Comparison between CPLEX and Metaheuristic for solving Symmetric TSP Drilling Problem

Methods and Models for Combinatorial Optimization

Student: Gabriel Rovesti - *ID Number* - 2103389
Supervisor: Prof. Luigi de Giovanni

A.Y. 2024/2025

Table of contents

Figures

List of Tables

1. Part I: Exact Method Implementation

1.1. Problem Description and Mathematical Formulation

A company manufactures circuit boards with pre-drilled holes to be used to construct electric panels. The process involves placing boards over a machine where a drill travels across the surface, stops at predetermined positions, and creates holes. Once a board is completely drilled, it is replaced by another one and the process is repeated. The company is interested in finding the optimum process by determining the best sequence of drilling operations for which the total time will be minimal, considering that creating each hole takes the same amount of time.

This optimization challenge can be mathematically represented as a *Traveling Salesman Problem (TSP)* on a weighted complete graph $G = (N, A)$, where:

- N represents the set of nodes corresponding to the positions where holes must be drilled
- A represents the set of arcs $(i, j), \forall i, j \in N$, corresponding to the drill's movement trajectory from hole i to hole j
- Each arc (i, j) has an associated weight c_{ij} representing the time needed for the drill to move from position i to position j

The optimal solution to this problem is the Hamiltonian cycle on G that minimizes the total weight, representing the most efficient drilling sequence.

The model proposed comes from the Gavish-Graves paper[1] in 1978, described generally as follows:

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

subject to:

$$\sum_{i:(i,k)\in A} x_{ik} - \sum_{j:(k,j),j\neq 0} x_{kj} = 1 \quad \forall k \in N \setminus \{0\} \tag{2}$$

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

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

$$x_{ij} \leq (|N| - 1) y_{ij} \quad \forall (i, j) \in A, j \neq 0 \tag{5}$$

$$x_{ij} \in \mathbb{R}^+ \quad \forall (i, j) \in A, j \neq 0 \tag{6}$$

$$y_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \tag{7}$$

Sets:

- $N =$ graph nodes, corresponding to the holes positions
- $A =$ arcs $(i, j), \forall i, j \in N$, serving as the movement from hole i to hole j

Parameters:

- $c_{ij} =$ time to move from i to $j, \forall (i, j) \in A$
- $0 =$ arbitrary starting node, $0 \in N$

Decision variables:

- $x_{ij} =$ amount of flow shipped from i to $j, \forall (i, j) \in A$

[1] https://dspace.mit.edu/handle/1721.1/5363

- $y_{ij} = 1$ if arc (i, j) ships some flow, 0 otherwise, $\forall (i, j) \in A$

where, in the model:

- x_{ij} represents the amount of flow on arc (i, j)
- y_{ij} is a binary variable indicating whether arc (i, j) is in the solution path
- Constraint (Equation 2) ensures flow conservation
- Constraints (Equation 3) and (Equation 4) enforce exactly one incoming and outgoing arc per node
- Constraint (Equation 5) links the flow and path variables
- Constraints (Equation 6) and (Equation 7) define variable domains

1.2. Implementation with CPLEX

Implementation will focus on modeling a circuit board drilling optimization problem as the Traveling Salesman Problem, where the drill head is considered a traveling salesman, and positions of holes-are cities to visit. This framework is specifically designed to cater to the constraints of manufacturing of PCBs and actual-world requirements for the layout.

1.2.1. Core architecture

The implementation follows a three-tier design pattern distributed across key source files:

- *model.h/model.cpp (TSPModel)*: This component implements the flow-based mathematical formulation adapted from the Gavish-Graves paper for the TSP. The core model is encapsulated in the TSPModel class, which handles both variable creation and constraint generation through CPLEX's APIs.
- *data_generator.h (TSPGenerator)*: The TSPGenerator class creates realistic PCB drilling instances while enforcing manufacturing constraints. The instances generated by this class are present within the *data* folder.
- *cpxmacro.h*: Provides CPLEX interface abstractions and error handling
- *main.cpp*: Implementation of the exact solution approach, which handles multiple board configurations and provides detailed performance analysis, with adaptive time management, with outputs present within the *results* folder.

