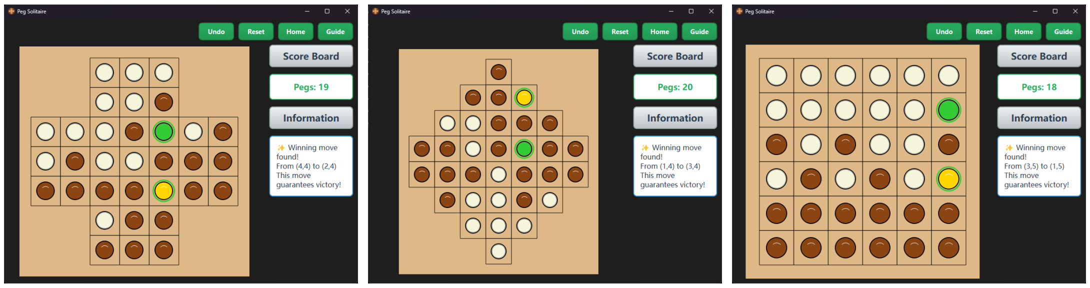
The Peg Solitaire Game Report

Name: Chenrui Wang

Student ID: 2021111664

Class: Statistics Experimental Class

June 13, 2025



✨ Winning move found!
From (4,4) to (2,4)
This move guarantees victory!

✨ Winning move found!
From (1,4) to (3,4)
This move guarantees victory!

✨ Winning move found!
From (3,5) to (1,5)
This move guarantees victory!

Figure 1: Different Peg Solitaire Board Layouts: English, Diamond, and Square

1 Introduction

In this project, I made a simple peg solitaire game using C++ with the Qt6 framework.

1.1 Main Objectives

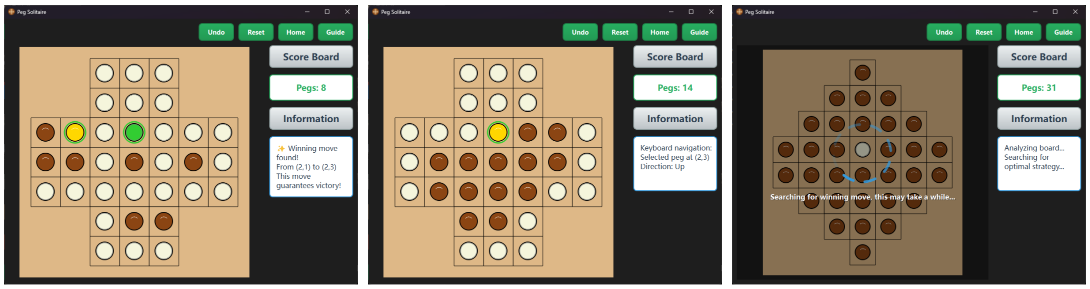
The primary aim of this project was to develop a fully functional and user-friendly Peg Solitaire game that provides an engaging puzzle-solving experience. The main objectives included:

- Implementing the classic Peg Solitaire game mechanics with proper move validation and game rules
- Creating a clean, intuitive user interface that facilitates easy interaction
- Applying the Model-View-Controller (MVC) architecture to ensure maintainable and extensible code
- Providing different board layouts and game modes to enhance gameplay variety
- Implementing intelligent strategy features to assist players in finding solutions
- Ensuring cross-platform compatibility through the Qt6 framework

1.2 Core Features

The game includes a comprehensive set of core features that form the foundation of the gameplay experience:

- **Multiple Board Layouts**: The game offers several classic board layouts as is shown in Figure 1, including:
  - English (traditional 33-hole cross-shaped board)
  - Diamond (diamond-shaped board)



Pegs: 8

✨ Winning move found!
From (2,1) to (2,3)
This move guarantees victory!

Pegs: 14

Keyboard navigation:
Selected peg at (2,3)
Direction: Up

Pegs: 31

Searching for winning move, this may take a while...

