Comparison Between a CPLEX-Based and a Proposed Improved Kruskal-Like Heuristic-Based Approach to an Industrial TSP for Electrical Panel Assembly

Andrea Cecchin
Dept. of Mathematics, University of Padua
andrea.cecchin.5@studenti.unipd.it

Abstract

This project investigates the Traveling Salesman Problem (TSP) applied to a real industrial scenario through two different solution strategies. The first is an exact optimization approach based on a mathematical formulation implemented using CPLEX, which guarantees the computation of an optimal tour. The second is a heuristic approach relying on the iteration of a 2-opt algorithm with greedy Kruskal-like initialization, designed to produce an approximated high quality solution in drastically reduced computational time. Both approaches address the objective of minimizing the total travel distance while ensuring that each location is visited exactly once and that the tour returns to the starting point. By comparing these methods, the project highlights the trade-off between optimality and computational efficiency in the solution of TSP instances.

1. Introduction

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 Code

In accordance with the specific assignment requirements, the project and its given code is entirely written in C++. The code for the X-th assignment is located in the `AssX/` folder and can be compiled and run using the provided `makefile`. Specifically, after navigating to that folder, simply run the `make` command, which using g++ will produce an executable named `project`. To run it, run the command `./project *`, where $\star$ indicates the cardinality of the subset of samples for the problem, which are explained in more detail in section 1.3, that we want to test.

1.3. Samples Generation

All dataset instances were generated using a C++ code named `generator`, which is located in the `Ass1/data/generator/` folder. With command `g++ -O2 generator.cpp -o generator` an executable named `generator` will be created, which through the command `./generator.exe` will create the whole samples stored in `data/`.

The dataset created for this problem takes into account instances of different sizes, considering problems with 10, 20, 30, 50, 70, 80 and 100 nodes. For each of these cardinalities, the generator creates 5 different samples, named `instance_X_Y`: `X` indicates the cardinality of the instance, i.e. how many nodes it consists of, while `Y` just represents the identifier for that specific cardinality.

In order to try to recreate cases as close to reality as possible, each single instance has 4 nodes positioned at the edges of the area, which would represent the fixing holes of the electrical panels' boards. Each board is 100x100 mm in size, with the fixing points placed near the four corners, 5 mm from each edge. The other points are generated within the area formed by these four points, completely randomly. Since in a realistic scenario a point cannot be too close or above another one, each new point (the center of the screw) must be at least 3 mm apart. This is based on a scenario where the screws have a radius of 1 mm and there must be at least 1 mm between each screw external margin. Every node is represented as a pair of coordinates in the final `.dat` file. An example is given below.

`instance_10_1.dat`

```
10                      <-- number of nodes
5 5                     <-- fixing point
95 5                    <-- fixing point
95 95                   <-- fixing point
```

```
5 95                    <-- fixing point
40.9352 31.3066         <-- coordinates
94.6107 10.2073
62.0795 75.9289
81.7351 74.336
93.6586 90.2169
52.1632 22.1926
```

Plotting the resulting generated instances, we will obtain samples shaped as follows.

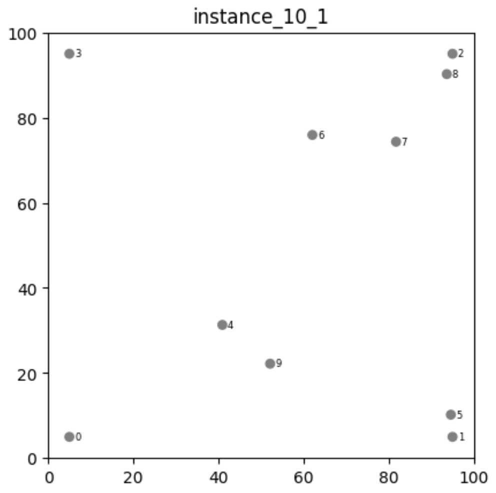

Figure 1. Example of instance with 10 points.

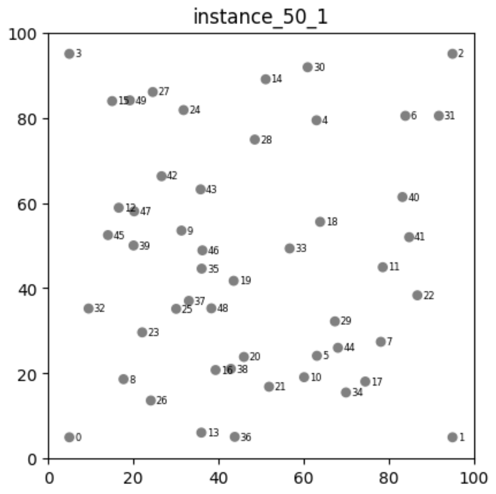

Figure 2. Example of instance with 50 points.

2. CPLEX-Based Implementation

2.1. Mathematical Formulation

The problem formulated in section 1.1 can be easily traslated into a Travelling Salesman Problem (TSP) on the graph represented by board's holes: we can consider a weighted complete graph $G = (N, A)$, where N is the set of nodes and corresponds to the set of the positions where holes have to be made, and A is the set of the arcs $(i, j), \forall i, j \in N$, corresponding to the trajectory of the drill moving from hole i to hole j. Additionaly, a weight c_{ij} can be associated to each arc $(i, j) \in A$, corresponding to the time needed to move from i to j. Since for this project we will assume that the drill is always moving with the same velocity, we can consider the weight c_{ij} just as the distance in mm between i and j.