The core model implementation revolves around the *TSPModel* class, which handles the optimization logic through several carefully structured components:

```cpp
class TSPModel {
private:
    // Efficient O(1) lookup matrices for CPLEX variable indices
    std::vector<std::vector<int>> map_x;  // Flow variables mapping
    std::vector<std::vector<int>> map_y;  // Path variables mapping

protected:
    // Core model construction methods
    void setupVariables(CEnv env, Prob lp, int N,
                        const std::vector<std::vector<double>>& costs);
    void setupFlowConservation(CEnv env, Prob lp, int N);
    void setupAssignmentConstraints(CEnv env, Prob lp, int N);
    void setupLinkingConstraints(CEnv env, Prob lp, int N);
};
```

The class implements the Gavish-Graves formulation conserving the flow ingoing and outgoing and handling systematically, so to optimize the sequence of drilling operations done on PCB (printed circuit boards). To describe precisely what happens inside of this class:

- Flow and path variables are created (first continuous, second ones binary)
- Flow is conserved at each node. The flow entering a node minus the flow exiting it is fixed to 1 for nodes other than the depot (subtour elimination)
- Each node has to be visited only once and the flow gets bounded to the network size

1.2.2. Instance generation framework

The *TSPGenerator* class implements the instance generation functionality, incorporating real-world manufacturing constraints. It manages board specifications through precisely defined constants that reflect industry standards, since board holes are not randomly distributed but follow specific patterns dictated by electronic components and manufacturing requirements. Board configurations are systematically defined through a carefully structured tuple system:

```
std::vector<std::tuple<int, int, int>> board_configs = {
    {50, 50, 2},     // Small boards
    {75, 75, 3},     // Medium-small boards
    {100, 100, 3},   // Medium boards
    {125, 125, 4},   // Medium-large boards
    {150, 150, 5}    // Large boards
};
```

Each configuration encapsulates three essential parameters: board width, board height, and the number of components to be placed. This structured approach reflects real manufacturing scenarios where board sizes are standardized and component density follows practical limits. Each configuration encapsulates three essential parameters: board width, board height, and the number of components to be placed. This structured approach reflects real manufacturing scenarios where board sizes are standardized and component density follows practical limits.

```
enum class BoardPattern {
    DIP_IC,         // Dual In-line Package / Integrated Circuit
    SOIC,           // Small Outline Integrated Circuit
    CONNECTOR,      // Edge connector
    MOUNTING,       // Mounting holes
    VIA,            // Through-hole vias
    CUSTOM          // Custom pattern
};
```

Each pattern represents a common circuit board component with specific hole arrangements. The implementation then creates standardized patterns that mirror actual electronic components, using common practices in PCB (Printed Circuit Board) manufacturing via defined constants.

1. *DIP (Dual In-line Package) Components*:

- Standard 2.54mm (0.1 inch) pin spacing
- Parallel rows of pins for through-hole mounting
- Example: 14-pin DIP IC requires 14 holes arranged in two rows of 7

```
{BoardPattern::DIP_IC,
 {Point(0,0),  Point(0,2.54),  Point(0,5.08),  Point(0,7.62),
  Point(7.62,0), Point(7.62,2.54), Point(7.62,5.08), Point(7.62,7.62)},
 2.54, // Minimum pin spacing
 "14-pin DIP IC"}
```

2. SOIC (Small Outline Integrated Circuit):

- 1.27mm pin spacing (half of DIP standard)
- Surface mount footprint conversion for through-hole drilling
- Tighter spacing requires precise drill positioning

```
{BoardPattern::SOIC,
 {Point(0,0), Point(0,1.27), Point(0,2.54), Point(0,3.81),
  Point(5.08,0), Point(5.08,1.27), Point(5.08,2.54), Point(5.08,3.81)},
 1.27, // Critical minimum spacing for SMT conversion
 "8-pin SOIC"}
```

To keep integrity with the representations, manufacturing safety constraints are to be respected precisely, with the following enforcements:

- Edge clearance of 5mm to prevent board damage
- Minimum hole spacing of 2.54mm for structural integrity
- Mounting hole placement at standardized board corners