Analyzing board...
Searching for optimal strategy...

Figure 2: Bonus Features: Anti-Peg Mode, Endgame Mode, and Strategy Assistant

  - Square (square grid-based board)

- **Intuitive Controls**: Players can interact with the game using:
  - Mouse input for selecting and moving pegs
  - Keyboard controls (WASD for peg selection, arrow keys for movement)
- **Game Mechanics**: Fully implemented game mechanics including:
  - Valid move detection and highlighting
  - Win and lose condition detection
  - Undo functionality for taking back moves
  - Reset option to restart the current board
- **User Interface**: A responsive UI with:
  - HomePage with navigation options
  - Game mode selection screen
  - Settings page for game preferences
  - In-game score tracking (remaining pegs counter)
  - Information display showing current game status
  - User guide board with instructions on how to play

1.3 Bonus Features

Beyond the core gameplay mechanics, I implemented several innovative features to enhance the player experience:

- **Special Game Modes**: Figure 2 illustrates two unique game modes that provide fresh challenges:

  - **Anti-Peg Mode**: An inverse version of the classic game where players start with a single peg and aim to fill the board by jumping pegs over empty spaces. The move mechanics are reversed, requiring a completely different strategy.
  - **Endgame Mode**: Randomly generated but guaranteed-solvable puzzle configurations that challenge players with mid-game scenarios. These puzzles are created by working backward from a winning position, ensuring they always have at least one solution.

- **Strategy Assistant**:
  - Players can press the Spacebar to receive move suggestions.
  - The system uses depth-first search (DFS) to determine if the current board state is winnable.
  - Visual feedback through a loading animation indicates when the strategy calculation is in progress.
  - Dead-game detection alerts players when no winning solution exists from the current position.

- **Technical Optimizations**:
  - Board symmetry recognition to optimize the solution-finding algorithm
  - Multithreaded strategy computation to maintain UI responsiveness
  - Dynamic sizing and responsive layout for different screen resolutions
  - Fullscreen mode option for an immersive gaming experience

2 The Peg Solitaire Game

Peg solitaire represents a traditional puzzle typically played on either a cross-shaped board with 33 holes (commonly referred to as ”the English board”) or a triangular board containing 15 holes.[1]

2.1 Boards

The game features multiple board layouts, each with its own unique characteristics and challenges:

2.1.1 English Board

The traditional English board is cross-shaped with 33 holes arranged in a distinctive pattern. The standard starting position has pegs in all holes except for the center, which is empty. The objective is to remove pegs by jumping over them until only one peg remains, ideally in the center position.

The English board layout is implemented in the code as follows:

```
int starLayout[rows][cols] = {
    {-1, -1,  1,  1,  1, -1, -1},
    {-1, -1,  1,  1,  1, -1, -1},
    { 1,  1,  1,  1,  1,  1,  1},
    { 1,  1,  1,  0,  1,  1,  1},
    { 1,  1,  1,  1,  1,  1,  1},
    {-1, -1,  1,  1,  1, -1, -1},
    {-1, -1,  1,  1,  1, -1, -1}
};
```

Where 1 represents a peg, 0 represents an empty hole, and -1 represents a position outside the board.

2.1.2 Diamond Board

The Diamond board is a compact, symmetrical layout with 41 holes arranged in a diamond pattern. The central hole is initially empty, and the goal remains the same as the English board—to finish with a single peg.

2.1.3 Square Board

The Square board provides a different challenge with a regular grid layout. This more geometrically uniform board offers different strategic considerations compared to the

English and Diamond boards.

2.1.4 Board Representation

In the implementation, boards are represented using a two-dimensional grid structure:

```
QVector<QVector<PegState>> grid;
```

Each cell in the grid can be in one of three states:

```
enum class PegState {
    Empty,
    Peg,
    Blocked // For cells that are not part of the playable board
};
```

2.2 Special Game Rules

In addition to the standard Peg Solitaire rules, the game includes special variants that provide novel gameplay experiences:

2.2.1 Standard Rules

In the classic Peg Solitaire game:

- A peg can jump over an adjacent peg into an empty hole.
- The peg that is jumped over is removed from the board.
- Jumps can only be made horizontally or vertically, not diagonally.
- The goal is to clear the board until only a single peg remains.
- A ”perfect” win is achieved when the final peg is in the center position.

