Methods and Models for Combinatorial Optimization

Lorenzo Schiavone

September 4, 2025

Contents

1 Problem

For the Traveling Salesman Problem (TSP) in the context of drilling holes in electrical panels:

- implement the exact mathematical model using the Cplex API and test it,
- design and implement an ad hoc optimization algorithm to solve it,
- test the performance of the implemented model comparing the performance with the exact Cplex model
- write a final report where you describe how you implemented the mathematical programming model, some design issue concerning the algorithm of Part II, the overall parameter tuning activity, and the computational results of both Part I and Part II, and their comparison.

2 Mathematical Model

The problem consists in finding a minimum cost Hamiltonian cycle in a undirected fully connected graph $G = (N, E)$ where the nodes are the points in the board where holes have to be made, and the set of edges (i, j) correspond to the trajectory of the drill from the node i to j. The weight c_{ij} of the edges represents the cost or the time the drill takes to move from node i to j.

Parameters:

- c_{ij} cost of the edge (i, j),
- 0 arbitrarily selected starting node.

Decision variable:

- x_{ij} amount of flow shipped from i to j,
- y_{ij} binary variable 1 if edge (i, j) ships some flow, 0 otherwise.

Integer Linear Programming Model:

$$\min \quad \sum_{(i,j)\in E} c_{ij} y_{ij} \tag{1}$$

$$\text{s.t.} \quad \sum_{i:(i,k)\in E} x_{ik} - \sum_{j:(k,j)\in E} x_{kj} = 1, \qquad \forall\, k \in N \setminus \{0\} \tag{2}$$

$$\sum_{j:(i,j)\in E} y_{ij} = 1, \qquad \forall\, i \in N \tag{3}$$

$$\sum_{i:(i,j)\in E} y_{ij} = 1, \qquad \forall\, j \in N \tag{4}$$

$$x_{ij} \leq (|N| - 1) y_{ij} \qquad \forall\, (i, j) \in E \tag{5}$$

$$x_{ij} \in \mathbb{R}_+ \qquad \forall\, (i, j) \in E \tag{6}$$

$$y_{ij} \in \{0, 1\} \qquad \forall\, (i, j) \in E. \tag{7}$$

3 Instances Generation

For instances generation, we consider a board of size 40×20 and select specific points within the discrete grid. The discrete unit might represent the minimum distance at which holes can be safely located in the board. We use the following patterns: vertical lines, horizontal lines, diagonal lines, and compositions of these elements to form triangles and rectangles. These patterns are combined with randomly selected points until the desired number of nodes is reached.

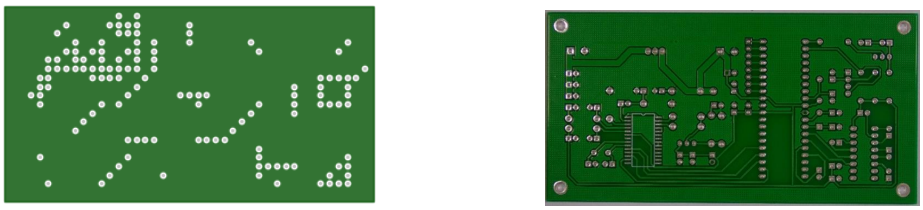


Figure 1: Comparsion of a generated board with 128 nodes and a real PCB.

The cost of the edge between nodes i and j is simply the Euclidean L2 distance. A more specific metric could be used if the drill machine's movements are non uniform, such as if a fully extended position far from the center increases the time required. We store the cost matrix in a "{instance}.dat" file where the first line contains the number of nodes, and the coordinates of the nodes in a "pos_dict{instance}.json" file to be able to rebuild the solution tour.

We create ten instances per size $\{16, 32, 64, 128, 256\}$ both for training and for testing with the python script "BoardGeneration.py".

Solution representation We choose to represent the solutions as tour starting and ending in 0. For example, for an instance with 6 nodes, the solution

0 3 1 4 2 5 0

corresponds to the Hamiltonian tour with edges {(0, 3), (3, 1), (1, 4), (2, 5), (5, 0)}.

4 CPLEX Model

The CPLEX model is set up as follows:

1. read the cost matrix $C = (c_{ij})$,
2. create the integer variables x_{ij} and the binary variables y_{ij} for every pair (i, j) such that the corresponding cost c_{ij} is positive,
3. add the constraints as defined in Model 1.

