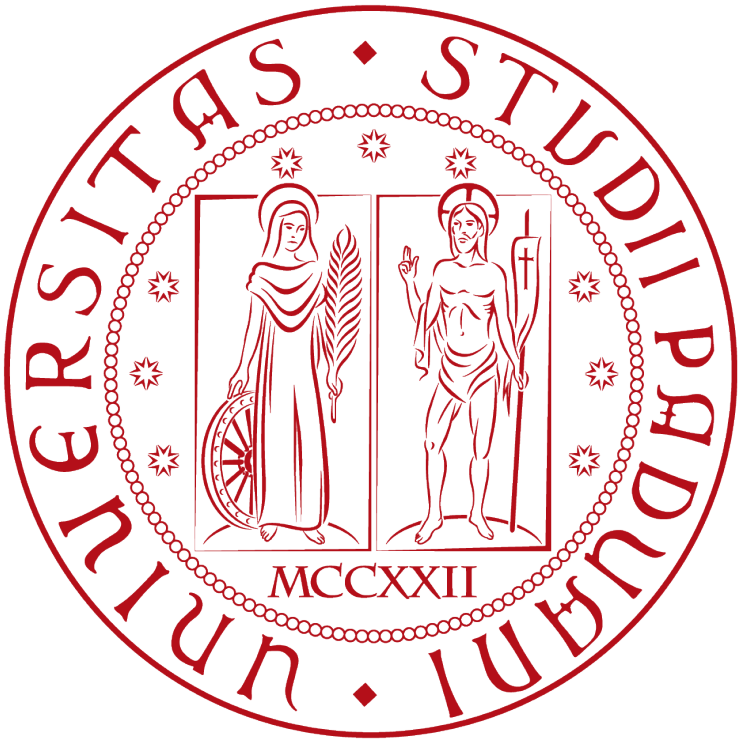

Tabu Search and CPLEX Approaches to the Traveling Salesman Problem for PCB Drilling Optimization

Simone Caregnato

simone.caregnato@studenti.unipd.it - 2154604

Methods and Models for Combinatorial Optimization

Computer Science
Università degli Studi di Padova

A.Y. 2024/2025

Contents

Figures

Tables

1. Introduction

This project addresses an optimization problem arising in the manufacturing of printed circuit boards (PCBs). Each board contains a set of holes that must be drilled, and the drill head must move across the board to reach all positions before returning to the starting point. Since the layout is repeated over many boards, minimizing the travel path can significantly reduce production time and increase efficiency.

To solve this, we implement and compare two approaches:

- an **exact method**, based on a mathematical formulation solved with the **CPLEX** optimization library;
- a **heuristic method**, using a **Tabu Search** algorithm designed to explore the solution space efficiently and return high-quality results within limited time constraints.

1.1. Problem description

The task can be reduced to a classic Travelling Salesman Problem (TSP), where each hole on the board corresponds to a node, and the drill must visit every node exactly once and return to the starting point. The goal is to find the tour (a permutation of nodes) that minimizes the total travel distance, assuming the drilling time at each hole is constant and can thus be ignored in the optimization.

Formally, the problem can be represented on a complete weighted graph $G = (N, A)$, where:

- **Sets**:

  N: set of nodes representing the holes

  A: set of arcs (i, j) with $i, j \in N$, representing valid transitions between holes

- **Parameters**:

  c_{ij}: Euclidean distance between node i and node j

  Node 0: arbitrarily selected starting node

- **Decision Variables**:

  $x_{ij} \in \mathbb{R}^+$: flow of a fictitious commodity from node i to node j

  $y_{ij} \in \{0, 1\}$: equals 1 if arc (i, j) is used in the tour, 0 otherwise

- **MILP Formulation**:

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

subject to:

$$\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j)\in A,\, j\neq 0} x_{kj} = 1 \qquad \forall k \in N \setminus \{0\} \tag{2.1}$$

$$\sum_{j:(i,j)\in A} y_{ij} = 1 \qquad \forall i \in N \tag{2.2}$$

$$\sum_{i:(i,j)\in A} y_{ij} = 1 \qquad \forall j \in N \tag{2.3}$$

$$x_{ij} \leq (|N|-1) y_{ij} \quad \forall (i,j) \in A, j \neq 0 \tag{2.4}$$

$$x_{ij} \in \mathbb{R}^+ \qquad \forall (i,j) \in A, j \neq 0 \tag{2.5}$$

$$y_{ij} \in \{0,1\} \qquad \forall (i,j) \in A \tag{2.6}$$

This compact flow-based formulation ensures subtour elimination and enforces valid transitions in the tour while minimizing the total travel cost.

1.2. Assumption

To ensure comparability across different test cases and maintain a clear problem definition, several assumptions were made during the modeling and implementation phases.

The drill is assumed to move freely in all directions, including diagonally, with constant speed. As a consequence, the cost associated with moving from one hole to another is defined solely by the Euclidean distance between them. The time required to perform the actual drilling is constant across all holes and therefore excluded from the optimization objective.

Boards are generated randomly based on two parameters: the board size and a density value $d \in [0, 1]$. The density controls the number of holes on the board. Hole positions are then sampled randomly across the grid, ensuring no duplicates.

To avoid trivial instances and reduce solving times for the exact model, only boards with at least 3 holes and at most 60% filled cells are considered. The generated files follow a naming convention that encodes the size, density, and repetition index of the instance, enabling automated retrieval of tuned parameters and results.

