `pip install matplotlib` and `pip install numpy`.

Note that any re run of the scripts will overwrite the existing data and plots. Since the data is generated randomly, the results may vary between runs.

All the tests have been made on the LabTA calculators from remote, on Ubuntu Operative System.

2. Instances generator

The instances were generated with the script `data_generator.py`. The script considers problems with 10, 25, 50, 75, and 100 points. For each problem size, it generates 9 examples. The number of examples is chosen considering the second part of the exercise: a significant portion of the training samples (66%) will be used to find the best parameters for the searches, while the remaining part will be used for testing.

To accurately recreate the problem description, the datasets are generated with specific criteria. Points are placed in shapes such as rectangles, squares, and circles. Shapes are picked randomly at each iteration. Additionally, there are points considered to be holes for bolts: there are at least 4 in each dataset, placed near the corners of the board.

Points are generated randomly according to a shape, but the script ensures that the distance between points is at least 0.3. This is done to avoid points being too close to each other, which could lead to numerical instability in the solution, particularly when subtracting numbers very close to each other.

The script also ensures that shapes do not overlap. Before adding a new shape, it checks for overlaps with existing shapes using specific functions:

- `check_overlap()` for circles
- `check_rectangle_overlap()` for rectangles and squares
- `check_circle_rectangle_overlap()` for checking overlaps between circles and rectangles

The data is stored in a `.dat` file, named `TSP_XX_X.dat`. The first line contains the number of nodes generated, and each subsequent line stores the coordinates of a point. The datasets are stored in the `/data` folder.

The script generates also plots to visualize data using the function `save_plot_as_file()`, which are stored in the `/plot` folder. Some examples can be seen in Figure 2 and Figure 3.

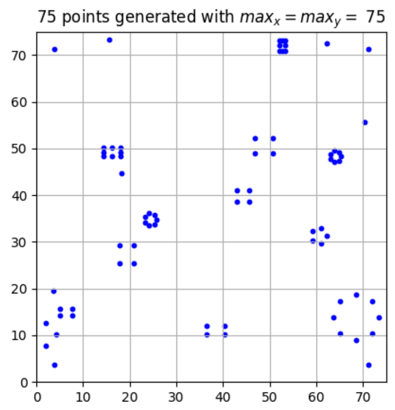

Figure 2: Instance with 75 points

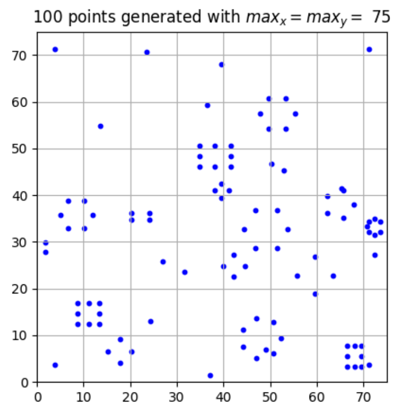

Figure 3: Instance with 100 points

To visualize solutions, the script `plot_solutions.py` plots the dataset with the path solution. In particular, it reads data from a `.dat` file containing the dataset with the points coordinates placed in the `data/` folder and from another `.dat` file the path solution, placed in the `data/sol/` folder. Some examples can be visualized in Figure 4 and Figure 5

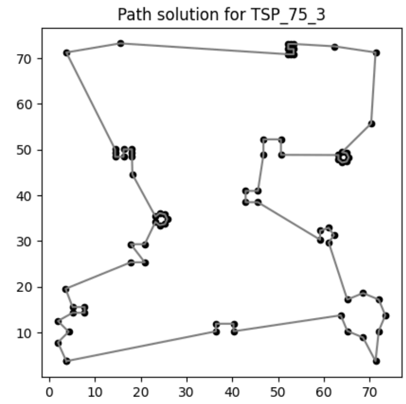

Figure 4: Optimal path with 75 points

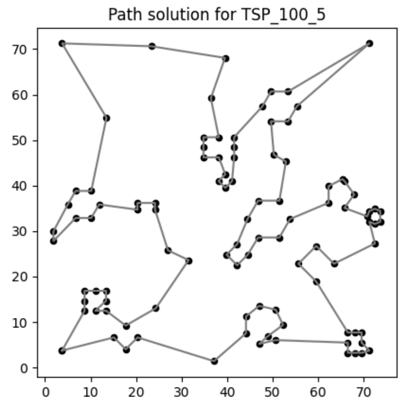

Figure 5: Optimal path with 100 points

3. Part I: Cplex Solver

This section details the implementation of the TSP model using the CPLEX API for C++, in the class `CplexSolver` .

3.1. MILP Model

Sets:

- N = graph nodes, representing the holes;
- A = arcs $(i, j), \forall i, j \in N$, representing the trajectory covered by the drill to move from hole i to hole j

Parameters:

- c_{ij} = time taken by the drill to move from i to j, $\forall (i, j) \in A$
- 0 = arbitrarily selected starting node, $0 \in N$.

Decision Variables:

- x_{ij} = amount of the flow shipped from i to j, $\forall (i, j) \in A$;
- y_{ij} = 1 if arc(i, j) ships some flow, 0 otherwise, $\forall (i, j) \in A$

Integer Linear Programming model:

$$
\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in A} c_{ij} y_{ij} && && (1)\\
s.t. \quad & \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} && = 1 && \forall k \in N \setminus \{0\} \quad (2)\\
& \sum_{j:(i,j)\in A} y_{ij} && = 1 && \forall i \in N \quad (3)\\
& \sum_{i:(i,j)\in A} y_{ij} && = 1 && \forall j \in N \quad (4)\\
& x_{ij} && \leq (|N| - 1) y_{ij} && \forall (i, j) \in A, j \neq 0 \quad (5)\\
& x_{ij} \in \mathbb{R}_+ && && \forall (i, j) \in A, j \neq 0 \quad (6)\\
& y_{ij} \in \{0, 1\} && && \forall (i, j) \in A \quad (7)
\end{aligned}
$$

3.2. Variables creation

Variables x and y are created using the `CPXnewcols` function with two calls, one for x variables and one for y variables.