The final problem can be seen as determining the minimum weight hamiltonian cycle on G, minimizing the distance covered by the drill, and therefore minimizing the drilling time. In order to do this, we apply the Gavish-Graves [2] mathematical formulation for TSP, described as follow.

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.

Decision variables:

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

Integer Linear Programming Model:

$$\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in A} c_{ij}\, y_{ij} && \\
\text{s.t.} \quad & \sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 && \forall k \in N \setminus \{0\} \\
& \sum_{j:(i,j)\in A} y_{ij} = 1 && \forall i \in N \\
& \sum_{i:(i,j)\in A} y_{ij} = 1 && \forall j \in N \\
& x_{ij} - (|N| - 1)\, y_{ij} \leq 0 && \forall (i,j) \in A,\ j \neq 0 \\
& x_{ij} \in R_+ && \forall (i,j) \in A,\ j \neq 0 \\
& y_{ij} \in \{0, 1\} && \forall (i,j) \in A
\end{aligned}$$

2.2. CPLEX Model Setup

2.2.1 Variables Initialization

The variables x and y are instantiated directly in the CPLEX model using the `CPXnewcols` function. In particular, two matrices of integers are used:

- `std::vector<std::vector<int>> map_x`, where `map_x[i][j]` stores the CPLEX index of the x_{ij} variable;

- `std::vector<std::vector<int>> map_y`, where `map_y[i][j]` stores the CPLEX index of the y_{ij} variable.

The variables are defined as follows:

- Continuous variables x_{ij}: represent the flow through node j for $i \neq j$ and $j \neq 0$. Each variable is defined with type 'C' (continuous), lower bound 0, upper bound `CPX_INFBOUND` ($+\infty$) and objective coefficient 0;
- Binary variables y_{ij}: indicate whether the edge (i, j) is selected in the optimal tour, for $i \neq j$. Each variable is defined with type 'B' (binary), lower bound 0, upper bound 1 and objective coefficient equal to the travel cost c_{ij}.

Each variable is added to the CPLEX problem individually with `CPXnewcols`, specifying the variable's name, its type, bounds and objective contribution.

2.2.2 Constraints Formulation

Constraints are inserted into the CPLEX model using `CPXaddrows`. For each constraint, vectors are created to store the involved variable indices and coefficients.

Assignment constraints for y

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

$$\sum_{i:(i,j)\in A} y_{ij} = 1 \qquad \forall j \in N \tag{2}$$

(1) ensures each node has exactly one outgoing edge, while (2) one incoming edge. For each of these constraints, we call `CPXaddrows` using:

- `vector<int> idx`, containing the indices of the variables involved;
- `vector<double> coef`, containing all coefficients set to 1;
- `char sense = 'E'`, to declare it's an equality constraint;
- `double rhs = 1.0`, to setup the right-hand side value of the equality.

Flow conservation constraints for x

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

This constraint guarantees the proper flow through each node and prevents subtours. We call `CPXaddrows` using:

- `vector<int> idx`, containing the indices of the variables involved;
- `vector<double> coef`, containing all coefficients, set to +1 for incoming arcs and -1 for outgoing arcs;
- `char sense = 'E'`, to declare it's an equality constraint;
- `double rhs = 1.0`, to setup the right-hand side value of the equality.

Linking constraints between x and y

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

This constraint, in order to be add in CPLEX, is modified to move all variables in the left-hand side of the inequality:

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

Then we can call `CPXaddrows` using:

- `vector<int> idx(2)`, containing the indices of the variables involved;
- `vector<double> coef(2)`, containing all coefficients: for x_{ij} is +1, while the coefficient of y_{ij} is $-(n-1)$;
- `char sense = 'L'`, to declare it's a "less-than-or-equal" constraint;
- `double rhs = 0`, to setup the right-hand side value of the equality.

2.2.3 Model Solution

The `solve()` method handles the solution process. It initializes the CPLEX environment (`DECL_ENV`) and problem (`DECL_PROB`), applying the set time limit. Specifically, the time limits set for this project are 1, 10, 20, 30, 60, 90, 120, 180, 240 and 300 seconds. Whenever the time limit expires without an optimal solution being found, a new test is attempted with the next limit.

Then, it builds variables and constraints by calling `setupLP()`, executes the optimizer via `CPXmipopt` and records the solution status, objective value and total solving time. If the solution is optimal or feasible, it extracts the optimal tour with extractTour(). The CPLEX problem and environment are then released to free memory.

The generated solution is reconstructed from the y_{ij} variables retrieved using `CPXgetx()`, starting from node 0 and marking it as visited. Then it iteratively, for the current node i, search for a node j such that $y_{ij} = 1$ and j has not been visited, adding the selected node to the tour and setting it as the new current node. This process is repeated until all nodes have been visited, finally appending the starting node 0 to complete a proper cycle. The resulting tour is then stored to be accessed through the `getTour()` method.