2.2.2 Anti-Peg Mode

This innovative mode reverses the traditional gameplay mechanics:

- The game begins with only one peg in the center of the board.
- A peg jumps over an empty space and lands on another empty space.
- After the jump, a new peg is placed in the empty space that was jumped over.
- The objective is to fill the board with as many pegs as possible.

- The game ends when no more moves are possible.
- A perfect win is achieved when only the starting position remains empty.

The Anti-Peg mode required significant code adaptations, as shown in this excerpt:

```
if (isAntiPegMode()) {
    // Anti-peg mode validation: peg jumps over empty to empty, placing peg in jumped
    if (getPegState(move.from) == PegState::Peg &&
        getPegState(move.jumped) == PegState::Empty &&
        getPegState(move.to) == PegState::Empty)
    {
        // Perform anti-peg move: move peg to destination, place peg in jumped cell
        setPegState(move.from, PegState::Empty);
        setPegState(move.jumped, PegState::Peg);    // Place peg in jumped cell
        setPegState(move.to, PegState::Peg);
        return true;
    }
}
```

2.2.3 Endgame Mode

The Endgame mode presents players with partially-completed board configurations:

- The board is initialized with a random but solvable mid-game position.
- These positions are generated by working backward from a winning state.
- The algorithm applies 8-15 reverse moves from the winning position to create challenging puzzles.
- This guarantees that every puzzle has at least one solution.
- Players must analyze the position carefully to find the correct sequence of moves.

2.3 Algorithm for Solving the Game

Finding solutions to Peg Solitaire puzzles poses a significant computational challenge due to the vast number of possible move sequences. To address this, I implemented an optimized backtracking algorithm with several performance enhancements.

2.3.1 Depth-First Search with Backtracking

The core solution algorithm uses depth-first search (DFS) with backtracking:

- The algorithm recursively explores all possible move sequences from the current board state.
- At each step, it tries every valid move, makes the move, and then continues searching.
- If a dead end is reached, it backtracks and tries another move.
- The search terminates when a winning state is found (only one peg remains).

2.3.2 Symmetry Optimization

A key innovation in the algorithm is the exploitation of board symmetry to drastically reduce the search space:

- The English board has 8 symmetrical variants (4 rotations $\times$ 2 reflections).
- Many board states are equivalent under rotation and reflection.
- By identifying and storing these symmetric states, the algorithm avoids redundant exploration.
- A unique 64-bit identifier is computed for each board state and its symmetrical variants.

The symmetry detection is implemented using the following functions:

```
quint64 Board::getBoardStateId() const {
    // Get the canonical representation (smallest ID across all symmetries)
    quint64 minId = std::numeric_limits<quint64>::max();
    for (int rotation = 0; rotation < 4; rotation++) {
        for (int flip = 0; flip <= 1; flip++) {
            quint64 id = boardToBits(rotation, flip ? true : false);
            if (id < minId) {
                minId = id;
            }
        }
    }
    return minId;
}
```

```
QVector<quint64> Board::getAllSymmetricStateIds() const {
    QVector<quint64> ids;
    for (int rotation = 0; rotation < 4; rotation++) {
        for (int flip = 0; flip <= 1; flip++) {
            ids.append(boardToBits(rotation, flip ? true : false));
        }
    }
    return ids;
}
```

2.3.3 Memoization

To further enhance performance, the algorithm uses memoization to avoid recomputing results for previously seen states:

- A static cache stores board states that have already been determined to be unsolvable.
- Before starting the search for a new position, the algorithm checks if the position (or any of its symmetric variants) is in the cache.
- This pruning technique significantly reduces the search space for complex positions.