The *TSPGenerator*'s core is built around representing real PCB components through two key structures:

```
struct Component {
    BoardPattern type;           // Component type (DIP, SOIC etc)
    std::vector<Point> holes;    // Hole pattern template
    double min_spacing;          // Required spacing between components
};

static constexpr double MIN_HOLE_SPACING = 2.54;  // Industry standard pitch
static constexpr double EDGE_MARGIN = 5.0;        // Required board clearance
```

The process follows these steps:

- *Board configuration*: The *generateCircuitBoard* function takes parameters for board width, height, and the number of components to place. This allows generating instances of varying size and complexity.
- *Mounting hole placement*: As the first step, the generator always places four mounting holes at the corners of the board, respecting the EDGE_MARGIN constant (usually 5mm). This is a non-negotiable requirement for PCB assembly. The mounting hole positions are added to the hole_positions vector.
- *Component placement*: The generator then iteratively places electronic components on the board:
  - It randomly selects a component type (DIP, SOIC, connector, etc.) from the predefined patterns.
  - For each component, it attempts to find a valid position on the board that respects the minimum spacing between components (*min_spacing*) and the edge clearance (*EDGE_MARGIN*).
  - If a valid position is found, the generator adds the holes for that component (defined in the holes vector of the *Component*) to the *hole_positions* vector.
  - This process repeats until the requested number of components have been placed or a maximum number of placement attempts is reached.

The generated distance matrix, along with metadata about the board size and number of holes, is saved to a *.dat* file using the *saveToFile* function. The naming convention includes the board dimensions and a timestamp to ensure unique filenames. An example appears here:

```
17
# Circuit board instance
Size: small
Nodes: 17
0.000000 40.000000 40.000000 56.568542 19.055934 19.590694 20.191280 20.852005
```

```
23.809891 24.240000 24.727921 25.270306 20.876381 23.385175 25.900052 28.419398
30.942121

40.000000 0.000000 56.568542 40.000000 23.701134 24.133181 24.623219 25.167861
18.950620 19.488271 20.091919 20.755806 19.715811 17.220810 14.741187 12.286256
9.874450
...
```

Such files are create dynamically in order to represent realistic instances of boards to be generated; the same logic is then applied for the second part of this project, so to compare the results between the two, using the logic of the same *TSPGenerator* class.

1.2.3. Data creation and problem solving

With a set of realistic test cases available in .dat files, the *main* file orchestrates the process of loading, solving, and analyzing each drilling problem instance.

- For each input .dat file, main loads the problem data into a TSP object which stores the number of holes (N), the distance matrix (cost), and a sentinel value (infinite) representing an invalid/absent distance.
- It then instantiates a *TSPModel* object and invokes its *createModel* function, passing in the raw problem data. This function sets up the Gavish & Graves flow-based formulation in CPLEX, creating the necessary variables and constraints.
- The solve function of *TSPModel* is then called with increasing time limits (0.1s, 1s, 10s). It invokes CPLEX to find an optimal drilling sequence that minimizes total travel distance. The resulting objective value (total distance) and the hole visiting sequence are retrieved.
- Main reports the problem size, CPLEX solution status (optimal or time-limit exceeded), optimality gap, and measures of solution quality (total/average travel distance) for each time limit.

To represent a diverse range of scenarios, the boards were parameterized by three key dimensions:

- Width: The horizontal size of the board, ranging from 50mm to 150mm
- Height: The vertical size, also spanning 50mm to 150mm
- Component count: The number of electrical components to place, varying from 2 on the smallest boards to 5 on the largest

By systematically varying these parameters, a comprehensive test suite was created, covering small (50x50mm), medium (75x75 to 125x125mm), and large (150x150mm) board sizes. Increasing the component count along with board dimensions allowed maintaining a realistic component density across all sizes. As an example, a 50x50 output is shown here, as what the console output would display after the creation of the instances:

```
Board Manufacturing Specifications:
- Dimensions: 50x50 mm
- Components: 2
- Total holes: 26
- Min hole spacing: 2.54 mm
- Edge clearance: 5 mm

Time limit 0.1s:
Performance Metrics:
- Model setup time: 0.012525 seconds
- Solution time: 0.121101 seconds
- Total time: 0.133626 seconds
- Optimality: Not optimal
- Gap: 31.37%
```

```
Solution Quality:
- Total drilling path length: 284.39 mm
- Average distance between holes: 10.94 mm

Drilling sequence: 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 -> 12 -
> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21 -> 22 -> 23 -> 24 -> 25 -> 0
```

1.3. Computational Results

To evaluate the performance and solution quality of the exact approach, the Gavish-Graves MILP model was tested on a suite of 50 realistic test instances covering a range of PCB sizes and component densities. The results are analyzed here in terms of several key dimensions: CPLEX solver performance, solution quality, and scaling behavior.