For example, the path obtained as the optimal solution to the sample shown in section 1.3 is:

```
0 3 6 2 8 7 5 1 9 4 0
```

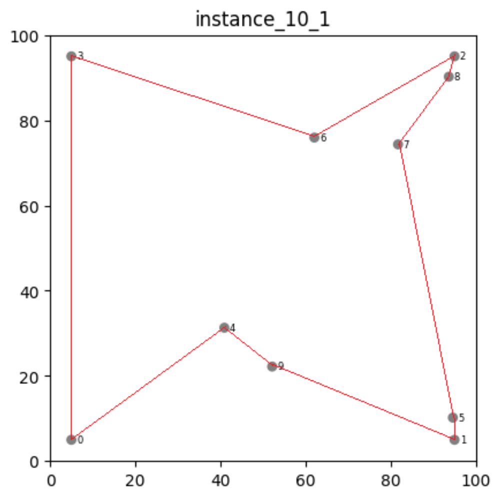

Figure 3. Example of obtained optimal solution with 10 points.

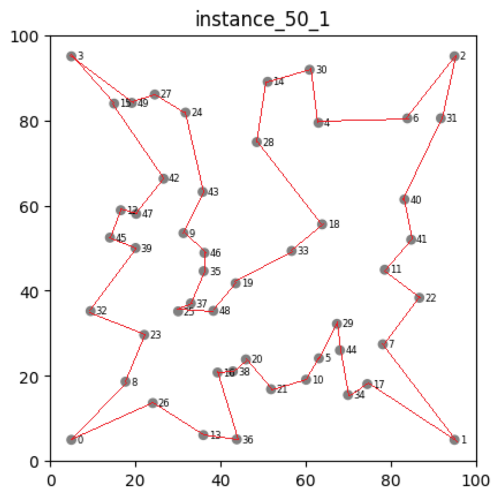

Figure 4. Example of obtained optimal solution with 50 points.

2.3. Tests and Results

All tests have been made with a remote SSH connection to the LabTA computers, using Linux Debian operative system. When the instructions in section 1.2 are correctly followed, the progress of the tested samples will be visible on the terminal during program execution, showing the actual instance name, the applied time limit, the status with the value of the objective function and, when optimal, the final path. All this information, reported in `Ass1/data/solution/log.txt`, are also stored as CSV files in the same `solution/` folder, and shown in the following table. Note that in formulating our problem, we try to minimize the distance traveled by the drill by considering 100x100 mm boards. For this reason, the value of the objective function should be understood with the same unit of measurement, in millimeters.

Table 1. CPLEX test results

Sample	Time Limit	Final Status	Objective Value	Solving Time (s)
10_1	1	OPTIMAL	388.838	0.08403
10_2	1	OPTIMAL	422.244	0.10514
10_3	1	OPTIMAL	400.679	0.14637
10_4	1	OPTIMAL	406.641	0.13828
10_5	1	OPTIMAL	392.784	0.15221
20_1	1	OPTIMAL	429.236	0.44862
20_2	1	OPTIMAL	444.603	0.52091
20_3	1	OPTIMAL	451.699	0.31307
20_4	1	OPTIMAL	471.405	0.32484
20_5	1	OPTIMAL	435.221	0.27473
30_1	1	OPTIMAL	471.068	0.51164
30_2	1	OPTIMAL	513.129	0.88481
30_3	1	OPTIMAL	496.506	0.97170
30_4	1	TIME_LIMIT	525.466	
	10	OPTIMAL	493.530	1.23403
30_5	1	TIME_LIMIT	555.863	
	10	OPTIMAL	501.052	1.36678
50_1	1	TIME_LIMIT	2383.750	
	10	TIME_LIMIT	611.692	
	20	OPTIMAL	606.134	11.9816
50_2	1	TIME_LIMIT	2347.750	
	10	TIME_LIMIT	572.866	
	20	OPTIMAL	569.800	13.8022
50_3	1	TIME_LIMIT	2375.730	
	10	OPTIMAL	583.408	9.8876
50_4	1	TIME_LIMIT	2349.410	
	10	TIME_LIMIT	597.154	
	20	OPTIMAL	578.867	12.9061
50_5	1	TIME_LIMIT	808.072	
	10	OPTIMAL	603.217	5.6344
70_1	1	TIME_LIMIT	3418.620	
	10	TIME_LIMIT	767.405	
	20	OPTIMAL	674.359	18.6599
70_2	1	TIME_LIMIT	3445.830	
	10	OPTIMAL	666.342	6.8903
70_3	1	TIME_LIMIT	3360.070	
	10	TIME_LIMIT	751.438	
	20	OPTIMAL	677.499	17.4976
70_4	1	TIME_LIMIT	3530.730	
	10	TIME_LIMIT	824.634	
	20	TIME_LIMIT	824.634	
	30	TIME_LIMIT	824.634	