2.3.4 Multi-threaded Strategy Computation

To maintain UI responsiveness while computing solutions, strategy calculations are offloaded to a separate worker thread:

- A `StrategyWorker` class handles the computational workload asynchronously.
- The main UI thread remains responsive during intensive calculations.
- A loading animation provides visual feedback to the user during computation.
- The worker can be interrupted if the board state changes or the user requests a different action.

These algorithmic optimizations transform an intractable problem (which might take hours to solve with naive backtracking) into one that can be solved in a reasonable amount of time for most board states, making the strategy assistant feature practical for real-time use.

3 Lessons Learned

The Peg Solitaire game is a fascinating puzzle that not only provides entertainment but also serves as an excellent exercise in problem-solving and algorithm design. Through the process of developing a solution for this game, I have learned several valuable lessons that extend beyond the confines of this specific project.

3.1 Project Management

Developing the Peg Solitaire game provided valuable lessons in project management and software development practices:

3.1.1 Incremental Development

I adopted an incremental development approach, focusing on one feature at a time. The project's evolution can be traced through the incremental prompts found in the `/prompts` directory, which document the step-by-step progression:

- Starting with a basic game structure and homepage (01-init.md)
- Adding game mode selection (02-start.md)
- Implementing settings functionality (03-settings.md)
- Creating various game boards (04-gamemodes.md)
- Developing the UI elements (05-buttonUI.md, 06-gameUI.md)
- Adding core gameplay features (07-winandlose.md, 08-keyboard.md, 09-selection.md)
- Implementing special modes (10-anti.md)
- Adding advanced features like strategy suggestions (11-strategy.md)
- Enhancing the user experience (12-information.md)
- Optimizing algorithms (13-optimize.md)
- Creating the loading animation (14-loading.md)
- Developing the endgame mode (15-endgame.md)

This approach allowed for continuous progress and clear direction, ensuring that each component functioned properly before moving on to the next.

3.1.2 Modular Architecture

Implementing the Model-View-Controller (MVC) pattern was crucial for maintaining code organization and facilitating feature additions:

- **Model**: The `Board` class encapsulates the game state and rules, independent of the user interface
- **View**: Classes like `BoardView` and `GameView` handle visual representation and user input
- **Controller**: The `BoardController` class mediates between the model and view, processing user actions and updating the game state

This separation of concerns made it easier to:

- Add new features without breaking existing functionality
- Test components in isolation
- Maintain a clean and understandable codebase

3.1.3 Version Control and Documentation

The project benefited from proper version control practices and thorough documentation:

- Git was used for tracking changes and managing development
- Commit messages provided clear explanations of implementation choices
- Code comments and documentation made the codebase more accessible
- A comprehensive README file provided build instructions and feature overviews

3.2 Algorithm Design

The development of this project provided significant insights into algorithm design and optimization techniques:

3.2.1 Recursive Backtracking

Implementing the solution-finding algorithm for Peg Solitaire offered valuable lessons in recursive algorithms:

- The depth-first search approach proved effective for exploring possible move sequences
- Proper backtracking ensured all potential solutions were considered
- Base cases needed careful consideration to prevent infinite recursion
- The practical limitations of pure recursion became evident as the search space grew exponentially

3.2.2 Optimization Techniques

Facing performance challenges with the naive implementation led to learning several important optimization strategies:

- **Pattern Recognition**: Identifying board symmetries reduced the search space significantly. The recognition that rotating or flipping the board produces equivalent game states led to a major algorithmic improvement:

```
Position Board::getTransformedPosition(
    const Position& pos, int rotations, bool flip) const {
    // Get board center for rotation reference
    int centerRow = rows / 2;
    int centerCol = cols / 2;

    // Adjust position to origin-centered coordinates
    int x = pos.col - centerCol;
    int y = pos.row - centerRow;

    // Apply flip if needed
    if (flip) {
        x = -x;
    }

    // Apply rotation (0, 90, 180, or 270 degrees)
    for (int r = 0; r < rotations; r++) {
        // 90-degree rotation: (x,y) -> (-y,x)
        int temp = x;
        x = -y;
        y = temp;
    }

    // Convert back to board coordinates
    return Position(y + centerRow, x + centerCol);
}
```