To keep a connection between the CPLEX variables representation, indexes are stored in a map, defined as follows:

```
std::map<std::tuple<char, int, int>, int> vars_map;
```

The `CplexSolver` class implements the following private method to store the indexes in the map, where `type` is the variable type, `i` and `j` are the indexes of the variable and `cur_index` is the current index of the variable in the CPLEX model:

```
void add_index_to_map(char type, int i, int j, int cur_index)
```

To retrieve the CPLEX index of a variable, the following private method is implemented:

```
int get_index_from_map(char type, int i, int j)
```

To efficiently insert variables in the CPLEX model:

1. vectors are created to define: variable types, lower and upper bounds, objective coefficients and variable names. For each variable, the corresponding index is stored in the map using the `add_index_to_map()` method.
2. The `CPXnewcols` function is called once to add all y variables and then all x variables to the CPLEX problem. This is more efficient than adding variables one by one.
3. Memory cleanup is performed: the allocated space for variables names (pointers to string) is released after the variables are added to the CPLEX model.

3.3. Constraints creation

Similarly to the variables creation, the row constraints (2), (3), (4), (5) are added in a single call for each one of them: the `CPXaddrows` function is called once for each constraint type, adding all the constraints of that type to the CPLEX model.

To efficiently insert constraints in the CPLEX model.

1. vectors are initialized to define: right-hand side values, the sense of the constraints, the indexes of the variables involved, and their coefficients.
2. The `CPXaddrows` function is called once to add all the constraints at once. This is more efficient than adding constraints one by one.
3. Memory cleanup is performed: the allocated space for the constraints names (pointers to string) is released after the constraints are added to the CPLEX model.

Let's take a look at the values of the vectors used to add the constraints (2), (3), (4), (5) to the CPLEX model.

Constraint (2)

$$\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \qquad \forall k \in N \setminus \{0\}$$

- `rowcnt = n - 1` is the number of constaints of this type. In fact we are considering all the nodes except the starting one.
- `nzcnt = rowcnt * (2*n - 3)` is the number of non-zero coefficients in the constraint matrix. In fact, for each node k we have $n-1$ variables x_{ik} and $n-2$ variables x_{kj}, because $j \neq 0$.
- `vector <double> rhs(rowcnt, 1)` is the right-hand side values of the constraints. In this case, all the values are set to 1.
- `vector <char> sense(rowcnt, 'E')` is the sense of the constraints. In this case, all the constraints are equalities, identified by char 'E'.
- `vector <int> rmatbeg(rowcnt)` is the starting index of the non-zero coefficients in the constraint matrix. In this case, the starting index of each row is given by $k*(2*n-3)$: in fact, at iteration k, we have already added $k*(n-1)$ variables x_{ik} and $k*(n-2)$ variables x_{kj}.
- `vector <int> rmatind(nzcnt)` is the indexes of the variables involved in the constraints. The indexes are stored in the vector at each iteration using the `get_index_from_map()` method. An internal counter is used to keep track of the current index in the vector.

- `vector <double> rmatval(nzcnt, 0)` is the coefficients of the variables involved in the constraints. The coefficients are set to 1 for the variables x_{ik} and -1 for the variables x_{kj}. As it happens for the indexes, the coefficients are stored in the vector using an internal counter to keep track of the current index.
- `vector <char*> rowname(rowcnt)` is the names of the constraints. The names are set using the `cpxString()` method which returns a pointer to a string after allocating the necessary space.

Constraint (3)

$$\sum_{j:(i,j)\in A} y_{ij} = 1 \qquad\qquad \forall i \in N$$

- `rowcnt = n` , in fact we are creating a constraint for each node $i \in N$.
- `nzcnt = rowcnt * n` , in fact, for each row i we are summing n variables y.
- `vector <double> rhs(rowcnt, 1)` : in this case, all the right hand sides are set to 1.
- `vector <char> sense(rowcnt, 'E')` all the constraints are equalities, identified by char 'E'.
- `vector <int> rmatbeg(rowcnt)`: in this case, the starting index of each row i is given by $i * n$: in fact, at iteration i, we have already added $i * n$ variables y_{ij}.
- `vector <int> rmatind(nzcnt)` the indexes are stored in the vector at each iteration using `get_index_from_map()` , in particular the variable y_{ij} is in the position $i * n + j$ of this vector.
- `vector <double> rmatval(nzcnt, 1)` the coefficients are set to 1 for all the variables y_{ij} .
- `vector <char*> rowname(rowcnt)` : at each iteration i a new pointer to a string is added to this vector.

Constraint (4)

$$\sum_{i:(i,j)\in A} y_{ij} = 1 \qquad\qquad \forall j \in N$$

- `rowcnt = n` , in fact we are creating a constraint for each node $j \in N$.
- `nzcnt = rowcnt * n` , in fact, for each row j we are summing n variables y.
- `vector <double> rhs(rowcnt, 1)` : in this case, all the right hand sides are set to 1.
- `vector <char> sense(rowcnt, 'E')` all the constraints are equalities, identified by char 'E'.
- `vector <int> rmatbeg(rowcnt)`: the starting index of each row j is given by $j * n$: in fact, at iteration j, we have already added $j * n$ variables y_{ij}.
- `vector <int> rmatind(nzcnt)` the indexes are stored in the vector at each iteration using `get_index_from_map()` , in particular the variable y_{ij} is in the position $j * n + i$ of this vector.
- `vector <double> rmatval(nzcnt, 1)` the coefficients are set to 1 for all the variables y_{ij} .

- `vector <char*> rowname(rowcnt)` : at each iteration j a new pointer to a string is added to this vector.