Sample	Time Limit	Final Status	Objective Value	Solving Time (s)
	60	TIME_LIMIT	741.674	
	90	OPTIMAL	693.402	78.4773
70_5	1	TIME_LIMIT	3583.180	
	10	TIME_LIMIT	725.661	
	20	TIME_LIMIT	682.817	
	30	OPTIMAL	680383	24.4650
80_1	1	TIME_LIMIT	3997.620	
	10	TIME_LIMIT	842.563	
	20	TIME_LIMIT	842.563	
	30	TIME_LIMIT	719.787	
	60	TIME_LIMIT	714.262	
	90	OPTIMAL	712.595	70.0615
80_2	1	TIME_LIMIT	3981.760	
	10	TIME_LIMIT	1034.180	
	20	TIME_LIMIT	763.836	
	30	TIME_LIMIT	732.753	
	60	OPTIMAL	718.047	43.6698
80_3	1	TIME_LIMIT	3881.540	
	10	TIME_LIMIT	727.108	
	20	OPTIMAL	698.062	11.8106
80_4	1	TIME_LIMIT	3986.180	
	10	OPTIMAL	689.657	8.7897
80_5	1	TIME_LIMIT	3933.640	
	10	TIME_LIMIT	1032.300	
	20	TIME_LIMIT	1032.300	
	30	TIME_LIMIT	1032.300	
	60	TIME_LIMIT	745.610	
	90	OPTIMAL	701.974	72.1836
100_1	1	TIME_LIMIT	5364.670	
	10	TIME_LIMIT	1568.44	
	20	TIME_LIMIT	1037.750	
	30	TIME_LIMIT	1036.550	
	60	TIME_LIMIT	797.321	
	90	TIME_LIMIT	782.609	
	120	TIME_LIMIT	793.052	
	180	OPTIMAL	757.827	126.4240
100_2	1	TIME_LIMIT	4646.310	
	10	TIME_LIMIT	967.940	
	20	TIME_LIMIT	852.311	
	30	TIME_LIMIT	852.311	
	60	TIME_LIMIT	785.421	
	90	OPTIMAL	785.421	72.0638
100_3	1	TIME_LIMIT	4828.400	
	10	TIME_LIMIT	1158.980	
	20	TIME_LIMIT	1048.680	
	30	TIME_LIMIT	1048.680	
	60	TIME_LIMIT	798.773	
	90	TIME_LIMIT	787.139	
	120	OPTIMAL	786.580	103.6120
100_4	1	TIME_LIMIT	5106.700	
	10	TIME_LIMIT	2106.580	
	20	TIME_LIMIT	949.091	
	30	TIME_LIMIT	949.091	
	60	TIME_LIMIT	782.471	
	90	OPTIMAL	782.471	69.6847
100_5	1	TIME_LIMIT	5051.710	
	10	TIME_LIMIT	1589.430	
	20	TIME_LIMIT	1589.430	
	30	TIME_LIMIT	1559.250	
	60	TIME_LIMIT	786.009	
	90	TIME_LIMIT	782.531	
	120	TIME_LIMIT	782.349	
	180	OPTIMAL	782.349	141.1110

3. Improved Kruskal-Like Heuristic-Based Implementation

3.1. Introduction

In this work, we implement a heuristic algorithm composed of two main phases. First, an initial feasible solution is constructed using a greedy approach inspired by Kruskal's algorithm, proposed in 2019 by Pasi Fränti et al. as alternative heuristic solution to TSP [1]. Then, the proposed feasible solution is iteratively improved by applying a local search based on 2-opt moves, where candidate exchanges are evaluated in a structured and deterministic way.

The algorithm is designed to be simple, efficient and suitable for all-sized instances, while still producing high quality solutions.

3.2. Algorithm

The conceptual pseudo-code of the implemented algorithm is reported.

```
INPUT v := starting nodes;
let e := distance between two nodes v and u;
sort e in increasing order of weight;
let T := intial void tour solution;
for edge e = (v,u) in the sorted list do:
·       if degree[v] = 2 or degree[u] = 2 then
·       ·   continue;
·       if e in T then
·       ·   continue;
·       add e in T;
·       degree[v] ← degree[v] + 1;
·       degree[u] ← degree[u] + 1;
·       if edges in T = n-1 then
·       ·   break;
add e in T where degree[v] = 1 and degree[u] = 1;
\\ now T is our initial solution

let w := weight of T;
let improved ← true
while improved = true do:
·       improved ← false;
·       let a := T edges;
·       sort a in decreasing order of weight;
·       for a possible first cut do:
·       ·   for a' possible second cut do:
·       ·   ·   T' ← 2opt(T,a,a');
·       ·   ·   let w' := weight of T';
·       ·   ·   if w' < w do:
·       ·   ·   ·   T ← T'; w ← w';
·       ·   ·   ·   improved ← true;
·       ·   ·   ·   break;
\\ now T is our final approximated solution
return T;
```

3.3. Initial Solution

The initial feasible tour is constructed using a greedy heuristic inspired by Kruskal's algorithm for the Minimum Spanning Tree (MST) problem. Kruskal's algorithm builds an MST by sorting all edges in non-decreasing order of weight and iteratively selecting the shortest edge that does not create a cycle. Its goal is to connect all nodes with minimum total edge weight while avoiding cycles.

Although the TSP requires a Hamiltonian cycle rather than a tree, a similar greedy principle can be exploited to generate a good initial solution. In our approach, all edges between pairs of nodes are first sorted in increasing order of length, as in Kruskal's algorithm, i.e. using a static dispatching rule since the score is based on static information that does not change during iterations. Edges are then added one by one according to the following constraints:

- Each node can have degree at most two, in order to preserve the structure of a tour;
- Cycles are forbidden until exactly $n-1$ edges have been selected.

To efficiently detect premature cycles, a Union-Find data structure is employed, based to the same cycle-detection mechanism used in Kruskal's algorithm. Once $n-1$ edges have been selected, the two remaining nodes of degree one are connected to close the Hamiltonian cycle, creating our initial feasible solution.

3.4. Solution Neighbourhood

The neighbourhood structure used in the local search phase is defined by the classical 2-opt move. A 2-opt move removes two edges from the current tour and reconnects the resulting paths by reversing a subsequence of nodes.