1.3.1. Solution Quality and Optimality

The table below summarizes the optimality and solution quality results for a representative subset of the test instances across the different board size categories. For each instance, the number of holes, optimality gap, objective value (total travel distance), and average distance between holes are reported for time limits of 0.1s, 1s, and 10s.

Instance size	**Holes**	**Time**	**Gap**	**Objective**
50x50	26	*0.1s*	31.37%	284.39
		1s	Optimal	201.32
		10s	Optimal	201.32
75x75	28	*0.1s*	33.65%	453.94
		1s	Optimal	310.43
		10s	Optimal	310.43
100x100	37	*0.1s*	100.00%	620.84
		1s	17.39%	509.57
		10s	0.01%	465.25
125x125	51	*0.1s*	100.00%	896.91
		1s	41.73%	896.91
		10s	2.55%	571.46
150x150	47	*0.1s*	100.00%	928.97
		1s	34.37%	928.97
		10s	13.74%	748.73

Table 1: Computational performance across board sizes

The results showcase several key trends:

- Small instances solve quickly to optimality. For the 50x50mm board with 26 holes, CPLEX finds and proves the optimal solution within 1 second.
- Medium instances are often still tractable. The 75x75mm board with 28 holes is also solved to optimality, although it requires closer to the full 1 second.
- Large instances are challenging to solve definitively. For the 100x100mm, 125x125mm, and 150x150mm boards with 37-51 holes, optimality is not achieved even with 10 seconds, although the *optimality gaps* (differences between the best solution found by CPLEX within a given time limit and the optimal solution to the problem) progressively diminish over time, indicating that CPLEX continues to improve the solution quality given sufficient runtime.
- Solution quality improves dramatically with more time, especially in the first second. For example, on the 150x150mm instance, the solution found in 0.1s has a 100% optimality gap, which drops to 34.37% in 1s and further to 13.74% in 10s, representing substantial travel distance savings. This represents substantial savings in the total travel distance of the drill.

1.3.2. Performance Analysis

Solver performance was evaluated by tracking the time needed to set up and solve the MILP model. The table below reports the setup time, solve time, and total runtime for the same representative test instances. This is to be evidenced by the produced *.txt* files present within the *results* folder of this project.

Instance size	**Holes**	**Setup Time (s)**	**Solve Time (s)**	**Total Time (s)**
50x50	26	0.01	0.12	0.11
75x75	28	0.01	1.03	1.03
100x100	37	0.02	7.75	7.76
125x125	51	0.03	10.03	10.06
150x150	47	0.02	10.07	10.09

Table 2: Computational performance metrics for different board sizes

The results show that the setup time is still very low, normally within the range of 0.01 to 0.03 seconds, while the problem size increases from a small board, 50x50mm, to a large one, 150x150mm. This reflects that the construction of the MILP model and data handling is very efficient and scalable, not really affecting the overall runtime. By contrast, solve time is strongly dependent on problem size and complexity: As the number of holes and the dimensions of the board increase, the solve time dominates the total runtime. For example, in the case of the 100x100mm board with 37 holes, 7.75 seconds of the 7.76-second total are used up by solving. Similarly, for the 150x150mm board with 47 holes, 10.07 seconds of the 10.09-second total are used up by solving.

1.3.3. Scalability assessment

The exact method demonstrated very good scalability for the small to medium instances, finding optimality in a few seconds. For larger problems considered here, performance degraded significantly.