Constraint (5)

$$x_{ij} \leq (|N| - 1)y_{ij} \qquad \forall (i, j) \in A, j \neq 0$$

Note that the constraint is added into the CPLEX model moving all the variables on the left-hand side:

$$x_{ij} \leq (|N| - 1)y_{ij} \quad \Rightarrow \quad x_{ij} - (|N| - 1)y_{ij} \leq 0$$

- `rowcnt = n * (n - 1)` , in fact we are creating a constraint for each pair of arc, except for the ones to the starting node.
- `nzcnt = 2 * rowcnt` , in fact, for each row ij we are summing two variables.
- `vector <double> rhs(rowcnt, 0)` : all the right hand sides are set to 0.
- `vector <char> sense(rowcnt, 'L')` all the constraints are less or equal, identified by char 'L'.
- `vector <int> rmatbeg(rowcnt)`: the starting index of each row constraint ij is given by $2 * row$, where row is an internal counter to keep track of the number of row constraints added.
- `vector <int> rmatind(nzcnt)` the indexes are stored in the vector at each iteration using `get_index_from_map()` , in particular, at a given row the variable x_{ij} is in position $row * 2$ and the variable y_{ij} is in position $row * 2 + 1$.
- `vector <double> rmatval(nzcnt)` the coefficients are set to 1 for all the variables x_{ij}, so the ones in position $row * 2$, and to $-(|N| - 1)$ for all the variables y_{ij}, so the ones in position $row * 2 + 1$.
- `vector <char*> rowname(rowcnt)` : at each iteration a new pointer to a string is added to this vector.

3.4. Optimal path

The optimal path is retrieved from the CPLEX model after the optimization process is completed. Using the CPLEX function `CPXgetx()` the values of the variables are stored in a private member of the `CplexSolver` class: `vector<double> varvals` .
Then, the optimal path is reconstructed by following the values of the variables y_{ij} using the private method `createPathSolution()` . It starts by setting the initial node as node 0 and adding it to the path. Then, it iteratively explores the subsequent nodes by checking, for each possible destination j , whether the variable y_{ij} indicates an active connection (when it is set to 1). When it finds a connected node j , it adds j to the path, updates the current node, and continues until all nodes have been visited. Once the tour is complete, it adds the starting node to the end of the path to represent the return to the starting point. In the end, the optimal path is stored in the `vector<int> path_solution` variable as an ordered sequence of nodes.

The optimal path is then returned by the public method `getPathSolution()` . This method is used to initialize a `TSPSolution` object, which is then used to print the path solution, like that:

```
>> Path solution: < 0 6 7 2 4 8 5 3 1 9 0 >
```

3.5. Tests

All the tests have been made in the LabTA calculators from remote, using Ubuntu operative system. The time limits were set to 1, 10, 30, 60, 120, 300 seconds. A cycle was made to collect the results of the tests and store them in a csv file. The results are shown in the following Table 1 and Table 2 figures.

Note that it was not possible to find an optimal path solution for the dataset `TSP_100_2.dat` within the time limit of 300 seconds.

Dataset	Problem size	Time limit	Solving Time	Status	Objetive value
`TSP_10_1.dat`	10	1 sec	0.2577 sec	Optimal	42.1044
`TSP_10_2.dat`	10	1 sec	0.2497 sec	Optimal	40.5850
`TSP_10_3.dat`	10	1 sec	0.2306 sec	Optimal	39.9835
`TSP_10_4.dat`	10	1 sec	0.2850 sec	Optimal	40.8729
`TSP_10_5.dat`	10	1 sec	0.2702 sec	Optimal	39.6283
`TSP_10_6.dat`	10	1 sec	0.2279 sec	Optimal	39.2479
`TSP_10_7.dat`	10	1 sec	0.2481 sec	Optimal	41.6571
`TSP_10_8.dat`	10	1 sec	0.3347 sec	Optimal	41.3277
`TSP_10_9.dat`	10	1 sec	0.2816 sec	Optimal	41.7131
`TSP_25_1.dat`	25	1 sec	1.0065 sec	Feasible	118.4207
`TSP_25_1.dat`	25	10 sec	1.4000 sec	Optimal	117.5848
`TSP_25_2.dat`	25	1 sec	0.7831 sec	Optimal	114.5453
`TSP_25_3.dat`	25	1 sec	1.0128 sec	Feasible	116.3888
`TSP_25_3.dat`	25	10 sec	1.2669 sec	Optimal	116.3888
`TSP_25_4.dat`	25	1 sec	0.9641 sec	Optimal	116.8552
`TSP_25_5.dat`	25	1 sec	1.0213 sec	Feasible	116.2516
`TSP_25_5.dat`	25	10 sec	1.0392 sec	Optimal	114.5004
`TSP_25_6.dat`	25	1 sec	1.0104 sec	Feasible	137.8108
`TSP_25_6.dat`	25	10 sec	1.6192 sec	Optimal	109.9704
`TSP_25_7.dat`	25	1 sec	0.6487 sec	Optimal	113.8732
`TSP_25_8.dat`	25	1 sec	0.3907 sec	Optimal	109.9128