Formally, given a tour $(v_0, v_1, \ldots, v_{n-1}, v_0)$, a 2-opt move selects two positions i and j with $1 \leq i < j \leq n-1$ and reverses the subsequence $(v_i, \ldots, v_j)$. This operation preserves feasibility and generates a neighbouring tour. An example of a 2-opt move:

```
0 1   5 4 3 2   6 7 0
            ↓
0 1   2 3 4 5   6 7 0
```

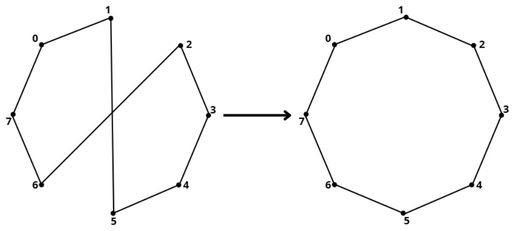

Figure 5. Example of a 2-opt move.

3.5. Iterated 2-opt Move

The local search applies iterated 2-opt moves using a first-improvement strategy. At each iteration all edges of the current tour are considered as potential cut positions and are sorted in decreasing order of their length. This prioritizes the removal of long edges, which are more likely to be part of suboptimal structures.

For each selected edge, feasible 2-opt exchanges involving that edge are evaluated sequentially. As soon as a move that improves the tour length is found, it's immediately accepted and the search restarts from the modified tour.

3.6. Stopping Criteria

The algorithm terminates when a complete pass over the sorted 2-opt neighbourhood fails to produce any improving move. At this point, the current tour is locally optimal with respect to the considered 2-opt neighbourhood.

No randomization elements or time-based stopping condition are employed, making the algorithm fully deterministic. Given the same input instance, the algorithm always produces the same final solution.

3.7. Final Solution

The final solution returned by the algorithm is the locally optimal tour obtained after the iterated 2-opt improvement phase. For each instance the total tour length, i.e. the total distance covered by the drill, and the computational time required to compute the solution are recorded and reported with the final nodes order.

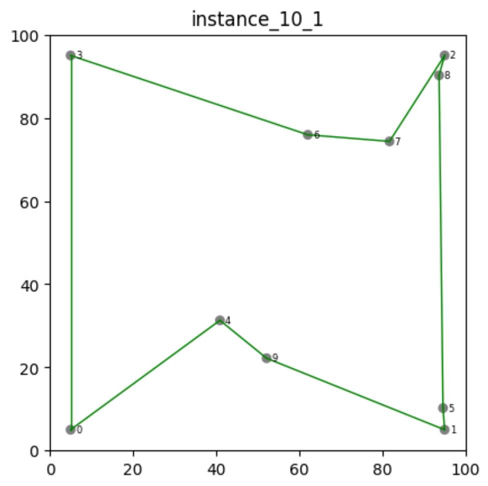

Figure 6. Example of approximated solution with 10 points.

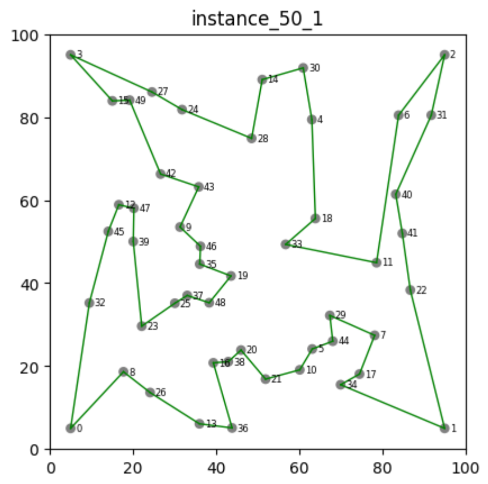

Figure 7. Example of approximated solution with 50 points.

3.8. Tests and Results

All tests have been made with a remote SSH connection to the LabTA computers, using Linux Debian operative system. When the instructions in section 1.2 are correctly followed, the progress of the tested samples (exactly the same generated and used for the first assignment) will be visible on the terminal during program execution, showing the actual instance name, the value of the objective function for the obtained feasible solution and its final path. All this information, reported in `Ass2/data/solution/log.txt`, are also stored as CSV files in the same `solution/` folder, and shown in the following table. We recall again that in formulating our problem, we try to minimize the distance traveled by the drill by considering 100x100 mm boards. For this reason, the value of the objective function should be understood with the same unit of measurement, in millimeters.

Table 2. Improved Kruskal-Like heuristic test results

Sample	Objective Value	Solving Time (s)
10_1	389.816	3.0397e^{-5}
10_2	423.186	1.8247e^{-5}
10_3	400.679	3.0463e^{-5}
10_4	406.641	1.9274e^{-5}
10_5	392.784	2.0014e^{-5}
20_1	431.554	1.2189e^{-4}
20_2	453.255	7.1430e^{-5}
20_3	459.566	8.2420e^{-5}
20_4	480.910	1.0219e^{-4}
20_5	455.173	8.1569e^{-5}
30_1	471.068	2.3610e^{-4}
30_2	533.029	2.5914e^{-4}
30_3	517.599	2.8442e^{-4}
30_4	496.672	2.6494e^{-4}
30_5	555.805	2.9685e^{-4}
50_1	643.156	8.6930e^{-4}
50_2	597.533	1.4549e^{-3}
50_3	610.881	1.5068e^{-3}
50_4	588.346	1.0171e^{-3}
50_5	663.903	1.0960e^{-3}
70_1	726.256	2.7456e^{-3}
70_2	682.427	2.8057e^{-3}
70_3	713.397	3.3267e^{-3}
70_4	715.712	2.6594e^{-3}
70_5	685.984	2.2597e^{-3}
80_1	734.248	3.8650e^{-3}
80_2	742.430	2.6614e^{-3}
80_3	722.772	2.4152e^{-3}
80_4	727.297	1.5908e^{-3}
80_5	738.108	2.6710e^{-3}
100_1	795.722	7.4326e^{-3}
100_2	803.303	5.6780e^{-3}
100_3	836.047	8.3787e^{-3}
100_4	804.087	8.7774e^{-3}
100_5	823.770	6.9216e^{-3}