Since all variables in CPLEX are stored in a single vector, it is necessary to track which component corresponds to each variable x_{ij} and y_{ij}. To achieve this, we define two matrices, `map_x` and `map_y`, where each entry (i, j) stores the index of the corresponding variable in the vector.

Additionally, to reconstruct the tour solution representation, we introduce a mapping vector `rev_map`, which allows us to retrieve the pair (i, j) from the variable vector index. Since the variables are stored in a single vector, we also record the index that separates the x variables from the y variables.

After the optimization is completed, we save the objective value and the optimal tour in a txt file in the ./solutions/exact folder.

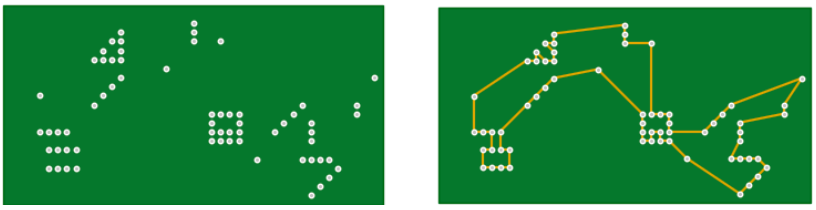


Figure 2: Exact solution for a board with 64 holes.

We record in Table 1 the average time taken by the exact method on ten instances per size $\{16, 32, 64\}$ (1 run per instance). All experiments were conducted on a MacBook Air (2017) with an Intel Core i5-5350U

processor (dual-core, 1.8 GHz, Turbo Boost up to 2.9 GHz, 3MB L3 cache) and 8GB LPDDR3 RAM at 1600 MHz. We are able to obtain the exact solution in reasonable time only for instances with a relatively small number of node. For instances with 64 nodes the running time was around 145 seconds on average and we stop to try to obtain the exact solution for larger instances. The largest instance size that the exact cplex model is able to solve in 10 seconds on average is around $n = 32$, in the used computer.

n	16	32	64
time	0.47	10.41	145.04

Table 1: Average CPU time (sec) for solving the exact model with n nodes.

5 Local Search

Given the large amount of time to run the exact model for instances with many nodes we have to rely on heuristics for finding good solution. However, this prevents any guarantee of optimality. As a first heuristic, we implement a multi-start local search with an increasing neighborhood approach. The neighborhoods are defined by 2-opt moves, and if no improving 2-opt move is found, we consider 3-opt moves. We evaluate neighborhoods incrementally, avoiding full recomputation of the entire tour. In Figure 3, we enumerate the possible 3-opt moves that are not already covered by 2-opt moves to properly distinguish modifications to the tour.

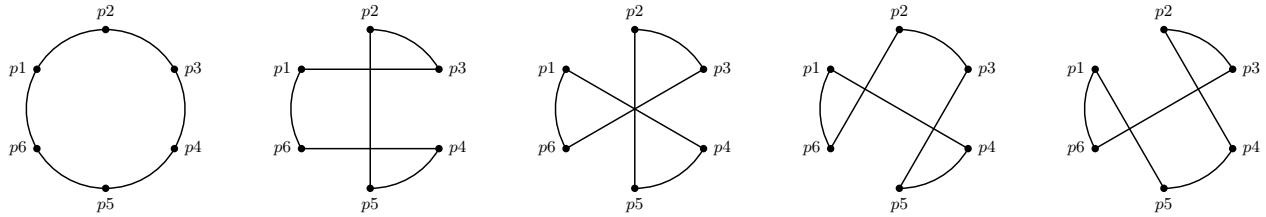


Figure 3: Enumeration of the strictly 3-opt moves.

To speed up neighborhood evaluation for the 3-opt moves, we employ an early break approach, where the neighborhood search is ended as soon as a move is found with an improvement greater than a specific fraction of the last successful improvement. Specifically, we choose the fraction to be 0.75. It allows the algorithm to still explore the neighborhood without accepting the very first improvement, giving it a better chance of finding a significantly good move and avoiding the whole neighborhood evaluation.

The search starts from the solution obtained by the farthest insertion heuristic to quickly reach a competitive solution. We then restart from random initial solutions and save the best solution whenever we find a better one until the prescribed `TimeLimit` is reached. The only tunable parameter left in this approach is the `TimeLimit`.

6 Genetic Algorithm

As an alternative, we implement a Genetic Algorithm for the TSP while maintaining the same solution representation for consistency. The main steps of the algorithm are summarized in Algorithm 1.