1.3. Input representation

All instances are based on square boards, represented as $s \times s$ grids. Each board is stored in a '`.dat`' file, where the first line contains the board size s, followed by a matrix of 0s and 1s. A value of `1` indicates the presence of a hole, while `0` denotes an empty cell.

1.4. Instance generation

The generation of input instances is handled via a C++ routine that produces random square drill boards of variable size and density. Each instance corresponds to a grid of size $s \times s$, where a set number of holes is randomly placed according to a specified density parameter.

The number of holes n is determined as $n = \lfloor d \cdot s^2 \rfloor$, and these are sampled without repetition using the C++ container to avoid duplicates. Hole positions are uniformly distributed across the grid.

The instance generator is integrated directly into the parameter tuning and benchmarking pipeline, which automatically generates multiple boards per configuration (size, density, repetition index). The naming of the .dat files reflects these parameters, facilitating consistent referencing across experiments.

2. Part I - CPLEX Solver

2.1. Implementation details

The exact approach is implemented in C++ using the CPLEX Callable Library. The main routine is responsible for loading the instance, computing the distance matrix, building the MILP model, invoking the solver, and exporting the results.

After parsing the `.dat` file and extracting hole positions (stored in a **`std::vector<Hole>`** where `Hole` is a simple struct representing the coordinates of each hole), a cost matrix is computed using pairwise Euclidean distances. Each node is uniquely identified by its index in the hole list.

The model is constructed inside the **`setupLP`** function, which receives the CPLEX environment, problem pointer, distance matrix, and number of holes. Two sets of variables are declared:

- binary variables y_{ij}, one for each arc (i, j), are used to represent the selection of edges in the final tour;
- continuous variables x_{ij} are used to implement the subtour elimination constraints based on a flow model.

For each pair (i, j), the corresponding variables are added using `CPXnewcols`, and their indices are stored in the lookup tables **`map_x[i][j]`** and **`map_y[i][j]`**.

Constraints are then added to the model in three groups:

- **Flow conservation** Equation 2.1: For each internal node k, the incoming and outgoing flow must sum to 1, enforcing connectivity and eliminating subtours.
- **Degree constraints** (Equation 2.2 & Equation 2.3): For each node, exactly one incoming and one outgoing arc must be selected.
- **Coupling constraints** (Equation 2.4): Ensure that flow can only travel on arcs included in the solution, by bounding x with y.

Each group of constraints is encoded by creating sparse row representations and passed to **`CPXaddrows`** in batch. Names and bounds for variables and constraints are generated using standard string formatting. The macros provided in **`cpxmacro.h`** handle environment and error checks.

Once the model is complete, **`CPXmipopt`** is called to solve the instance. The solution cost is printed to the terminal and also saved to a `.sol` file using **`CPXsolwrite`**. Timing is tracked via `chrono` library to measure solver performance.

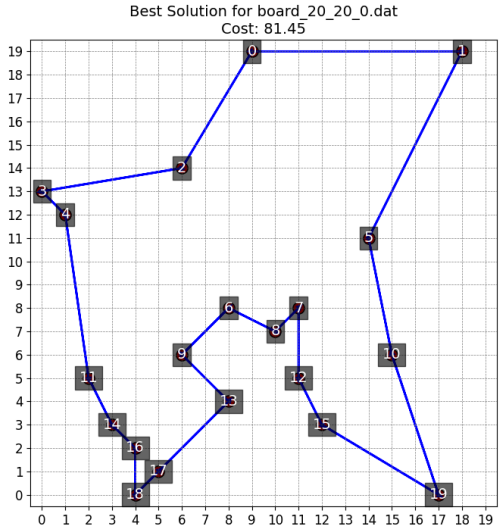

Figure 1: Board 20x20 with 20 holes, solved with CPLEX

3. Part II - Tabu Search

In Part II, Tabu Search was selected as the metaheuristic method to address the combinatorial optimization problem introduced in Part I. Tabu Search is an iterative improvement algorithm that explores the neighborhood of the current solution using local search operators, typically accepting only improving moves unless an aspiration criterion is met. To avoid cycling and promote exploration, recently visited solutions are marked as 'tabu' for a limited number of iterations via a tabu list. In this work, several enhancements to the basic Tabu Search scheme were introduced, including adaptive tenure control, frequency-based penalization, intensification through elite memory, and diversification via shaking mechanisms. These variations are described in the following sections. The algorithm was developed in C++ and uses the same `.dat` input files as in Part I. This ensures consistent input format and allows for direct comparison of results and solving times between the Tabu Search and the CPLEX-based approach.

3.0.1. Solution Representation

The `TSPSolution` struct represents a candidate solution, storing the tour as a `std::vector<int>` that encodes the visiting order of nodes. It includes basic utility methods (e.g., copy constructor, assignment operator, print function) to facilitate manipulation and inspection of solutions.

Problem instances are handled by the `TSP` class, which reads a square grid from file and identifies valid nodes (as previously described, cells marked with 1). It extracts their coordinates and computes a full pairwise distance matrix using Euclidean distances. The class stores the number of nodes and the (symmetric) cost matrix.