Dataset	Problem size	Time limit	Solving Time	Status	Objetive value
`TSP_25_9.dat`	25	1 sec	0.6006 sec	Optimal	124.9924
`TSP_50_1.dat`	50	1 sec	1.0174 sec	Feasible	196.0226
`TSP_50_1.dat`	50	10 sec	4.2766 sec	Optimal	146.0800
`TSP_50_2.dat`	50	1 sec	1.0107 sec	Feasible	419.2273
`TSP_50_2.dat`	50	10 sec	8.1265 sec	Optimal	144.3220
`TSP_50_3.dat`	50	1 sec	1.0087 sec	Feasible	464.6008
`TSP_50_3.dat`	50	10 sec	6.5681 sec	Optimal	146.3909
`TSP_50_4.dat`	50	1 sec	1.0131 sec	Feasible	181.4801
`TSP_50_4.dat`	50	10 sec	6.0332 sec	Optimal	138.1332
`TSP_50_5.dat`	50	1 sec	1.0103 sec	Feasible	356.3201
`TSP_50_5.dat`	50	10 sec	5.8066 sec	Optimal	136.5960
`TSP_50_6.dat`	50	1 sec	1.0116 sec	Feasible	478.9975
`TSP_50_6.dat`	50	10 sec	5.5610 sec	Optimal	145.1683
`TSP_50_7.dat`	50	1 sec	1.0133 sec	Feasible	498.5362
`TSP_50_7.dat`	50	10 sec	10.0246 sec	Feasible	130.2013
`TSP_50_7.dat`	50	30 sec	12.1113 sec	Optimal	130.2013
`TSP_50_8.dat`	50	1 sec	1.0123 sec	Feasible	467.8946
`TSP_50_8.dat`	50	10 sec	5.6500 sec	Optimal	135.8707
`TSP_50_9.dat`	50	1 sec	1.0136 sec	Feasible	469.0935
`TSP_50_9.dat`	50	10 sec	5.8418 sec	Optimal	134.6858
`TSP_75_1.dat`	75	1 sec	1.0097 sec	Feasible	1459.4778
`TSP_75_1.dat`	75	10 sec	10.0514 sec	Feasible	507.3574
`TSP_75_1.dat`	75	30 sec	30.0223 sec	Feasible	450.7407
`TSP_75_1.dat`	75	60 sec	37.2917 sec	Optimal	450.5143
`TSP_75_2.dat`	75	1 sec	1.0155 sec	Feasible	1150.0906
`TSP_75_2.dat`	75	10 sec	10.0691 sec	Feasible	478.0992

Dataset	Problem size	Time limit	Solving Time	Status	Objetive value
`TSP_75_2.dat`	75	30 sec	30.0389 sec	Feasible	474.8855
`TSP_75_2.dat`	75	60 sec	56.4847 sec	Optimal	418.8488
`TSP_75_3.dat`	75	1 sec	1.0196 sec	Feasible	1206.3164
`TSP_75_3.dat`	75	10 sec	10.1449 sec	Feasible	571.3963
`TSP_75_3.dat`	75	30 sec	30.0226 sec	Feasible	534.4228
`TSP_75_3.dat`	75	60 sec	51.6793 sec	Optimal	389.0620
`TSP_75_4.dat`	75	1 sec	1.0104 sec	Feasible	1319.4935
`TSP_75_4.dat`	75	10 sec	10.0252 sec	Feasible	552.9256
`TSP_75_4.dat`	75	30 sec	30.0244 sec	Feasible	451.1586
`TSP_75_4.dat`	75	60 sec	60.0304 sec	Feasible	444.8608
`TSP_75_4.dat`	75	120 sec	63.7516 sec	Optimal	444.3585
`TSP_75_5.dat`	75	1 sec	1.0107 sec	Feasible	1158.4734
`TSP_75_5.dat`	75	10 sec	10.0316 sec	Feasible	533.9445
`TSP_75_5.dat`	75	30 sec	30.0391 sec	Feasible	458.2926
`TSP_75_5.dat`	75	60 sec	34.0651 sec	Optimal	458.2926
`TSP_75_6.dat`	75	1 sec	1.0139 sec	Feasible	1591.8847
`TSP_75_6.dat`	75	10 sec	10.0198 sec	Feasible	577.7397
`TSP_75_6.dat`	75	30 sec	30.0305 sec	Feasible	577.7397
`TSP_75_6.dat`	75	60 sec	52.7740 sec	Optimal	475.2658
`TSP_75_7.dat`	75	1 sec	1.0251 sec	Feasible	1492.1890
`TSP_75_7.dat`	75	10 sec	10.0249 sec	Feasible	623.1681
`TSP_75_7.dat`	75	30 sec	30.0311 sec	Feasible	417.7575
`TSP_75_7.dat`	75	60 sec	40.6911 sec	Optimal	415.6373
`TSP_75_8.dat`	75	1 sec	1.0429 sec	Feasible	1341.8736
`TSP_75_8.dat`	75	10 sec	10.0205 sec	Feasible	611.0727
`TSP_75_8.dat`	75	30 sec	30.0282 sec	Feasible	445.3625

Dataset	Problem size	Time limit	Solving Time	Status	Objetive value
`TSP_75_8.dat`	75	60 sec	51.7496 sec	Optimal	417.5275
`TSP_75_9.dat`	75	1 sec	1.0298 sec	Feasible	1332.3518
`TSP_75_9.dat`	75	10 sec	10.0193 sec	Feasible	581.9756
`TSP_75_9.dat`	75	30 sec	30.0373 sec	Feasible	449.0168
`TSP_75_9.dat`	75	60 sec	43.3595 sec	Optimal	449.0168
`TSP_100_1.dat`	100	1 sec	1.0147 sec	Feasible	1859.4852
`TSP_100_1.dat`	100	10 sec	10.0569 sec	Feasible	812.8012
`TSP_100_1.dat`	100	30 sec	31.5660 sec	Feasible	575.2419
`TSP_100_1.dat`	100	60 sec	60.0495 sec	Feasible	551.7561
`TSP_100_1.dat`	100	120 sec	120.0306 sec	Feasible	517.1964
`TSP_100_1.dat`	100	300 sec	121.7011 sec	Optimal	517.1964
`TSP_100_2.dat`	100	1 sec	1.0151 sec	Feasible	1579.5440
`TSP_100_2.dat`	100	10 sec	10.0158 sec	Feasible	836.5979
`TSP_100_2.dat`	100	30 sec	30.1112 sec	Feasible	836.5979
`TSP_100_2.dat`	100	60 sec	60.2576 sec	Feasible	504.9482
`TSP_100_2.dat`	100	120 sec	120.0497 sec	Feasible	479.1978
`TSP_100_2.dat`	100	300 sec	300.0476 sec	Feasible	473.2960
`TSP_100_3.dat`	100	1 sec	1.0222 sec	Feasible	1784.1474
`TSP_100_3.dat`	100	10 sec	10.1491 sec	Feasible	910.8624
`TSP_100_3.dat`	100	30 sec	30.0316 sec	Feasible	910.2184
`TSP_100_3.dat`	100	60 sec	60.0432 sec	Feasible	910.2184
`TSP_100_3.dat`	100	120 sec	120.0332 sec	Feasible	571.8121
`TSP_100_3.dat`	100	300 sec	184.6020 sec	Optimal	491.1374
`TSP_100_4.dat`	100	1 sec	1.0118 sec	Feasible	1592.8707
`TSP_100_4.dat`	100	10 sec	10.0222 sec	Feasible	658.7710
`TSP_100_4.dat`	100	30 sec	30.5145 sec	Feasible	609.9025

