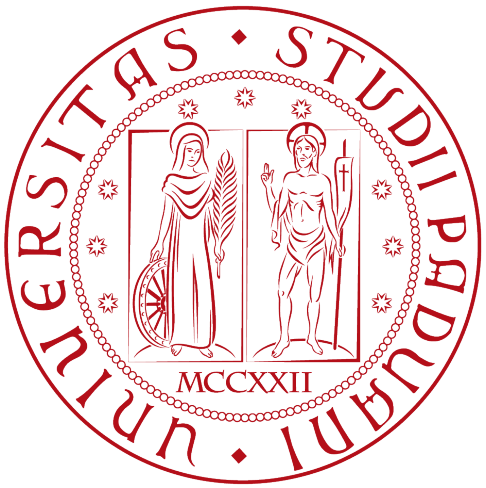

Comparison between CPLEX and Metaheuristic for solving Symmetric TSP Drilling Problem

Methods and Models for Combinatorial Optimization

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

A.Y. 2024/2025

Table of contents

1. Part I: Exact Method Implementation 3
1.1. Problem Description and Mathematical Formulation 3
1.2. Implementation with CPLEX 4
1.2.1. Core Model Implementation 4
1.2.2. Instance Generation Framework 5
1.2.3. Component Pattern Generation 7
1.2.4. Model Generation and Data Structures 7
1.2.5. Instance Generation and Management 7
1.3. Computational Results 7
1.3.1. Test Instance Generation 7
1.3.2. Performance Analysis 7
1.3.3. Solution Quality and Optimality 7
2. Part II: Tabu Search Implementation 7
2.1. Algorithm Design 7
2.1.1. Solution Representation 7
2.1.2. Neighborhood Structure 7
2.1.3. Memory Structures 7
2.1.4. Search Strategies 7
2.2. Implementation Details 7
2.2.1. Class Architecture 7
2.2.2. Tabu List Management 7
2.2.3. Intensification and Diversification 7
2.2.4. Visualization Components 7
2.3. Parameter Calibration 7
2.3.1. Calibration Methodology 7
2.3.2. Parameter Analysis 7
2.3.3. Final Parameter Settings 7
2.4. Computational Results 7
2.4.1. Solution Quality Analysis 7
2.4.2. Performance Metrics 7
2.4.3. Convergence Analysis 7
3. Comparative Analysis 7
3.1. Solution Quality Comparison 7
3.2. Runtime Performance Analysis 7
3.3. Scalability Assessment 7
3.4. Memory Usage Analysis 8
4. Conclusions 8
4.1. Implementation Insights 8
4.2. Recommendations 8
4.3. Future Improvements 8

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

The core of the implementation revolves around the flow-based TSP formulation and its representation using CPLEX's callable library. The model architecture was designed to efficiently handle manufacturing-specific constraints while maintaining the performance characteristics required for industrial circuit board drilling applications. Everything is based on a standardized interface provided by an helping interface in *cpxmacro.h*, implementing error checking and proper exception handling in the implementation choices being made.

1.2.1. Core Model Implementation

The *TSPModel* class serves as the primary interface with CPLEX and encapsulates the complete implementation of the mathematical model. This class contains two critical data structures:

```
private:
    std::vector<std::vector<int>> map_x;  // Flow variables x[i][j], j≠0
    std::vector<std::vector<int>> map_y;  // Path variables y[i][j]
```

This mapping structure was chosen over alternatives like std::map for its O(1) access time complexity, which is crucial during both model construction and solution extraction phases. The resolution process begins with model creation, where variables and constraints are systematically generated. The *createModel* method orchestrates this process:

```
void TSPModel::createModel(CEnv env, Prob lp, int N,
                           const std::vector<std::vector<double>>& costs) {
    setupVariables(env, lp, N, costs);
    setupConstraints(env, lp, N);
}
```

Variables are created in two phases: first, the continuous flow variables x[i][j] that track the amount of flow between nodes, and second, the binary path variables y[i][j] that determine the selected arcs; this way, constraints are handled efficiently.

The constraint generation process is methodically divided into three components to maintain numerical stability:
- Flow conservation constraints ensure proper network flow balance
- Assignment constraints guarantee exactly one incoming and outgoing arc per node
- Linking constraints establish the relationship between flow and path variables

The actual optimization is performed through the solve method, which executes several critical steps:

```
bool TSPModel::solve(CEnv env, Prob lp, double& objval, std::vector<int>& tour) {
    // Execute optimization
```

```
    CHECKED_CPX_CALL(CPXmipopt, env, lp);

    // Extract objective value
    CHECKED_CPX_CALL(CPXgetobjval, env, lp, &objval);

    // Retrieve solution values
    int n = CPXgetnumcols(env, lp);
    std::vector<double> x(n);
    CHECKED_CPX_CALL(CPXgetx, env, lp, &x[0], 0, n - 1);
```

The solution extraction process carefully constructs the drilling sequence from the optimized variable values. Starting from node 0 (depot), the method traces the path through the network by examining the binary path variables, ensuring a complete and valid tour is constructed.

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.

```
static constexpr double MIN_HOLE_SPACING = 2.54;    // mm (0.1 inch standard)
static constexpr double MOUNTING_HOLE_SIZE = 3.2;   // mm
static constexpr double EDGE_MARGIN = 5.0;          // mm
```

Such values are to be found within existing standards to accommodate different printing boards size and margins, in order to create a coherent standard implementation; the choice was made to show practical manufacturing requirements to ensure reliable board production. The validation system

enforces these constraints systematically, preventing the generation of instances that would be problematic in real manufacturing scenarios.

1.2.3. Component Pattern Generation

1.2.4. Model Generation and Data Structures

1.2.5. Instance Generation and Management

1.3. Computational Results

1.3.1. Test Instance Generation

1.3.2. Performance Analysis

1.3.3. Solution Quality and Optimality

2. Part II: Tabu Search Implementation

2.1. Algorithm Design

2.1.1. Solution Representation

2.1.2. Neighborhood Structure

2.1.3. Memory Structures

2.1.4. Search Strategies

2.2. Implementation Details

2.2.1. Class Architecture

2.2.2. Tabu List Management

2.2.3. Intensification and Diversification

2.2.4. Visualization Components

2.3. Parameter Calibration

2.3.1. Calibration Methodology

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