3.1. Tabu List representation

A crucial component of any Tabu Search implementation is the tabu list, used to temporarily forbid recently applied moves and thus prevent cycling. In this implementation, the tabu list is realized as a **`std::vector<int>`** storing, for each node, the last iteration in which it was involved in a move (2-opt).

```cpp
std::vector<int> tabuList;
```

```
2
3 void updateTabuList( int nodeFrom, int nodeTo , int iter) {
4   tabuList[nodeFrom] = iter;
5   tabuList[nodeTo] = iter;
6 }
```

A move involving nodes i and j is considered tabu if both were used in the last tabuLength iterations.

This approach offers constant-time (O(1)) updates and queries at each iteration. The logic is encapsulated in the following function:

```cpp
1 bool TSPSolver::isTabu(int nodeFrom, int nodeTo, int iter) {
2     return ((iter - tabuList[nodeFrom] <= tabuLength) &&
3             (iter - tabuList[nodeTo] <= tabuLength));
4 }
```

The tabu tenure (tabuLength) is dynamically adapted during the search, increasing after periods of stagnation (diversification) and decreasing after improvements (intensification), as detailed in Section 3.3.1.

3.2. Neighborhood and Move Evaluation

The search starts from a randomly generated initial solution. At each iteration, the algorithm explores the neighborhood of the current solution by applying a move operator that modifies the tour. The default neighborhood structure is based on **2-opt moves**, where two non-adjacent edges are removed and the resulting paths are reconnected in reversed order. This type of move is efficient to compute and significantly alters the tour structure, promoting deeper local exploration.

Candidate moves are ranked based on their effect on the total cost. The algorithm selects the best admissible move according to the tabu condition, aspiration criterion, and the cost penalty induced by edge frequencies.

Each iteration continues until a predefined maximum number of iterations (`maxIter`) is reached, which constitutes the **stopping criterion** adopted in this implementation.

An **aspiration criterion** is also employed: if a move is classified as tabu but leads to a globally better solution (i.e., improves upon the best-known cost), the move is accepted regardless of its tabu status.

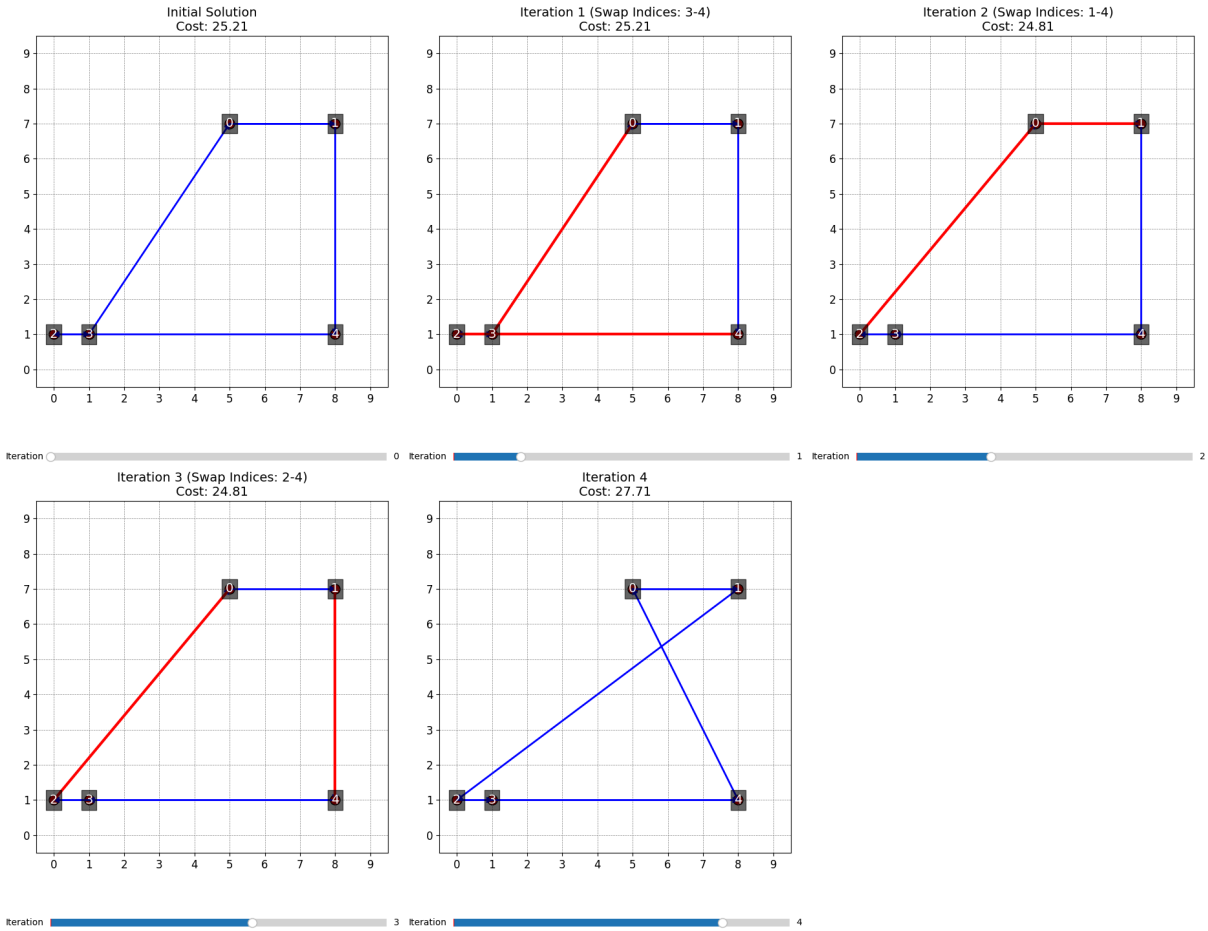

Figure 2: Tabu Search iterations on a 10x10 board with 5 holes. Each figure shows the current tour, the cost, and the 2-opt moves applied in that iteration. The best cost is reached in the 4th iteration, with a total of 5 iterations performed.