Dataset	Problem size	Time limit	Solving Time	Status	Objetive value
`TSP_100_4.dat`	100	60 sec	60.0478 sec	Feasible	506.5066
`TSP_100_4.dat`	100	120 sec	120.0308 sec	Feasible	506.1950
`TSP_100_4.dat`	100	300 sec	140.4908 sec	Optimal	480.4839
`TSP_100_5.dat`	100	1 sec	1.0377 sec	Feasible	1458.2235
`TSP_100_5.dat`	100	10 sec	10.0210 sec	Feasible	834.0529
`TSP_100_5.dat`	100	30 sec	31.1143 sec	Feasible	823.6652
`TSP_100_5.dat`	100	60 sec	60.0387 sec	Feasible	555.5668
`TSP_100_5.dat`	100	120 sec	120.2293 sec	Feasible	555.3637
`TSP_100_5.dat`	100	300 sec	144.6668 sec	Optimal	541.8146
`TSP_100_6.dat`	100	1 sec	1.0195 sec	Feasible	1534.8214
`TSP_100_6.dat`	100	10 sec	10.4829 sec	Feasible	687.1978
`TSP_100_6.dat`	100	30 sec	30.0390 sec	Feasible	686.8297
`TSP_100_6.dat`	100	60 sec	60.0564 sec	Feasible	603.9285
`TSP_100_6.dat`	100	120 sec	120.3188 sec	Feasible	536.9845
`TSP_100_6.dat`	100	300 sec	191.0833 sec	Optimal	477.2790
`TSP_100_7.dat`	100	1 sec	1.0361 sec	Feasible	1792.4031
`TSP_100_7.dat`	100	10 sec	10.1240 sec	Feasible	887.8424
`TSP_100_7.dat`	100	30 sec	30.0509 sec	Feasible	797.0106
`TSP_100_7.dat`	100	60 sec	60.0497 sec	Feasible	524.2325
`TSP_100_7.dat`	100	120 sec	120.0249 sec	Feasible	511.9271
`TSP_100_7.dat`	100	300 sec	200.8718 sec	Optimal	502.4607
`TSP_100_8.dat`	100	1 sec	1.0200 sec	Feasible	1870.0924
`TSP_100_8.dat`	100	10 sec	10.0270 sec	Feasible	992.9269
`TSP_100_8.dat`	100	30 sec	30.0370 sec	Feasible	992.9269
`TSP_100_8.dat`	100	60 sec	60.0659 sec	Feasible	577.8256
`TSP_100_8.dat`	100	120 sec	88.9921 sec	Optimal	478.1800

Dataset	Problem size	Time limit	Solving Time	Status	Objetive value
`TSP_100_9.dat`	100	1 sec	1.0152 sec	Feasible	1996.9046
`TSP_100_9.dat`	100	10 sec	10.0219 sec	Feasible	740.6538
`TSP_100_9.dat`	100	30 sec	30.0269 sec	Feasible	740.6538
`TSP_100_9.dat`	100	60 sec	60.0634 sec	Feasible	740.6538
`TSP_100_9.dat`	100	120 sec	120.0594 sec	Feasible	629.1568
`TSP_100_9.dat`	100	300 sec	201.8372 sec	Optimal	503.1572

Table 1: Cplex API Benchmarks

The average performances for the optimal path setup and solving time are shown in the following Table 2.

Problem size	Avg. time for setup	Avg. time for solving
10 nodes	0.00074 sec	0.265 sec
25 nodes	0.00181 sec	0.963 sec
50 nodes	0.01511 sec	6.857 sec
75 nodes	0.02768 sec	48.101 sec
100 nodes	0.03779 sec	141.551 sec

Table 2: Cplex API average performances

4. Part II - Simulated Annealing

This section details the implementation of the TSP model using the Simulated Annealing algorithm, in the `SASolver` class. Simulated Annealing is a metaheuristic algorithm used to find the global optimum of a function. As the number of nodes in the TSP increases, the time required by the CPLEX solver to find the optimal solution also increases significantly. For this reason, a metaheuristic algorithm can provide a good solution within a reasonable time.

4.1. Implementation

The `SASolver` class is responsible for solving the TSP using the Simulated Annealing algorithm. In order to create a new instance of the solver, the user must provide the following parameters: the maximum number of iterations for each multistart, the cooling parameter, the initial temperature, the maximum number of non-improving iterations before stopping the algorithm. The algorithm is implemented in the `solve()` method, which takes as input the TSP instance and returns the best solution found.

The algorithm starts by generating an initial solution. Then, it iteratively generates new solutions by choosing a neighbor of the current solution and evaluating its cost. The new solution is accepted with a probability that depends on the difference in cost between the current and new solution and a temperature parameter that decreases over time. The algorithm follows a multirestart approach, meaning that it is executed multiple times with different initial solutions to improve the chances of finding a high-quality solution. It stops when the temperature reaches zero or a predefined number of iterations is reached. The best solution found across all runs is returned as the final solution.

4.1.1. Initial solution

Based on the number of iterations of multistart, different initial solutions are generated. The chosen number for the multistart is 5, so the algorithm will generate 5 different initial solutions. This approach is used to improve the chances of finding a high-quality solution. The value 5 represents a trade-off between small and large TSP instances. For smaller instances, too many multistarts would be unnecessary since the solution space is limited, and each SA run converges quickly. Conversely, for larger instances, multiple starting points help escape poor local optima. A proper tuning of the algorithm's parameters should ideally include optimizing the number of multistarts as well. However, due to time constraints, I have chosen to focus on tuning other parameters instead.