4. Comparison

In Table 3 and 4, and in the following figures, we can see a comparison between the CPLEX-Based and the Improved Kruskal-Like Heuristic-Based algorithm implemented for our real scenario TSP. While on the one hand the solution obtained through the CPLEX model is always optimal, on the other hand the heuristic algorithm allows to obtain almost optimal results using only fractions of the time required by the exact method.

In particular, in 11% (4 samples) of the cases the Iterated 2-opt implementation produced an optimal solution, in 20% (7 out of 35) it produced a solution nearly comparable to the optimal one with a percentage difference gap lower than 2. Furthermore, only in 22% of the cases (8 samples out of 35) the solution obtained differed by more than 5% from the one obtained with the exact method, of which only 2 cases out of 35 (5.7%) presented a gap >10%.

The biggest difference, immediately visible from the collected results, is the enormous difference in solving times between the two algorithms, despite the limited difference in the quality of the solutions obtained. Every single instance of the heuristic method was at least 2167 times faster than the CPLEX model, with the largest time gap recorded of x29509 for a 70 points instance (78.4 seconds for CPLEX against the 2.65e^{-3} for the heuristic implementation) with only a 3.2% objective value gap.

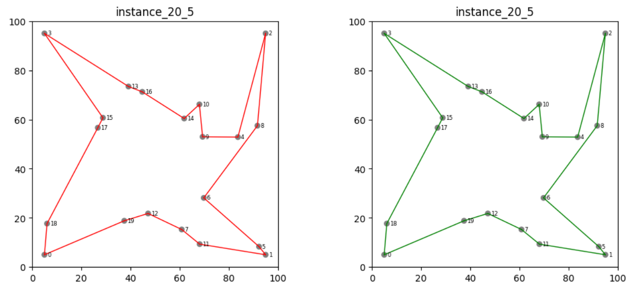

Figure 8. Visual comparison between CPLEX optimal solution (left) and Heuristic approximated solution (right) with 20 points.

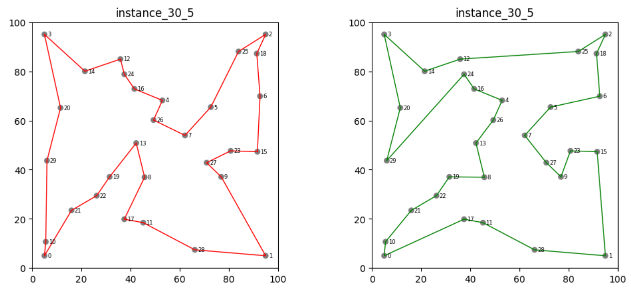

Figure 9. Visual comparison between CPLEX optimal solution (left) and Heuristic approximated solution (right) with 30 points.

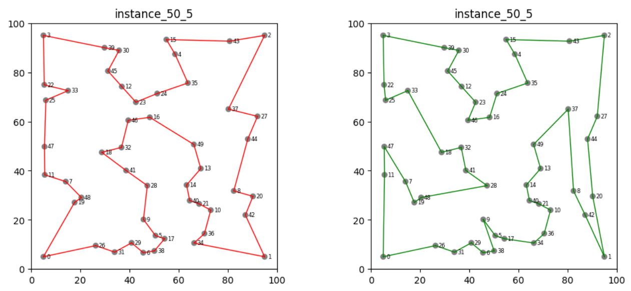

Figure 10. Visual comparison between CPLEX optimal solution (left) and Heuristic approximated solution (right) with 50 points.

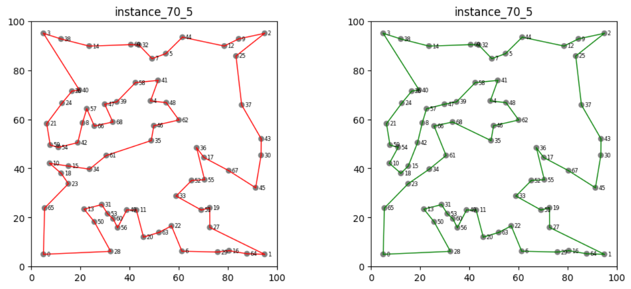

Figure 11. Visual comparison between CPLEX optimal solution (left) and Heuristic approximated solution (right) with 70 points.

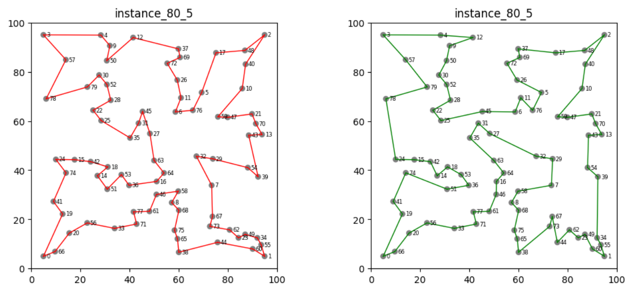

Figure 12. Visual comparison between CPLEX optimal solution (left) and Heuristic approximated solution (right) with 80 points.