3.3. Intensification and diversification

The effectiveness of a Tabu Search algorithm often relies on its ability to balance intensification (deep search in promising areas) and diversification (exploration of new regions of the solution space). This implementation combines several mechanisms to adaptively control the search behavior based on its progress.

3.3.1. Adaptive Tenure - Reactive Search

The tabu tenure is not static, but dynamically adjusted during the search based on the algorithm's recent progress. After discovering a new global best solution, the tenure is temporarily **reduced** (halved) to **intensify** the search around that high-quality region. This allows the algorithm to perform more local moves and explore nearby solutions more thoroughly. The reduction is bounded below by a predefined `minTenure`. Conversely, if no improvement is observed for a number of iterations, the tenure is increased (up to a maximum limit) to promote **diversification**. This encourages the search to move away from over-explored regions. These adjustments are inspired by Reactive Search principles, where the algorithm self-regulates based on its own behavior.

The tenure is also initialized depending on the size and density of the board, providing a baseline sensitivity adapted to instance complexity.

3.3.2. Intensification

A set of **elite solutions** is maintained throughout the search. When the algorithm enters an intensification phase, triggered when the number of consecutive non-improving iterations exceeds a given **threshold** (parameters `alpha` and `beta`), it either restarts from a randomly selected elite solution (intensification) or applies a double-bridge move (diversification), depending on a probabilistic decision. The diversification mechanism is described in more detail in the following Section 3.3.3. Elite memory is maintained as **`std::vector<ScoredSolution> eliteSolutions`**. Whenever a new global best solution is found, it is added to the elite set, replacing the worst solution if the memory is already full. This ensures that the elite memory always contains the highest-quality solutions discovered during the search.

3.3.3. Diversification

To escape local optima, a **double-bridge move** is used as a **shaking** mechanism. This perturbation, commonly found in Iterated Local Search and Tabu Search literature, involves removing four edges and reconnecting the resulting paths in a new order — effectively a **4-opt move**. Since a 4-opt move splits the tour into five segments, there are multiple ways to reorder them. In this implementation, the specific configuration applied is `segment1 + segment3 + segment2 + segment4 + segment5`. This logic is implemented in the function `TSPSolver::applyDoubleBridgeMove(const TSPSolution& sol)`. The resulting solution differs significantly from the original, allowing the algorithm to jump to unexplored regions of the search space. After applying a double-bridge, the tabu list is typically reset to allow fresh exploration.

Additionally, **frequency-based penalties** are applied during move evaluation. A matrix **`std::vector<std::vector<double>> freq`** tracks how often each edge appears in accepted solutions, and all edges receive a penalty based on how frequently they were used in the past. Overused edges incur a larger penalty, which biases the search away from frequently traversed components. This encourages diversification by guiding the search away from repeatedly used components, without overriding the true objective.

The penalty is computed during the evaluation of each candidate 2-opt move. Specifically, for every move that would replace the arcs (h->i) and (j⟶l) with (h⟶j) and (i⟶l), a penalty is computed based on the frequency of the edges involved: `freq[h][i]`, `freq[i][j]`, and `freq[j][l]`. The total penalty is scaled by a tunable parameter **`lambda`**, which controls the strength of the frequency penalty relative to the true objective function:

```cpp
double freqPenalty = lambda * (freq[i][j] + freq[h][i] + freq[j][l]);
double neighCostVariation = - tsp.cost[h][i] - tsp.cost[j][l]
                            + tsp.cost[h][j] + tsp.cost[i][l]
                            + freqPenalty;
```

The value of lambda is determined through parameter tuning and allows adjusting the balance between intensification (following cost improvements) and diversification (penalizing overused paths). A higher lambda encourages more exploration by favoring moves that use rarely visited edges, while a lower value prioritizes pure cost minimization.

The `freq` matrix is updated periodically, not at every iteration. This is done to avoid overwhelming the algorithm with short-term fluctuations and to maintain meaningful penalty values. The update is triggered every `decayInterval` iterations (typically 100), or immediately after a tenure adaptation event. During the update, the frequency of all edges used in the current best solution is incremented proportionally to a tunable parameter **`decayFactor`**.

```cpp
void TSPSolver::updateFrequencies(const TSPSolution& sol) {
    int n = sol.sequence.size() - 1; // exclude return to start
    for (int k = 0; k < n; ++k) {
        int a = sol.sequence[k];
        int b = sol.sequence[k + 1];
        freq[a][b] += decayFactor;
    }
}
```

This means that edges appearing in good-quality solutions are gradually penalized over time. The `decayFactor` controls the magnitude of the update: a higher value leads to stronger reinforcement (and thus stronger penalties later), while a lower value produces weaker and slower growth in frequency. The tuning of `decayFactor` allows fine-grained control over how quickly the search is pushed away from familiar paths.

These components work together to form a reactive and adaptive Tabu Search, capable of both intensifying near elite tours and diversifying when progress stalls.