The initial solution, of the first iteration of the algorithm, is generated using the Nearest Neighbour heuristic. This heuristic uses a greedy approach: it starts from node 0 and iteratively selects the nearest unvisited node until all nodes are visited. The time complexity of this approach is $O(n^2)$. The method that is responsible for the Nearest Neighbour Initial solution is `initialNeighborSolution()`.

The second multistart of the algorithm uses the initial generated by `CheapestInsertionSolution()` method: starting from a tour with two nodes $\{0, 1, 0\}$, the algorithm iteratively selects the unvisited node that result in the minimum addiotional cost of inserting it in the tour. The time complexity of this implemented approach is $O(n^3)$ in the worst case.

The remaining multistarts of the algorithm use a random initial solution. The method that is responsible for the random initial solution is `randomize()`, of the `TSPSolution` class.

This approach allows the algorithm to explore different regions of the solution space: the Nearest Neighbour heuristic and the Cheapest Insertion heuristic provide a good starting point, while the random initial solution allows the algorithm to identify different local optima. In general, as said by the professor during the lectures, no evidences attest that better solutions are obtained starting from better initial solutions, however i chose to implement this approach in order to differentiate the origianl Simulated Annealing algorithm.

Note that the two heuristics were found online searching for TSP heuristics[1], however the implementations are made by myself.

4.1.2. Neighbour generation

The neighbourhood of a solution is defined as the set of solutions that can be obtained by applying the following moves:

- 2-opt move: reverse the order of the nodes between two randomly selected nodes. It is implemented in the `twoOptMove()` method; in order to evaluate the cost of the new solution, an override of the `evaluate()` method is used: given the two selected nodes i and j in the sequence $< 1...h, i, ..., j, l, ..., 1$, the method calculates the new costs of the solution using the following formula: $c_{\text{new}} = c_{\text{old}} - c_{hi} - c_{jl} + c_{hj} + c_{il}$
- 3-opt move: reverse the order of the nodes between three randomly selected nodes. It is implemented in the `threeOptMove()` method; in order to evaluate the cost of the new solution the standard evaluation method is used: starting from the initial node 0, the method calculates the cost of the new solution by summing the costs of the edges between the nodes in the sequence.
- swap move: swap the position of two randomly selected nodes. It is implemented in the `swapMove()` method; in order to evaluate the cost of the new solution the standard evaluation method is used, as for the 3-opt move.

At each iteration, one of the moves is selected at random and applied to the current solution to produce a neighboring solution. The method responsible for generating the new solution is `generateNeighbor()`. The moves have different probabilities of being selected: the distribution is based on the effectiveness of the 2-opt move in improving solutions, while the 3-opt and swap moves enhance exploration by escaping local optima. For large TSP instances, the 3-opt and swap moves are particularly useful, as they facilitate greater diversification in the solution space, while the 2-opt move is more effective for small instances. The probabilities of selecting the moves are defined in the `generateNeighbor()` method. Due to time constraints, I chose to set the probability of selecting the swap move to 0.4 and to fine tune the probabilities of selecting the 2-opt and 3-opt moves.

4.1.3. Cooling schedule

The temperature parameter is used to control the acceptance of new solutions. The temperature decreases over time according to a cooling parameter. The cooling schedule is th original one proposed by Kirkpatrick (1983): $T_{k+1} = \alpha * T_k$, where T_k is the temperature at iteration k and α is the cooling parameter. The temperature is initialized to a high value and decreases gradually to zero. The temperature is decreased at the end of each iteration.

[1] *Some Important Heuristics for TSP, 2006*

4.1.4. Stopping criteria

The algorithm stops when a predefined number of iterations is reached, depending on the problem size.

4.2. Parameters tuning

The Simulated Annealing algorithm has several parameters that can be tuned to improve its performance.

As said before in the section 2. Instances generator, the training set is composed of 66% of the generated instances, while the remaining 33% is used for testing. The training set is used to find the best parameters for the Simulated Annealing algorithm. For each problem size, instances from 1 to 6 are used for training, while instances from 7 to 9 are used for testing. The algorithm is executed five times for each instance and the results are reported in different tables.

In general, the goal was to find the best parameters for the Simulated Annealing algorithm that would allow it to find a solution closer to the optimal one in a reasonable amount of time.

The parameters that are tuned are:

- N_{max}: the maximum number of iterations for each multistart
- α: the cooling parameter
- T: the initial temperature
- N_{bad}: the maximum number of non-improving iterations before stopping the algorithm
- P_{2opt}: the probability of selecting the 2-opt move at each iteration
- P_{3opt}: the probability of selecting the 3-opt move at each iteration

Due to the exponential growth in the number of tests required, an exhaustive parameter search was not feasible. Instead, a set of empirical tests was performed, and the best-performing combination of parameters is reported along with the corresponding results. In general the following considerations were made:

- N_{max}: the maximum number of iterations for each multistart should be proportional to the problem size. A higher number of iterations would allow the algorithm to explore the solution space more freely, while a lower number would make the algorithm act like a local search. As the number of iterations increases, the computational time also increases.
- α: the cooling parameter influences the convergence speed of the algorithm. For smaller instances, a higher value of alpha would allow the algorithm to converge faster, while for larger instances, a lower value would be more appropriate to allow the algorithm to explore the solution space more freely.
- T: the initial temperature should be proportional to the problem size. A lower temperature would make the algorithm act like a local search, accepting only improving solutions, while a higher temperature would allow the algorithm to explore the solution space more freely but with a higher probability of accepting worsening solutions.
- N_{bad}: the maximum number of non-improving iterations before stopping the algorithm should be proportional to the problem size. For bigger instances, an higher number of

non-improving iteration would allow the algorithm to find better solutions, because the solution space is larger and the probabilities of improving solutions are lower.