Algorithm 1 Genetic Algorithm for TSP

```
population ← initPopulation()
for it = 1, ..., MAXIT do
    matingPool ← getMatingPool(population)              ▷ Select parents
    newPopulation ← getNewPopulation(matingPool)        ▷ Generate offspring
    population ← NaturalSelection(population, newPopulation)   ▷ Select new individuals
    Population Management
end for
```

What characterizes the Genetic Algorithm is the design of each component. We make the following choices:

1. **Initial population:** A mix of random solutions and heuristic solutions (one per type), generated using localSearch from a random solution, Farthest insertion and Nearest insertion.
2. **Mating pool:** Individuals are selected based on linear ranking proportional to total path cost. Every pair in the mating pool is used as parents to generate a new child. The pool size is chosen such that the number of offspring generated meets the required `newPopSize`.
3. **Crossover:** The *Order Crossover* operator is applied to generate a new solution from two parents.
4. **Mutation:** A swap mutation operator is used, where two indices in the tour representation are exchanged. The mutation process happens with an input-defined probability.
5. **Natural selection:** The top 5% of elite individuals are retained, while the remaining 95% are selected using linear ranking from both the old populations and the new offspring.

After all iterations, we apply a local search to the best solution found as a final refinement. Additionally, offspring undergo a cheaper 2-opt local search to enhance solution quality with a low probability (`lsRate`).

To balance exploration and convergence, we adopt a linearly decreasing temperature-based cooling scheme for the `mutationRate` per child.

6.1 Population Management

To monitor population diversity, we compute the average Hamming distance:

$$\text{hamming}(x, y) = \frac{1}{N-1} \sum_{i=1}^{N} 1_{x_i \neq y_i}.$$

where $\text{hamming}(x, y) = 1$ if all elements differ and 0 if they are identical. Since our path representation is not unique (i.e., a sequence y and its reverse y^{rev} represent the same path), we compute

$$\min\{\text{hamming}(x, y), \text{hamming}(x, y^{\text{rev}})\}$$

as the effective distance. If the average Hamming distance drops below 0.15, we replace a portion `replace_factor` of the population with random individuals and restart the cooling schedule. This value was chosen empirically to allow the population to converge on promising solutions and prevent premature convergence. A lower threshold risks deep stagnation to a local optima, while a higher threshold could prevent the algorithm to refine good solutions.

6.2 Parameter Tuning

The Genetic Algorithm involves multiple parameters that influence performance. Since parameter calibration is an optimization problem in itself, we only sketch a rough attempt to find good values for

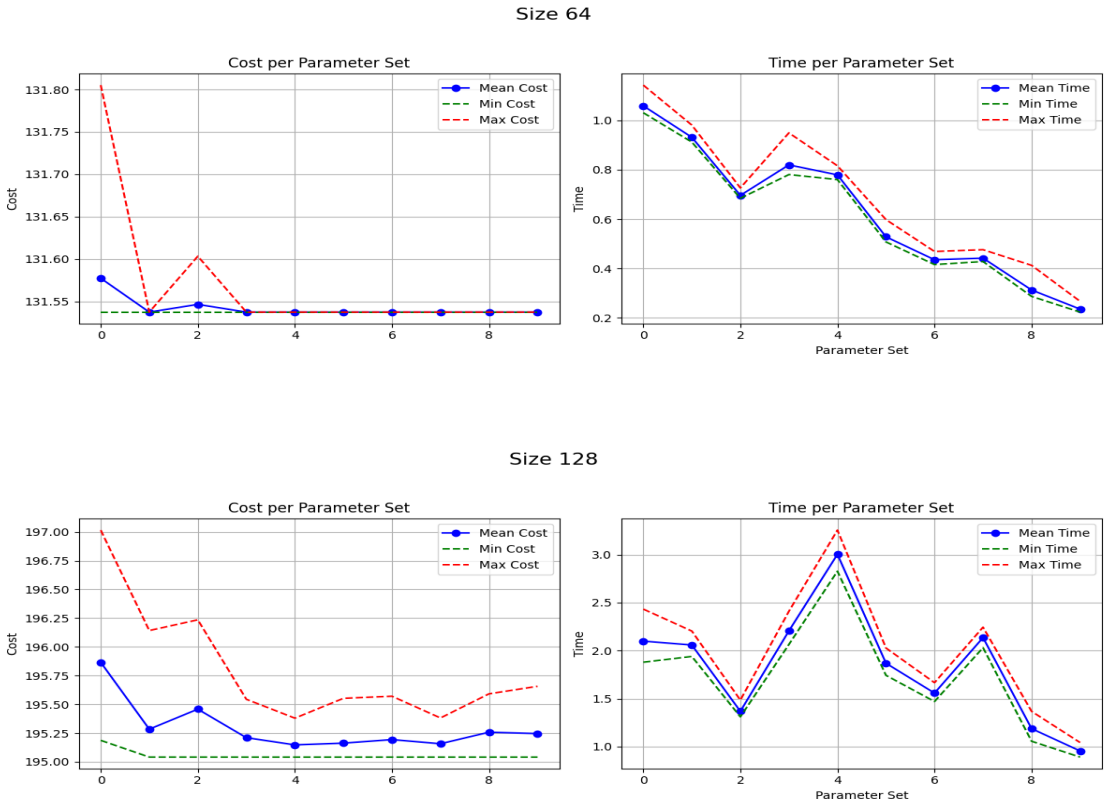
- `popSize`,
- `newPopSize`,
- `lsRate`,
- `maxMutationRate`,
- `replace_factor`,
- `MAXIT`.