3.4. Results

3.4.1. Parameters Tuning

To improve the performance of the Tabu Search algorithm across a wide range of instance sizes and densities, a parameter tuning phase was conducted. The following parameters were tuned:

- **`alpha`**: scales the problem size (`tsp.n`, the number of holes) to set `noImproveThreshold`, which is the number of iterations allowed without improvement before taking some action.
- **`beta`**: further scales `noImproveThreshold` to set `tenureAdaptThreshold`, which is the number of iterations before adapting the tabu tenure.

```cpp
int noImproveThreshold = static_cast<int>(alpha * tsp.n);
int tenureAdaptThreshold = static_cast<int>(beta * noImproveThreshold);
```

`noImproveThreshold` is then used to initialize `shakeThreshold`, which is the number of iterations after which we choose randomly to restart from an elite solution (so applying intensification) or to apply a double-bridge move (diversification).

Similarly, `tenureAdaptThreshold` determines how often the tabu tenure (tabu length) is adapted to intensify the search by reducing the tabu length after improvements. By introducing both `alpha` and `beta` parameters, we can independently tune the frequency of these two mechanisms, allowing tenure adaptation (controlled by `beta`) to occur more often or differently than the shaking/intensification step (controlled by `alpha`).

- **`decayFactor`**: controls the rate at which the frequency matrix decays over time.
- **`lambda`**: determines the weight of the frequency-based penalty term added to the objective.

A dedicated C++ tool (**`find_best_parameters.cpp`**) systematically explores combinations of these parameters across a grid of predefined values. For **each pair** of instance size and density, **three boards** are generated (repeated trials with different random seeds). Then, for each board, the algorithm is run multiple times (with different parameters), and the best result is retained. The tool logs the best result per board in a CSV file, recording the final cost, runtime, and parameters used. This data is later used to determine the **optimal configuration for each board class**.

For each combination of board size and density, the script extracts the best result (lowest final cost) and the corresponding parameter values.

Size	Density	Best Cost	Average Cost	Standard Deviation	Alpha	Beta	Decay Factor	Lambda
5	0.15	7.6344	8.1824	0.6154	0.5	0.3	0.85	0.0
5	0.2	7.2361	8.6718	1.2465	0.5	0.3	0.85	0.0
10	0.05	15.6964	20.273	4.0619	0.5	0.3	0.85	0.0
10	0.1	23.6057	29.9345	6.722	0.5	0.3	0.85	0.0
10	0.15	29.8576	36.4861	5.7422	0.5	0.7	0.95	0.05
10	0.2	40.8269	41.7235	0.9482	0.75	0.3	0.85	0.0
15	0.05	45.7997	48.1127	2.1358	0.5	0.3	0.85	0.0
15	0.1	56.2955	60.3912	5.6545	0.5	0.3	0.85	0.0
15	0.15	65.5362	71.6897	5.4906	0.5	0.3	0.85	0.0
15	0.2	86.9872	88.087	0.9803	0.5	0.3	0.85	0.0
20	0.05	68.8313	76.8593	10.9789	1.0	0.5	0.95	0.05
20	0.1	101.1809	104.7478	5.2652	0.5	0.3	0.85	0.01
20	0.15	119.5898	124.7382	6.0537	0.5	0.3	0.85	0.0
20	0.2	140.4367	144.6177	4.9787	0.5	0.5	0.9	0.01
30	0.05	155.0487	161.7169	6.1228	0.5	0.3	0.95	0.0
30	0.1	223.3371	227.8775	5.4099	0.5	0.5	0.95	0.0
30	0.15	281.4842	284.6683	4.4363	0.5	0.7	0.85	0.05
30	0.2	321.7716	327.2175	7.5025	0.5	0.5	0.95	0.05

Table 1: `summary_tuning.csv` — best parameters and performance per board size/density (first ranges explored - keep as definitive)

The parameter space explored during tuning was selected based on recommendations from prior literature on Tabu Search and Reactive Search heuristics, and refined through empirical intuition. The selected ranges were the following:

- `alpha`: [0.5, 0.75, 1.0]
- `beta`: [0.3, 0.5, 0.7]
- `decayFactor`: [0.85, 0.9, 0.95]
- `lambda`: [0.0, 0.01, 0.05]

However, an analysis of the tuning results reveals that optimal configurations tend to cluster near the lower bounds of some tested intervals (as illustrated in Table 1). For example, the value **`alpha` = 0.5** was consistently selected across many instance classes, suggesting that encouraging early diversification often leads to better outcomes. Similarly, **`beta` = 0.3** frequently emerged as the most effective setting, indicating that a relatively slow adaptation of the tabu tenure provides better control over the search dynamics.

Based on the insights from the initial tuning phase, the search was refined by narrowing the parameter ranges to focus on the most promising regions. In particular, we explored `alpha` : [0.35, 0.5, 0.75] and `beta` : [0.2, 0.3, 0.5], which better reflect the values that previously were found to yield the best results.