- We observe that for larger boards, 100x100mm and above, CPLEX cannot prove optimality within the maximum time limit of 10 seconds. For example, for the 125x125mm instance with 51 holes, CPLEX stopped after a gap of 2.55% after 10 seconds, whereas the 150x150mm board with 47 holes has a gap of 13.74% after 10 seconds.
- This agrees well with expectations, since TSP is an NP-hard problem. For larger problem sizes, the number of possible solutions (hole sequences) increases exponentially. Both the time found within the instances creation and problem solution is in line with the nature of the actual problem.
- These limits of scalability suggest that exact methods are most suitable for small or medium-scale PCB drilling problems, whereby in generally reasonable time frames optimality can be achieved. In the case of larger boards with a lot of holes, the computation of optimal solutions could be prohibitively expensive; hence, heuristic or approximate approaches may be necessary.

2. Part II: Tabu Search Implementation

2.1. Introduction

The previous part highlighted the scalability challenges in using using exact methods to solve the TSP problem. As the number of holes and board dimensions increase, the runtime required to achieve optimality grows exponentially, making it impractical for real-world scenarios with tight production deadlines.

To address this limitation, the *Tabu Search*[2] approach is used, in order to efficiently explore the solution space and find high-quality drilling sequences within acceptable timeframes. Tabu Search is a local search-based optimization technique that iteratively moves from one solution to another in the neighborhood, while using memory structures to guide the search and avoid getting trapped in suboptimal regions: introducing temporary prohibitions to certain moves (tabu) prevent the search from revisiting recently explored areas (thus becoming stuck), encouraging the exploration of new regions and the discovery of better solutions.

The presented Tabu Search implementation for the PCB drilling problem incorporates several problem-specific adaptations and enhancements to improve its effectiveness and efficiency, based on what seen during the classes:

- A custom neighborhood structure based on 2-opt and 3-opt moves is defined (keeping as base the Lin-Kerninghan heuristic[3]), which involve swapping or rearranging a subset of the holes in the current drilling sequence. These moves allow for local perturbations while maintaining the overall structure of the solution.
- Instead of a fixed-size tabu list, we employ a *dynamic tabu tenure* that adjusts the length of the prohibition period based on the search progress. This allows for a balance between intensification (exploiting promising regions) and diversification (exploring new areas) during the search.
- An *aspiration criterion* is introduced that overrides the tabu status of a move if it leads to a solution better than the currently known best solution (TSAC). This ensures that promising moves are not overlooked due to the tabu restrictions.
- To further guide the search, we incorporate *intensification* and *diversification* mechanisms. Intensification focuses on thoroughly exploring promising regions of the solution space by

[2]https://www.ida.liu.se/~zebpe83/heuristic/papers/TS_tutorial.pdf

[3]https://pubsonline.informs.org/doi/10.1287/opre.21.2.498

temporarily reducing the tabu tenure and allowing more localized search. Diversification, on the other hand, encourages the exploration of unvisited regions by applying perturbations to the current solution or restarting the search from a new initial solution.
- Instead of a fixed number of iterations, we employ an *adaptive termination criterion* based on the search progress. If the best solution does not improve for a specified number of iterations, the search is terminated to avoid wasting computational resources on unproductive explorations.

The actual implementation will be documented and explained better in the subsequent subsections. It's mainly composed of the following:
- *TSPSolver.h/TSPSolver.cpp*: This class serves as the heart of the Tabu Search implementation, containing the complete search logic and optimization mechanisms. It implements both the basic local search framework and advanced features like reactive tabu tenure and memory structures
- *TSPSolution.h/TSPSolution.cpp*: This component provides the solution representation for the drilling sequence optimization. The solution is encoded as a vector of indices where both first and last elements are fixed at 0, representing the drill's home position. The solution representation is immutable after construction, with modifications happening through controlled move operations in the solver
- *TSP.h*: It encapsulates the problem instance data structure. It stores the number of holes (n), the cost matrix representing distances between holes, and an infinite value used as an upper bound for invalid solutions.
- *visualization.h (BoardVisualizer)*: This component generates SVG representations of board layouts and drilling sequences. The *BoardVisualizer* class can create both static visualizations of individual solutions and comparative views showing the progress of the optimization process.
- *parameter_calibration.h/parameter_calibration.cpp*: Implements an automated parameter tuning system that systematically evaluates different parameter combinations to find optimal settings for various instance sizes. The calibrator considers factors like tabu tenure, maximum iterations, and intensification/diversification triggers.
- *data_generator.h (TSPGenerator)*: Carries forward the instance generation functionality from Part I, ensuring consistent problem instances for comparative analysis. As before, the instances being created are present within the *data* folder.
- *main.cpp*: The orchestrator for the Tabu Search solution process, providing a comprehensive framework for testing, parameter calibration, and performance analysis, creating a hierarchical structure for results organizations, present in *visualizations* and *results* folder.