- P_{2opt}: is the the best move for small instances, because it is the most effective in improving solutions, because it is the most effective in improving solutions.
- P_{3opt}: is the best move for larger instances, because it allows the algorithm to escape local optima and explore the solution space more freely.

4.2.1. TSP size: 10

N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
500	0.99	12	200	0.2	0.4

Training					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_10_1.dat`	42.1044	42.1044	42.1044	0.0000	0.1421 sec
`TSP_10_2.dat`	40.5850	40.9087	40.6498	0.1295	0.1611 sec
`TSP_10_3.dat`	39.9835	39.9835	39.9835	0.0000	0.1464 sec
`TSP_10_4.dat`	40.8729	41.1680	40.9319	0.1180	0.1157 sec
`TSP_10_5.dat`	39.6283	39.6283	39.6283	0.0000	0.1793 sec
`TSP_10_6.dat`	39.2479	39.2479	39.2479	0.0000	0.1595 sec
Testing					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_10_7.dat`	41.6571	41.6571	41.6571	0.0000	0.1521 sec
`TSP_10_8.dat`	41.3277	41.3277	41.3277	0.0000	0.1334 sec
`TSP_10_9.dat`	41.7131	42.0401	41.8439	0.1602	0.1627 sec

Table 3: SA performances for TSP size 10

4.2.2. TSP size: 25

N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
1000	0.96	50	4000	0.15	0.45

Training					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_25_1.dat`	121.0208	127.7170	124.6791	2.1436	0.4801 sec
`TSP_25_2.dat`	116.9371	120.4846	118.5675	1.2130	0.5295 sec
`TSP_25_3.dat`	119.5117	124.4823	122.4935	2.4134	0.4263 sec
`TSP_25_4.dat`	118.3463	124.6497	120.9251	2.1064	0.5039 sec
`TSP_25_5.dat`	118.2204	120.7781	119.4260	0.8774	0.5577 sec

`TSP_25_6.dat`	112.7662	114.5473	113.9544	0.6400	0.5026 sec
Testing					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_25_7.dat`	114.5640	125.2926	118.6536	3.8626	0.4637 sec
`TSP_25_8.dat`	109.9128	120.6628	114.4376	3.7549	0.5222 sec
`TSP_25_9.dat`	131.0302	136.9749	134.3330	2.1988	0.5286 sec

Table 4: SA performances for TSP size 25

4.2.3. TSP size: 50

N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
3000	0.94	100	1000	0.15	0.45

Training					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_50_1.dat`	158.2897	167.0534	161.9167	2.9616	3.0452 sec
`TSP_50_2.dat`	158.7165	163.5184	161.1484	2.0541	3.3085 sec
`TSP_50_3.dat`	156.1423	164.1950	160.3999	2.8786	3.0735 sec
`TSP_50_4.dat`	149.3260	149.3260	149.3260	0.0000	2.8179 sec
`TSP_50_5.dat`	142.0716	159.0605	148.5610	5.9536	3.2037 sec
`TSP_50_6.dat`	152.0125	153.9758	153.5831	0.7853	3.2483 sec
Testing					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_50_7.dat`	135.9899	143.3034	140.0773	2.8414	3.2215 sec
`TSP_50_8.dat`	146.4733	159.4977	152.7684	5.2280	3.1128 sec
`TSP_50_9.dat`	144.1011	156.2556	151.2625	4.4977	3.2332 sec

Table 5: SA performances for TSP size 50

4.2.4. TSP size: 75

N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
15000	0.92	250	5000	0.1	0.5

Training					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
`TSP_75_1.dat`	481.8080	496.1988	489.0034	7.1954	16.0154 sec
`TSP_75_2.dat`	442.5938	453.9232	448.2585	5.6647	15.5979 sec
`TSP_75_3.dat`	402.5947	448.8334	425.7140	23.1194	17.7486 sec

TSP_75_4.dat	460.0896	477.1074	468.5985	8.5089	17.4065 sec
TSP_75_5.dat	495.8942	501.2556	498.5749	2.6807	16.0858 sec
TSP_75_6.dat	513.6851	556.0474	534.8663	21.1811	18.8757 sec
Testing					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
TSP_75_7.dat	451.2295	455.2584	453.2440	2.0145	15.5708 sec
TSP_75_8.dat	467.2923	477.4068	472.3495	5.0573	16.8591 sec
TSP_75_9.dat	481.4110	503.1316	492.2713	10.8603	15.8453 sec

Table 6: SA performances for TSP size 75

4.2.5. TSP size: 100

N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
25000	0.91	350	10000	0.1	0.5

Training					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
TSP_100_1.dat	562.0855	616.8389	589.4622	27.3767	32.8537
TSP_100_2.dat	517.6495	523.4737	520.5616	2.9121	36.3481
TSP_100_3.dat	538.9852	554.0208	546.5030	7.5178	38.6829
TSP_100_4.dat	511.0467	534.6725	522.8596	11.8129	36.0500
TSP_100_5.dat	604.2694	612.9694	608.6194	4.3500	33.7766
TSP_100_6.dat	524.8286	561.9574	543.3930	18.5644	33.5939
Testing					
Instance	**Best Sol.**	**Worst Sol.**	**Avg.**	**Std. Dev.**	**Avg. Time**
TSP_100_7.dat	557.1537	557.1537	557.1537	0.0000	32.9219
TSP_100_8.dat	518.6339	585.9742	552.3040	33.6702	32.3897
TSP_100_9.dat	568.9786	610.3024	589.6405	20.6619	31.9742

Table 7: SA performances for TSP size 100

4.3. Parameters Recap

Size	N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
10	500	0.99	12	200	0.2	0.4
25	1000	0.96	50	4000	0.15	0.45
50	3000	0.94	100	1000	0.15	0.45
75	15000	0.92	250	5000	0.1	0.5

Size	N_{max}	α	T	N_{bad}	P_{2opt}	P_{3opt}
100	25000	0.91	350	10000	0.1	0.5

Table 8: SA parameters for different TSP size

5. Performances comparison

Table 9 compares the performances of the CPlex solver and the SA algorithm. The table shows that the CPlex implementation is always able to find the optimal solution, while the SA algorithm is not. However, the SA algorithm is able to find a resonable solution in small amount of time, even for the largest instances. The table also shows that the SA algorithm is able to find a solution that is close to the optimal one, with a relative error of less than ≈ 10% for all instances.