Size	Density	Best Cost	Average Cost	Standard Deviation	Alpha	Beta	Decay Factor	Lambda
5	0.15	6.4721	7.694	1.0584	0.35	0.2	0.85	0.0
5	0.2	10.8929	11.5275	0.5885	0.35	0.2	0.85	0.0
10	0.05	15.1727	19.6928	3.9676	0.35	0.2	0.85	0.0
10	0.1	16.8191	26.0689	8.4426	0.35	0.2	0.85	0.0
10	0.15	31.117	34.321	2.7832	0.35	0.2	0.85	0.0
10	0.2	33.3651	38.0803	4.0838	0.35	0.2	0.85	0.0
15	0.05	32.3014	40.2812	6.9108	0.35	0.2	0.85	0.0
15	0.1	56.6747	59.2676	4.2072	0.5	0.2	0.85	0.0
15	0.15	71.7098	72.7935	1.2639	0.5	0.2	0.95	0.0
15	0.2	81.8544	86.1068	4.4892	0.35	0.2	0.95	0.0
20	0.05	74.3079	75.8986	1.4409	0.35	0.2	0.85	0.0
20	0.1	92.9431	99.8308	8.6406	0.35	0.2	0.85	0.0
20	0.15	120.7813	125.3677	3.9891	0.35	0.2	0.85	0.0
20	0.2	138.7981	144.8844	6.2203	0.5	0.2	0.9	0.01
30	0.05	158.1184	159.8046	1.4604	0.35	0.3	0.95	0.01
30	0.1	220.8691	228.4078	7.0561	0.75	0.5	0.9	0.01
30	0.15	276.6466	282.716	5.6777	0.35	0.2	0.9	0.0
30	0.2	329.4913	332.311	2.7808	0.35	0.3	0.85	0.0

Table 2: `summary_tuning.csv` — best parameters and performance per board size/density (refined ranges)

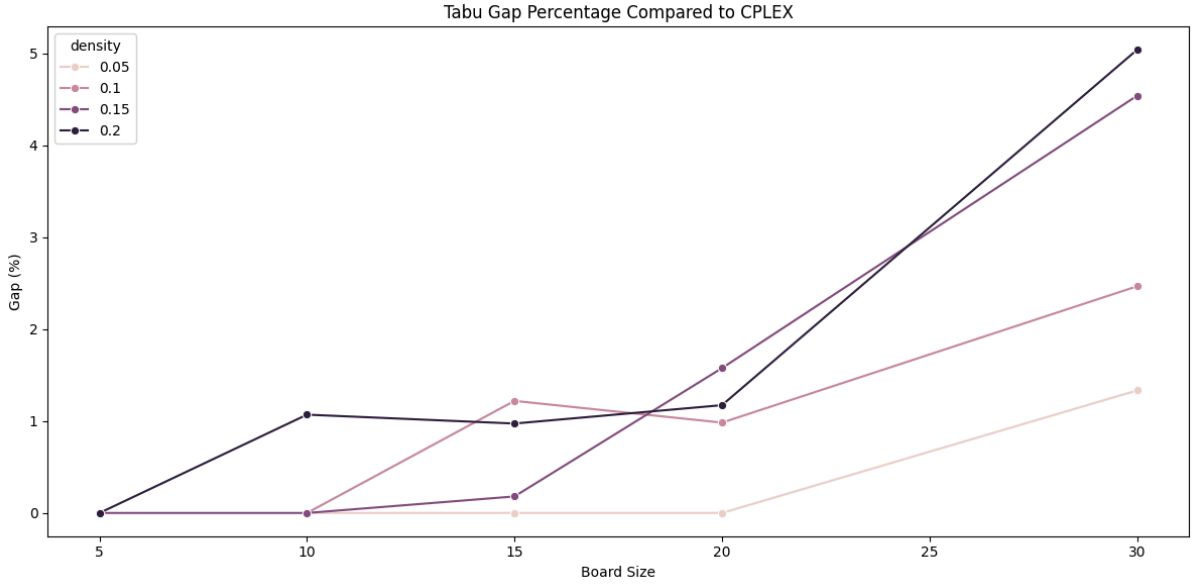

Figure 3: performance gap between Tabu Search and CPLEX for each board configuration - first ranges explored

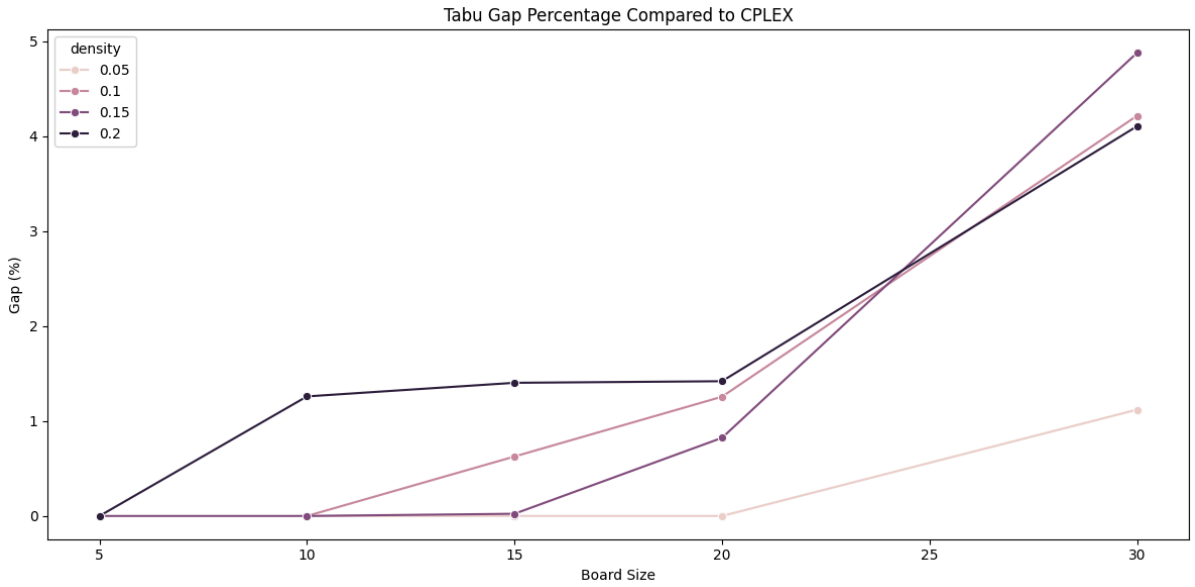

Figure 4: performance gap between Tabu Search and CPLEX for each board configuration - refined ranges