2.2. Algorithm Design

The algorithm begins with an initial solution generated either randomly or using a nearest neighbor heuristic, then systematically explores the solution space through a series of local modifications. A critical innovation in our implementation is the use of a dynamic tabu tenure mechanism that automatically adjusts based on search progress, helping balance intensification and diversification of the search process. The core search logic is implemented in the *solveWithTabuSearch* method, which orchestrates the interaction between various components:

```
bool TSPSolver::solveWithTabuSearch(const TSP& tsp, const TSPSolution& initSol,
    TSPSolution& bestSol, const std::vector<std::pair<double, double>>& points) {

    initializeMemoryStructures(tsp.n);
    TSPSolution currSol(initSol);
    double bestValue = evaluate(currSol, tsp);

    while (!stop) {
        Move move = findBestNeighbor(tsp, currSol, iteration);
        if (shouldAcceptMove(move)) {
            updateTabuList(currSol.sequence[move.from],
                           currSol.sequence[move.to], iteration);
            currSol = applyMove(currSol, move);

            if (shouldIntensify(currValue, bestValue)) {
                intensifySearch(tsp, currSol);
            }
            else if (shouldDiversify()) {
                diversifySearch(currSol);
            }
        }
        adjustTabuTenure(currValue);
    }
}
```

2.2.1. Solution Representation

The search process is guided considering each solution is encoded as a sequence of hole indices representing the drilling order. A key design decision was to always maintain the start/end point (index 0) as fixed, reflecting the physical constraint of the drill's home position. This structure is encapsulated in the TSPSolution class:

```
class TSPSolution {
public:
    std::vector<int> sequence;  // drilling sequence
    // First and last elements are always 0 (home position)
    TSPSolution(const TSP& tsp) {
        sequence.reserve(tsp.n + 1);
        for (int i = 0; i < tsp.n; i++) {
            sequence.push_back(i);
        }
        sequence.push_back(0);  // Return to home position
    }
};
```

The representation mirrors the actual drilling path, making it intuitive to understand and validate while making it suitable for manufacturing constraints like minimum hole spacing, which are handled implicitly during cost evaluation rather than through complex sequence validation.

2.2.2. Neighborhood Structure

The neighborhood structure defines how the algorithm explores the solution space through local modifications, being particularly relevant for the effectiveness of the TS implementation. It centers on an enhanced 2-opt move operator, which systematically improves the drilling sequence by reversing subsequences when beneficial. The neighborhood is defined by considering all possible 2-opt moves from the current solution. The move evaluation process is implemented through the *findBestNeighbor* method:

```
Move TSPSolver::findBestNeighbor(const TSP& tsp, const TSPSolution& currSol,
    int iteration) {
    Move bestMove;
    bestMove.cost_change = tsp.infinite;

    for (std::size_t a = 1; a < currSol.sequence.size() - 2; a++) {
        int h = currSol.sequence[a - 1];  // Node before potential reversal
        int i = currSol.sequence[a];      // First node to reverse
        for (std::size_t b = a + 1; b < currSol.sequence.size() - 1; b++) {
            int j = currSol.sequence[b];     // Last node to reverse
            int l = currSol.sequence[b + 1]; // Node after potential reversal

            if (!isTabu(i, j, iteration)) {
                double costChange = calculateMoveCost(tsp, currSol, Move(a, b));
                if (costChange < bestMove.cost_change) {
                    bestMove.from = static_cast<int>(a);
                    bestMove.to = static_cast<int>(b);
                    bestMove.cost_change = costChange;
                }
            }
        }
    }
    return bestMove;
}
```

The move cost calculation achieves $O(1)$ complexity per move evaluation:

```
double calculateMoveCost(const TSP& tsp, const TSPSolution& sol, const Move& move) {
    int h = sol.sequence[move.from - 1];  // Previous node
    int i = sol.sequence[move.from];      // First reversed
    int j = sol.sequence[move.to];        // Last reversed
    int l = sol.sequence[move.to + 1];    // Next node

    // Only need to consider edges that change
    return -tsp.cost[h][i] - tsp.cost[j][l]     // Remove old edges
           +tsp.cost[h][j] + tsp.cost[i][l];    // Add new edges
}
```

The neighborhood structure naturally preserves the drill's home position constraint by never modifying the first and last elements of the sequence.