Instance	Cplex Sol.	Cplex Solving Time	SA Sol.	SA Solving Time	Gap %	Speedup SA
`TSP_10_1.dat`	42.1044	0.2577	42.1044	0.1421	0.0	1.81
`TSP_10_2.dat`	40.585	0.2497	40.585	0.1611	0.0	1.55
`TSP_10_3.dat`	39.9835	0.2306	39.9835	0.1464	0.0	1.58
`TSP_10_4.dat`	40.8729	0.285	40.8729	0.1157	0.0	2.46
`TSP_10_5.dat`	39.6283	0.2702	39.6283	0.1793	0.0	1.51
`TSP_10_6.dat`	39.2479	0.2279	39.2479	0.1595	0.0	1.43
`TSP_10_7.dat`	41.6571	0.2481	41.6571	0.1521	0.0	1.63
`TSP_10_8.dat`	41.3277	0.3347	41.3277	0.1334	0.0	2.51
`TSP_25_1.dat`	117.5848	1.4	121.0208	0.4801	-2.8392	2.92
`TSP_25_2.dat`	114.5453	0.7831	116.9371	0.5295	-2.0454	1.48
`TSP_25_3.dat`	116.3888	1.2669	119.5117	0.4263	-2.613	2.97
`TSP_25_4.dat`	116.8552	0.9641	118.3463	0.5039	-1.2599	1.91
`TSP_25_5.dat`	114.5004	1.0392	118.2204	0.5577	-3.1467	1.86
`TSP_25_6.dat`	109.9704	1.6192	112.7662	0.5026	-2.4793	3.22
`TSP_25_7.dat`	113.8732	0.6487	114.564	0.4637	-0.603	1.4
`TSP_25_8.dat`	109.9128	0.3907	109.9128	0.5222	0.0	0.75
`TSP_25_9.dat`	124.9924	0.6006	131.0302	0.5286	-4.6079	1.14
`TSP_50_1.dat`	146.08	4.2766	158.2897	3.0452	-7.7135	1.4
`TSP_50_2.dat`	144.322	8.1265	158.7165	3.3085	-9.0693	2.46
`TSP_50_3.dat`	146.3909	6.5681	156.1423	3.0735	-6.2452	2.14
`TSP_50_4.dat`	138.1332	6.0332	149.326	2.8179	-7.4955	2.14
`TSP_50_5.dat`	136.596	5.8066	142.0716	3.2037	-3.8541	1.81
`TSP_50_6.dat`	145.1683	5.561	152.0125	3.2483	-4.5024	1.71
`TSP_50_7.dat`	130.2013	12.1113	135.9899	3.2215	-4.2566	3.76
`TSP_50_8.dat`	135.8707	5.65	146.4733	3.1128	-7.2386	1.82

Instance	Cplex Sol.	Cplex Solving Time	SA Sol.	SA Solving Time	Gap %	Speedup SA
TSP_50_9.dat	134.6858	5.8418	144.1011	3.2332	-6.5338	1.81
TSP_75_1.dat	450.5143	37.2917	481.808	16.0154	-6.4951	2.33
TSP_75_2.dat	418.8488	56.4847	442.5938	15.5979	-5.365	3.62
TSP_75_3.dat	389.062	51.6793	402.5947	17.7486	-3.3614	2.91
TSP_75_4.dat	444.3585	63.7516	460.0896	17.4065	-3.4191	3.66
TSP_75_5.dat	458.2926	34.0651	495.8942	16.0858	-7.5826	2.12
TSP_75_6.dat	475.2658	52.774	513.6851	18.8757	-7.4792	2.8
TSP_75_7.dat	415.6373	40.6911	451.2295	15.5708	-7.8878	2.61
TSP_75_8.dat	417.5275	51.7496	467.2923	16.8591	-10.6496	3.07
TSP_75_9.dat	449.0168	43.3595	481.411	15.8453	-6.729	2.74
TSP_100_1.dat	517.1964	121.7011	562.0855	32.8537	-7.9862	3.7
TSP_100_2.dat	473.296	300.0476	517.6495	36.3481	-8.5682	8.25
TSP_100_3.dat	491.1374	184.602	538.9852	38.6829	-8.8774	4.77
TSP_100_4.dat	480.4839	140.4908	511.0467	36.05	-5.9804	3.9
TSP_100_5.dat	541.8146	144.6668	604.2694	33.7766	-10.3356	4.28
TSP_100_6.dat	477.279	191.0833	524.8286	33.5939	-9.06	5.69
TSP_100_7.dat	502.4607	200.8718	557.1537	32.9219	-9.8165	6.1
TSP_100_8.dat	478.18	88.9921	518.6339	32.3897	-7.8001	2.75
TSP_100_9.dat	503.1572	201.8372	568.9786	31.9742	-11.5683	6.31

Table 9: Performances comparison of SA and CPlex Solver

6. Conclusion

This exercise showed that the TSP is a complex problem that becomes increasingly challenging to solve as its size grows. The CPLEX solver can find the optimal solution for all instances, but it fails to do so within a reasonable amount of time for the largest instances. On the other hand, the Simulated Annealing algorithm can quickly provide high-quality solutions, even for the largest problem sizes. The implemented SA consistently finds solutions within ≈10% of the optimal value, demonstrating that it is a viable alternative to CPLEX for large-scale TSP instances.

The effectiveness of SA heavily depends on parameter tuning, as the results are highly sensitive to the chosen values. A trade-off must be established between solution quality and computational time: in some cases, prioritizing a faster execution may be more important, while in others, achieving the best possible solution within a reasonable timeframe may be preferred. In general, the parameters can be adjusted following the suggestions provided in the section 4.2. Parameter tuning.