A comparison of the two tuning phases shows that, although small performance variations may be partially due to randomness in board generation, the original parameter ranges consistently yield better results across most configurations. Only a few combinations from the refined search outperform the initial settings. Therefore, the first tuning phase is retained as the final configuration.

3.4.2. Benchmarking framework

To assess the performance of the Tabu Search algorithm against the exact method developed in Part I (CPLEX), a dedicated benchmarking script was developed. This tool runs both solvers on a common set of randomly generated instances and collects **solving time** and **final cost** for each.

During benchmarking, the script **`run_experiments.cpp`** reads from `summary_tuning.csv` to extract the best parameters for Tabu Search for each (size, density) pair. For each configuration, **three** distinct **board instances** are generated. Each instance is then solved by both the Tabu Search (with tuned parameters) and the CPLEX-based solver.

Each run is timed, and the final cost is parsed from the solver output. All results are saved in a `benchmark_results.csv` file, which includes solver name, board features, repeat index, final cost, and solving time.

3.5. Performance gap analysis

After collecting the results from both solvers, we compute the **performance gap** to quantify the difference between the heuristic (Tabu Search) and the exact (CPLEX) solutions. The performance gap is defined as the percentage difference between the cost found by Tabu Search and the optimal cost found by CPLEX, relative to the CPLEX cost. For example, if Tabu Search finds a solution with the same cost as CPLEX, the gap is 0%.

The script **`results_analysis.py`** computes this gap for each instance and aggregates the results by board size and density, reporting the mean and standard deviation of the gap across all test cases

Size	Density	Mean Gap %	Std Gap %	Mean Time Gap (s)
5	0.15	0	0	0.09659
5	0.2	0	0	0.29098
10	0.05	0	0	0.2326
10	0.1	0	0	0.81827
10	0.15	0	0	0.93293
10	0.2	1.07	1	1.27719
15	0.05	0	0	0.52025
15	0.1	1.22	0.59	1.88981
15	0.15	0.18	0.31	3.71452
15	0.2	0.97	0.92	5.01147
20	0.05	0	0	1.79556
20	0.1	0.98	0.99	2.39261
20	0.15	1.58	0.58	11.15606
20	0.2	1.17	0.28	64.62243
30	0.05	1.33	1.71	9.26047
30	0.1	2.47	1.13	115.53437
30	0.15	4.54	3.08	553.80055
30	0.2	5.04	1.88	2284.81992

Table 3: `benchmark_gap_summary.csv` — mean and standard deviation of the performance gap between Tabu Search and CPLEX per board configuration

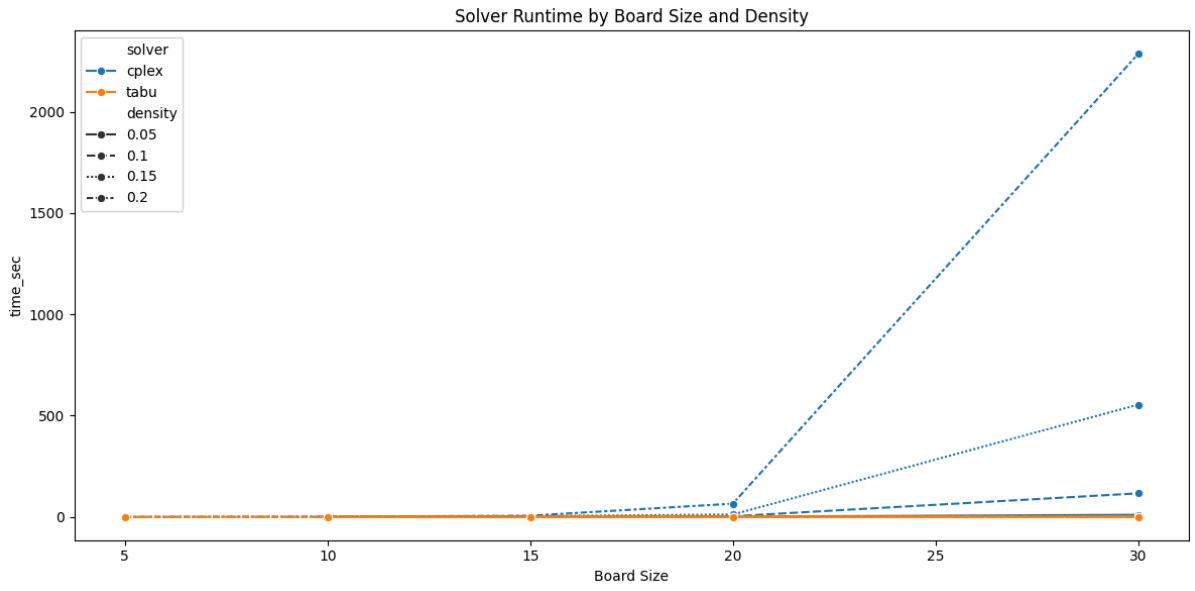

Figure 5: runtime comparison between Tabu Search and CPLEX for each board configuration

The **runtime** analysis highlights the scalability of Tabu Search: while CPLEX runtime grows sharply with board size and density, Tabu Search maintains nearly constant performance. Considering the application context and the observed runtime gap, it is acceptable to use CPLEX for instances up to approximately 15x15 in size. Beyond this threshold, the runtime becomes prohibitive for practical use.