2.2.3. Memory Structures

The implementation present here employs three complementary memory mechanisms that work together to guide the search process efficiently:

1. Short-term Memory (Tabu List):

```
class TSPSolver {
private:
    std::deque<std::pair<int, int>> tabu_list;
    int tabu_tenure;

    void updateTabuList(int from, int to, int iteration) {
        tabu_list.push_back(std::make_pair(from, to));
        if (tabu_list.size() > tabu_tenure) {
            tabu_list.pop_front();
        }
        frequency_matrix[from][to]++;
        frequency_matrix[to][from]++;
    }
};
```

The tabu list prevents cycling by temporarily prohibiting recently explored moves. Its size is dynamically adjusted based on search progress through the tabu tenure mechanism.

2. Medium-term Memory (Frequency Matrix):

```
std::vector<std::vector<int>> frequency_matrix;
std::map<std::pair<int, int>, MoveFrequency> move_history;
```

The frequency matrix keeps the frequency with which certain moves have been executed along the whole search process up to that moment. This proves really valuable for intensification and diversification strategies. When the frequency of some moves seems too high, this may indicate cycling in some local region, and this can be used by the algorithm to force the exploration of less-visited areas of the solution space. Move history also keeps the average improvement each move provided, adding context to the choice.

3. Long-term Memory (Search Statistics):

```
struct SearchStats {
    int iteration;
    double solution_value;
    int current_tenure;
    bool was_improvement;
    double improvement_percentage;
    double time_elapsed;
};
std::vector<SearchStats> search_history;
```

This comprehensive tracking mechanism captures the search trajectory over time, enabling sophisticated analysis of algorithm behavior and performance. The statistics help identify patterns in solution improvement and guide parameter adjustments for future runs.

2.2.4. Search Strategies

The search process combines multiple strategies to effectively navigate the solution space. We first start with the *intensification*:

```
void intensifySearch(const TSP& tsp, TSPSolution& current_sol) {
    TSPSolution backup = current_sol;
    int original_tenure = tabu_tenure;
    tabu_tenure = min_tenure;  // Reduce restrictions

    for (int i = 0; i < INTENSIFICATION_ITERATIONS; i++) {
        Move move = findBestNeighbor(tsp, current_sol, i);
        if (move.cost_change >= tsp.infinite) break;

        current_sol = applyMove(current_sol, move);
        updateMoveFrequency(move, backup_value - evaluate(current_sol, tsp));
    }

    tabu_tenure = original_tenure;  // Restore original settings
}
```

During intensification phases, the algorithm temporarily reduces the tabu tenure to allow more thorough exploration of promising regions. This approach proves particularly effective when the search has identified a high-quality solution area that merits deeper investigation.

We then have the *diversification*:

```
void diversifySearch(TSPSolution& current_sol) {
    std::vector<std::pair<int, int>> least_used_moves;
    // Identify moves used less than 25% of average frequency
    for (size_t i = 0; i < frequency_matrix.size(); i++) {
        for (size_t j = i + 1; j < frequency_matrix.size(); j++) {
            if (frequency_matrix[i][j] < frequency_matrix.size() / 4) {
                least_used_moves.push_back({i, j});
            }
        }
    }

    // Apply sequence of underutilized moves
    applyDiversificationMoves(current_sol, least_used_moves);
}
```

The diversification strategy uses the frequency matrix to identify underused moves, thereby forcing the search into new areas. This mechanism is highly important to escape from local optima and to maintain the effectiveness of long runs.