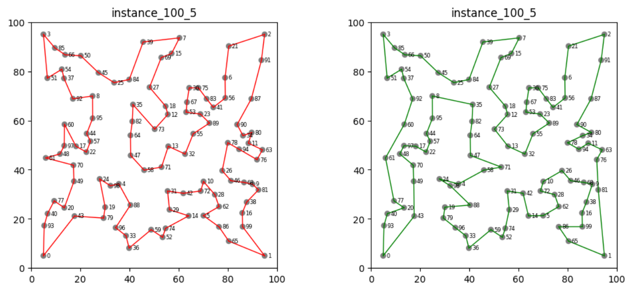

Figure 13. Visual comparison between CPLEX optimal solution (left) and Heuristic approximated solution (right) with 100 points.

Table 3. Comparison of test results

Sample	Objective Value Gap (%)	Heuristic Speedup
10_1	0.251	x2764
10_2	0.223	x5762
10_3	0	x4804
10_4	0	x7174
10_5	0	x7605
20_1	0.540	x3680
20_2	1.946	x7292
20_3	1.741	x3798
20_4	2.016	x3178
20_5	4.584	x3368
30_1	0	x2167
30_2	3.878	x3414
30_3	4.248	x3416
30_4	0.636	x4657
30_5	10.927	x4604
50_1	6.107	x13783
50_2	4.867	x9486
50_3	4.709	x6561
50_4	1.637	x12689
50_5	10.060	x5140
70_1	7.695	x6796
70_2	2.413	x2455
70_3	5.298	x5259
70_4	3.217	x29509
70_5	0.823	x10826
80_1	3.038	x18127
80_2	3.395	x16408
80_3	3.539	x4890
80_4	5.457	x5525
80_5	5.147	x27024
100_1	5.000	x17009
100_2	2.276	x12691
100_3	6.288	x12366
100_4	2.762	x7939
100_5	5.294	x20387

Table 4. Comparison of average results

Problem Size	Avg. CPLEX Solving Time	Avg. Heuristic Solving Time	Avg. Obj. Value Gap
10	0.1252	$2.3679e^{-5}$	0.094
20	0.3764	$9.1899e^{-5}$	2.165
30	0.9937	$2.6829e^{-4}$	3.937
50	10.8423	$1.1888e^{-3}$	5.476
70	29.1980	$2.7594e^{-3}$	3.889
80	41.3030	$2.6406e^{-3}$	4.115
100	102.5791	$7.4376e^{-3}$	4.324

5. Considerations

In this project we solved a TSP applied to a real scenario with two different methods: an exact one, through a CPLEX model, and an heuristic one, with a Kruskal-like approach improved with a 2-opt moves local search.

The first approach allowed us to always obtain tours with a guarantee of solution optimality, thus minimizing the total distance traveled by the drill in our problem. However, this comes at the cost of computational time: as the problem size increases, the time required grows exponentially.

The heuristic approach, on the other hand, allowed us to drastically reduce the time needed to obtain a solution, in the most significant cases going from several minutes to milliseconds. This speed is, of course, the main advantage, although the use of the heuristic almost never produces an optimal solution. Anyway, it is important to highlight that the quality of the solutions provided was still excellent, with very few cases where the gap with the CPLEX solution exceeded 10%.

In the specific case analyzed in this project, involving a TSP applied to the production of electrical boards to minimize the time required to drill holes, the best approach is the Heuristic-Based one. Although this basic implementation almost never produces the optimal solution, we must remember that the gap between optimal and given tour solution translates into an increase in the time required for the drill to travel the Hamiltonian cycle. However, given the enormous speedup of the heuristic method, the time saved in producing the final cycle (i.e. the drill's travel path) is much greater than the difference in travel time in the case of an optimal solution. Let's take the most striking example of instance 100_5: the exact CPLEX-Based method took 114 seconds, while the heuristic approach returned a good solution in less than a hundredth of a second. It is rational to assume that the quality gap between the two solutions cannot produce a two minute slowdown in the drill's movement: indeed, the time needed to obtain an optimal solution is completely unjustifiable when compared to that spent to obtain a solution close to optimal.

Furthermore, the implemented heuristic is extremely simplified, as it is completely deterministic, always using the same greedy constructive heuristic to produce the initial solution, without implementing any random element or any method to improve the following local search in the neighborhood, e.g. a stepest descent exploration strategy, a diversification of the neighborhood with a different k-opt after reaching the local optima with 2-opt etc...

It was decided to limit ourselves to the implementation of such a simple heuristic algorithm since, for the specific use case of the project, a more robust method, with a slightly better quality of final solutions but requiring longer solving time, would not have represented a real improvement in terms of saved drilling time.

Indeed, to test the validity of more robust heuristics, a second version of the Heuristic-Based algorithm was developed. In this alternative version, after generating the initial solution with the Kruskal-like approach, the solution space is continuously diversified iteratively changing the definition of the neighborhood from 2-opt to 3-opt and back to 2-opt, until a better solution is no longer achievable. However, this more complex algorithm was discarded because the actual improvement in the objective function was too small compared to the large increase in solving time, as shown in Table 5.