The results show that for small and medium-sized boards, Tabu Search consistently produces solutions within a few percent of the optimal cost computed by CPLEX. As the board size and density increase, the performance gap tends to grow, yet remains within acceptable limits for practical applications, particularly when considering the significant runtime advantage of Tabu Search.

An examination of the best parameters reveals that smaller boards tend to benefit from configurations with **lower** values of **alpha** and **beta**, **no penalization** (**lambda** equal to zero), and standard decay factors, indicating that in **simpler scenarios** a more **intensification**-driven search without penalties is sufficient. Conversely, as the board size increases, the optimal configurations shift toward **higher alpha** values and **more frequent decay resets**, suggesting a greater need for **diversification** and memory to effectively navigate more complex solution spaces. The use of non-zero `lambda` values in larger instances further supports the idea that penalizing frequently visited moves becomes useful only when the instance size makes revisiting more likely.

4. Conclusion

This project addressed the Traveling Salesman Problem in the context of PCB drilling optimization, comparing an exact CPLEX-based solver with a metaheuristic Tabu Search approach. While CPLEX always returns optimal solutions, its runtime grows quickly with instance size and density, making it unsuitable for large-scale configurations. Tabu Search, in contrast, provides high-quality solutions in a fraction of the time.

Quantitatively, the Tabu Search algorithm consistently finds solutions with a gap below 2% for instances up to size 20. For the largest size (30x30), the mean gap rises gradually, reaching 5.04% in the densest case. Despite this increase, the trade-off is justified by a substantial runtime advantage.

The performance of Tabu Search heavily depends on parameter tuning, with specific configurations showing better robustness across different board densities.

In conclusion, Tabu Search emerges as a scalable and efficient heuristic, capable of producing near-optimal solutions even for large, complex instances where exact solvers are computationally prohibitive.

5. Setup and execution instructions

This section outlines the steps necessary to compile and run the entire Tabu Search and CPLEX-based benchmarking pipeline, from board generation to result visualization.

5.1. Compilation

The repository is organized into modular components, each with its own `main.cpp` file and local `Makefile`. Compilation must therefore be performed in three separate stages:

```bash
# Step 1: Compile CPLEX-based tools and the board generator
cd part1
make
cd ..

# Step 2: Compile the Tabu Search solver
cd part2
make
cd ..

```

```
# Step 3: Compile the parameter tuning and experiments framework
make
```

5.2. Board generation

To create a board instance for testing:

```bash
part1/generate_board.out <size> <num_holes> <output_file.dat>
```

- `<size>`: the side length of the square board.
- `<num_holes>`: number of drill points to place randomly on the board.
- `<output_file.dat>`: output path for the generated board file.

5.3. Solving an Instance

Once a board has been generated, it can be solved using either of the two solvers:

CPLEX-based optimal solver:

```bash
part1/main_cplex.out <board_file.dat>
```

Tabu Search solver:

```bash
part2/main_tabu.out <board_file.dat> --alpha=... --beta=... --
decayFactor=... --lambda=...
```

The Tabu Search accepts hyperparameters as command-line arguments. These can be tuned manually or retrieved from tuning results.

5.4. Parameter Tuning

To perform automated parameter tuning:

```bash
./find_best_parameters.out
```

This tool not only tests various combinations of parameters across multiple configurations but also generates the corresponding board instances internally. It stores the best results in tuning_results.csv.

To aggregate the tuning results:

```bash
python summarize_results.py  # Produces summary_tuning.csv
```

Requirements to run this script in laboratory environment:

```bash
python -m virtualenv venv
source venv/bin/activate
pip install pandas
```

5.5. Running Benchmark Experiments

To run the full set of benchmarking experiments comparing Tabu Search to CPLEX:

```bash
./run_experiments.out
```

This script loads the best parameters from `summary_tuning.csv`, generates the required boards (using the same board generation tool), runs both solvers, and logs costs and runtimes in `benchmark_results.csv`.

5.6. Result analysis

To quantitatively assess the quality of Tabu Search solutions, the **`results_analysis.py`** script computes the performance gap between the heuristic and the optimal CPLEX results across all instances. Final summaries are saved in a CSV file named `benchmark_gap_summary.csv`.

The data can be further visualized using the companion plotting script **`plot_results.py`**.

5.7. Visualization

The pipeline includes visualization tools to inspect solver behavior:

CPLEX Solution Path (XML):

```bash
python part1/visualize_drill_path.py <basename>
# Requires <basename>.dat and <basename>.sol
```

The result will be a grid representation of the board and the path, as shown in Figure 1.

Tabu Search Tour Evolution:

```bash
python part2/visualize_ts.py <basename>
# Requires <basename>.dat and <basename>_log.txt
```

This allow to visualize each step of the Tabu Search, showing how the tour evolves over iterations, including also the 2-opt moves applied and the current cost at each step (Figure 2).