2.2.5. Implementation Details

The implementation incorporates several sophisticated mechanisms to enhance search effectiveness. First, the adaptive tabu tenure:

```
void adjustTabuTenure(double current_value) {
    if (current_value >= best_known_value) {
        iterations_without_improvement++;
        if (iterations_without_improvement > MAX_ITERATIONS_WITHOUT_IMPROVEMENT / 2)
{
            tabu_tenure = std::min(tabu_tenure + 2, max_tenure);
        }
    } else {
        tabu_tenure = std::max(tabu_tenure - 1, min_tenure);
        iterations_without_improvement = 0;
        best_known_value = current_value;
    }
}
```

The dynamic tabu tenure varies with search progress: While rare improvements evidence potential entrapment within a local optimum, tenure is increased by forcing broader explorations; when improvements are frequent, tenure decreases, allowing more intensive local searches.

The aspiration criterion provides an essential escape mechanism, allowing the algorithm to override tabu restrictions when a particularly promising move is identified:

```
bool shouldAcceptMove(const Move& move, double current_value,
                     double best_value) {
    // Accept if move leads to new best solution regardless of tabu status
    if (current_value + move.cost_change < best_value - IMPROVEMENT_THRESHOLD) {
        return true;
    }

    // Otherwise, apply standard tabu rules
    return !isTabu(move.from, move.to, current_iteration);
}
```

The termination logic combines multiple criteria to determine when to stop the search. Rather than relying solely on a fixed iteration count, it considers the search's progress and current phase, guiding parameter adjustments during the search and identifying when intensification or diversification is needed and supporting termination decisions where needed.

For this reason, the move management system maintains sophisticated tracking of move effectiveness, combining immediate evaluation with historical performance data. This hybrid approach helps the algorithm learn from past successes while remaining responsive to the current search context.

```
struct MoveManager {
    std::priority_queue<Move> candidate_moves;
    std::vector<Move> historical_best_moves;

    void updateMoveHistory(const Move& move, double improvement) {
        if (improvement > SIGNIFICANT_IMPROVEMENT_THRESHOLD) {
            historical_best_moves.push_back(move);
            if (historical_best_moves.size() > MAX_HISTORY_SIZE) {
                historical_best_moves.pop_front();
            }
        }
    }

    Move selectMove(const std::vector<Move>& current_candidates) {
        // Combine historical knowledge with current candidates
        return rankAndSelectBestMove(current_candidates,
                                     historical_best_moves);
    }
};
```

2.3. Parameter Calibration

The effectiveness of Tabu Search heavily depends on proper parameter tuning. Our implementation employs a systematic approach to parameter calibration, ensuring robust performance across different problem sizes.

2.3.1. Calibration Methodology

The *ParameterCalibration* class orchestrates this process, testing various combinations of tabu tenure values and iteration multipliers:

```
struct Parameters {
    int small_tenure;     // For instances n ≤ 20
    int medium_tenure;    // For instances 20 < n ≤ 35
    int large_tenure;     // For instances n > 35
    int small_iterations;
    int medium_iterations;
    int large_iterations;
};

CalibrationResult testParameterCombination(
    const std::vector<TSP>& instances,
    int tenure,
    int iteration_multiplier) {

    std::vector<double> qualities;
    double total_time_ms = 0.0;

    for (const auto& instance : instances) {
        TSPSolver solver;
        solver.setTabuTenure(tenure);
        solver.setMaxIterations(instance.n * iteration_multiplier);

        // Test combination and collect metrics
        auto start = std::chrono::high_resolution_clock::now();
        if (solver.solveWithTabuSearch(...)) {
            qualities.push_back(quality);
            total_time_ms += duration;
        }
    }
    // Calculate statistical measures
    return CalibrationResult{tenure, iteration_multiplier,
                            avg_quality, avg_time, std_dev};
}
```

For each combination, multiple metrics are collected including solution quality, execution time, convergence behavior and solution stability. Tabu tenure values tested are $\{5, 7, 9, 11, 13\}$, while iteration multipliers are $\{10, 15, 20, 25, 30\}$.

2.3.2. Parameter Analysis

2.3.3. Final Parameter Settings

2.4. Computational Results

2.4.1. Solution Quality Analysis

2.4.2. Performance Metrics

2.4.3. Convergence Analysis

3. Comparative Analysis

3.1. Solution Quality Comparison

3.2. Runtime Performance Analysis

3.3. Scalability Assessment

3.4. Memory Usage Analysis

4. Conclusions

4.1. Implementation Insights

4.2. Recommendations

4.3. Future Improvements