We consider ten candidate sets of parameters as in Table 2.

	popSize	newPopSize	lsRate	maxMutationRate	replace_factor	MAXIT
0	200	100	0.05	0.05	0.05	1000
1	150	150	0.05	0.15	0.15	1000
2	150	50	0.1	0.05	0.1	1000
3	100	200	0.1	0.1	0.1	1000
4	100	100	0.1	0.05	0.25	1000
5	50	150	0.05	0.1	0.2	1000
6	50	100	0.05	0.2	0.25	1000
7	50	50	0.05	0.1	0.75	1000
8	20	50	0.02	0.2	0.5	1500
9	20	20	0.02	0.15	0.95	2000

Table 2: Parameters candidate sets.

For each parameters set, we run the algorithm ten times per instances (ten per size) in the train set for the instances of size 64, 128 and 256. Then, we compute the ten run max, min and mean of objective value and time for instance per parameter setting and finally the average among the instance of the same size.



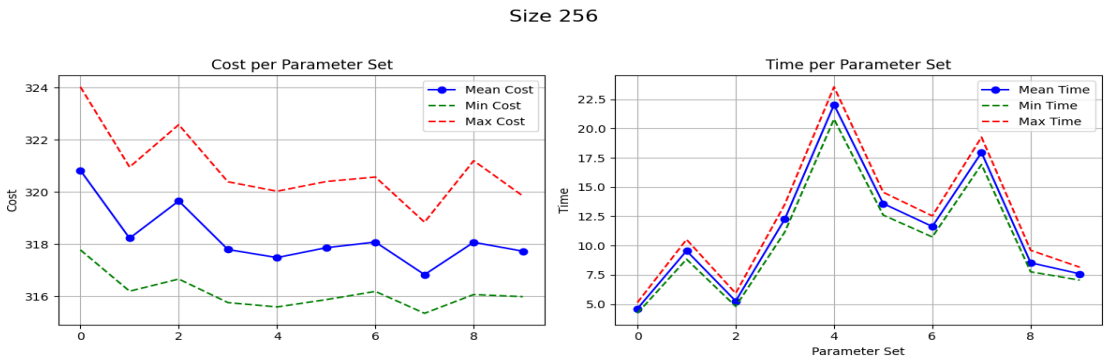


Figure 4: Minimum, maximum and average time and objective value per parameter setting and size.

Figure 4 summarizes the results of our analysis. Considering the trade-off between computing time and objective value, we select candidate parameter setting 9 as the most promising configuration for the three considered size. It corresponds at keeping in memory a small amount of population and generate few offspring. The healthy and prolific diversity is assured by the aggressive injection of new random individuals whenever the average hamming distance is too low. The small amount of population allows a larger `MAXIT` without increasing much the running time.

Still, it is worth noting that a different setting might be better for specific needs or if more time and computational power are available to find a better solution.