Table 5. Comparison between proposed algorithm and advanced discarded one

Instance	Adv. Heuristic Solving Time	Adv. Heur. Obj. V. Gap*	Adv. Heur. Slowdown**
100_1	0.5179	3.584	x69.68
100_2	0.3455	1.848	x60.86
100_3	0.7842	0.737	x93.65
100_4	0.3596	0.931	x40.97
100_5	0.5513	1.988	x79.64
Average	0.5117	1.817	x68.96

* improvement w.r.t. the optimal solution's objective value

** w.r.t. the proposed algorithm's solving time

Considering the specific domain this project addresses, where the distances involved are in the order of millimeters, the advanced algorithm would have returned a drilling distance approximately 20 mm smaller in the larger instances, while requiring a solving time approximately half a second longer. Since it is reasonable to assume that the drill can travel that minimum distance in a time no greater than the gap between the solutions obtained by the two algorithms, it was decided to propose the simplest version as the final Heuristic-Based algorithm.

Although not implemented as the final proposed algorithm, the source code of the Advanced Improved Kruskal-Like algorithm for diversification of the neighbourhood iterating 2-opt and 3-opt moves is available in `Ass2/TSPAdvHeuristic.cpp` as complement to this paper's work. If the problem domain had involved higher units of measurement, such as meters, then this more complex algorithm would have been chosen as the final proposal.

6. Conclusions

In conclusion, we have shown that, depending on the type of problem to be solved and the scenario in which it is applied, it is necessary to find a trade-off between speed and solution quality. If obtaining a solution with guaranteed optimality is the main requirement, then using an exact method such as a CPLEX solver is the best approach.

On the other hand, if the problem allows for even a minimal tolerance in solution quality relative to the optimum, using a very simple and basic heuristic, such as the one proposed in this work, using local search based on 2-opt moves to extend and improve the first starting feasible solution obtained using Fränti and Nenonen's proposal, is certainly the better choice as it provides a solution close to the optimum or, in the worst case, a solution of still good quality but with a speedup of several orders of magnitude.

References

[1] Pasi Fränti and Henrik Nenonen. Modifying kruskal algorithm to solve open loop tsp. 2019.

[2] Bezalel Gavish and Stephen Graves. The travelling salesman problem and related problems. 1978.

* Project code is available in its Github repository here.

Appendix A – Project Structure

The project is organized as follows.

```
Project/
+-- Ass1/
|   +-- cpxmacro.h
|   +-- main.cpp
|   +-- makefile
|   +-- TSPInstance.cpp
|   +-- TSPInstance.h
|   +-- TSPModel.cpp
|   +-- TSPModel.h
|   +-- data/
|       +-- generator/
|       |   +-- generator.cpp
|       |   +-- generator.exe
|       +-- plot/
|       |   +-- instance_10_1_plot.png
|       |   +-- instance_10_1_sol.png
|       |   +-- ...
|       +-- solution/
|       |   +-- log.txt
|       |   +-- result_10.csv
|       |   +-- ...
|       +-- instance_10_1.dat
|       +-- ...
+-- Ass2/
    +-- main.cpp
    +-- makefile
    +-- TSPAdvHeuristic.cpp
    +-- TSPHeuristic.cpp
    +-- TSPHeuristic.h
    +-- TSPInstance.cpp
    +-- TSPInstance.h
    +-- data/
        +-- plot/
        |   +-- instance_10_1_approx_sol.png
        |   +-- ...
        +-- solution/
        |   +-- log.txt
        |   +-- log_adv.txt
        |   +-- result_10.csv
        |   +-- ...
        +-- instance_10_1.dat
        +-- ...
```

The above files are described as follows.

- `Ass1/` contains all files related to the first assignment:
  - ✦ `cpxmacro.h` contains macros for CPLEX API;
  - ✦ `main.cpp` runs TSPInstance and TSPModel on the dataset, contains the logic to handle tests results;
  - ✦ `makefile` compiles the project;
  - ✦ `TSPInstance.cpp` represents a sample from the dataset;
  - ✦ `TSPModel.cpp` contains the logic for the CPLEX solver;
  - ✦ `data/` contains all files related to the dataset and the obtained solution:
    - ✧ `generator/` contains the .cpp and .exe files that generated the dataset;
    - ✧ `plot/` contains the images used for this report related to the assignement;
    - ✧ `solution/` contains the log and the CSV files with all results and solutions;
    - ✧ `.dat` files that make up the dataset.
- `Ass2/` contains all files related to the second assignment:
  - ✦ `main.cpp` runs TSPInstance and TSPHeuristic on the data samples, contains the logic to handle tests results;
  - ✦ `makefile` compiles the project;
  - ✦ `TSPAdvHeuristic.cpp` contains the discarded implementation of the Advanced Improved Kruskal-Like algorithm with 2-opt and 3-opt iterated moves;
  - ✦ `TSPHeuristic.cpp` contains the implementation of the Improved Kruskal-Like algorithm;
  - ✦ `TSPInstance.cpp` represents a sample from the dataset, it is identical to the file located in `Ass1/`;
  - ✦ `data/`
    - ✧ `plot/` contains the images used for this report related to the assignement;
    - ✧ `solution/` contains the log and the CSV files with all results and solutions, including the log with results of the advanced algorithm;
    - ✧ `.dat` files that make up the dataset, they are identical to the ones in `Ass1/`.