7 Results

For instances with small size, i.e., below 64 in our case, we are able to compare the exact method with the heuristics and they always find the optimal solution in a fraction of the time as we can see with some examples in Table 3. Therefore heuristics have to be preferred. Still, if large resources are available or a guaranteed performance is needed, one can rely on the exact model or the Christofides algorithm.

	exact		localSearch		genetic	
instance	Objective value	Time	Objective value	Time	Objective value	Time
16_0	55.0367	0.249	55.0367	0.1	55.0367	0.072455
16_1	78.3420	0.286	78.3420	0.1	78.3420	0.073564
16_2	81.1642	0.260	81.1642	0.1	81.1642	0.073391
32_3	117.2420	31.466	117.2420	0.1	117.2420	0.089795
32_4	89.2019	1.146	89.2019	0.1	89.2019	0.092180
32_5	81.8696	2.198	81.8696	0.1	81.8696	0.090264
64_6	131.3860	99.486	131.3860	0.5	131.3860	0.241862
64_7	129.3630	515.696	129.3630	0.5	129.3630	0.244171
64_8	139.4730	41.194	139.4730	0.5	139.4730	0.232181

Table 3: Objective value and running time per instance comparison between exact method (1 run) and heuristics (10 runs average). Time for the localSearch is fixed a priori.

Moreover, we compare the multistart localSearch with the genetic algorithm with the parameter setting chosen above. For fairness we set as `timeLimit` for the localSearch the max time taken by the genetic algorithm for the same size, i.e. 0.3 seconds for size 64, 1 for size 128 and 8 for size 256. We run the two methods against 10 instances per size in the test set. Table 4 summarizes the obtained results. We can notice that both method for size 64 find the best solution with consistency but the localSearch falls short for larger size instances where the genetic algorithm with this parameter setting outperforms it both in min,max and mean.

It is relevant to notice that since for larger instances the optimal solution is not known, the quality of the results are evaluated only by comparing them to available solutions.

	localSearch				genetic			
size	Min	Max	Mean	Std	Min	Max	Mean	Std
64	136.03790	136.06770	136.04832	0.013345	136.03790	136.03790	136.03790	0.000000
128	200.27490	201.44230	200.83147	0.390799	199.81380	200.49950	200.03751	0.262122
256	327.86070	332.19420	330.07724	1.432281	321.72390	326.39600	323.65060	1.545797

Table 4: Objective values statistics of ten runs for localSearch and genetic Algorithm on test instances. Values are obtained first per instance and then averaged per size.

Table 5 records the same statistics for the running time of the genetic algorithm. For the local search time is fixed a priori and there is no need for the same table.

size	Min	Max	Mean	Std
64	0.243184	0.386576	0.290644	0.046370
128	0.900180	1.149564	1.005217	0.080882
256	7.064713	8.139161	7.562328	0.344552

Table 5: Running time statistics of ten runs for genetic Algorithm on test instances. Values are averaged per instance and then per size.

8 Conclusions

For the Traveling Salesman Problem in the context of drilling holes in panels, we have written the exact model and solved with the Cplex API.

As the exact model can be solved in reasonable time only for small instances, we have developed and implemented two ad hoc heuristic optimization algorithms.

The first approach is a problem-specific multi-start local search, while the second is a general-purpose genetic algorithm. Although with the chosen parameter settings, the genetic algorithm outperforms local search, its effectiveness comes at the cost of time consuming parameter tuning. The large number of parameters, their interdependence and their dependence on instance characteristics, makes the genetic algorithm more challenging to be deployed efficiently.

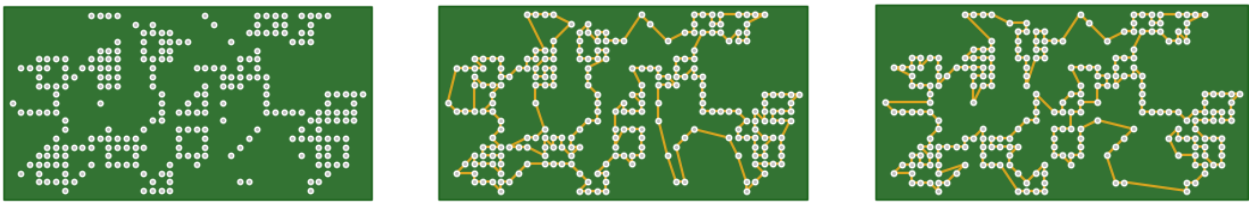


Figure 5: Example of solution for a board with 256 holes. Local Search solution on the left (objval = 327.607), Genetic solution on the right (objval = 322.242)