- **Memoization**: Caching previously computed results avoided redundant calculations. The static cache of unsolvable board states dramatically improved performance:

```
// Static cache to optimize board solving
static QSet<quint64> failedStates;

bool BoardController::solveBoard(Board* board) {
    // Check if we've already determined this board is unsolvable
    quint64 boardId = board->getBoardStateId();
    if (failedStates.contains(boardId)) {
        return false;
    }

    // Check symmetrical variants as well
    QVector<quint64> allSymmetricIds = board->getAllSymmetricStateIds();
    for (const quint64& id : allSymmetricIds) {
        if (failedStates.contains(id)) {
            return false;
        }
    }

    // Continue with solving algorithm...
}
```

- **Pruning**: Early termination of search paths that couldn't lead to solutions reduced computation time

- **Bit Manipulation**: Using bit operations to create compact board state representations improved memory efficiency and comparison speed

3.2.3 Multithreading

Implementing the strategy calculation worker thread taught important principles of concurrent programming:

- Separating CPU-intensive tasks from the UI thread maintained application responsiveness
- Proper synchronization mechanisms (mutexes, signals/slots) prevented race conditions
- Cancellation mechanisms allowed for interrupting long-running operations when the board state changed
- Thread-safe data structures ensured consistent state across threads

3.2.4 Generative Algorithms

The Endgame mode required a novel approach to generating random but solvable puzzles:

- Working backward from the winning state guaranteed solvability
- Applying random ”reverse moves” created diverse puzzle configurations
- This reverse-engineering approach ensured the generated puzzles were both challenging and fair

3.3 Understanding of the C++ Language

This project significantly deepened my understanding of modern C++ and object-oriented programming principles:

3.3.1 Object-Oriented Design

Working with C++ reinforced key OOP concepts:

- **Encapsulation**: Properly hiding implementation details behind public interfaces improved maintainability. For example, the `Board` class encapsulates all board state management:

```
class Board : public QObject {
private:
    QVector<QVector<PegState>> grid;
    int rows;
    int cols;
    int pegCount;
    BoardType currentBoardType;
    QVector<Move> moveHistory;

public:
    // Public interface methods
    QVector<Move> getValidMoves() const;
    bool performMove(const Move &move);
    bool undoLastMove();
    PegState getPegState(Position pos) const;
    // ...
};
```

- **Inheritance**: Extending Qt classes allowed for customizing behavior while leveraging existing functionality
- **Polymorphism**: Using virtual functions and overriding methods like `paintEvent` in UI components
- **Composition**: Building complex objects from simpler ones, such as composing the game view from multiple UI elements

3.3.2 Modern C++ Features

The project provided opportunities to utilize modern C++ features:

- **Strong Typing**: Using enum classes for type-safe enumerations (`BoardType`, `PegState`)
- **Auto Type Deduction**: Simplifying code with automatic type inference where appropriate
- **Smart Pointers**: Managing object lifetimes, although Qt's parent-child relationship system was primarily used

- **Lambda Expressions**: Implementing concise callbacks for event handling and signal connections
- **Range-based Loops**: Writing cleaner iteration code for collections

3.3.3 Qt Framework Integration

Working with the Qt framework enhanced my understanding of:

- **Signal-Slot Mechanism**: Implementing loosely coupled communication between components:

```
// In constructor
connect(boardView, &BoardView::pegClicked,
        this, &BoardController::onPegCellClicked);
connect(this, &BoardController::highlightMovesSignal,
        boardView, &BoardView::highlightMoves);
connect(this, &BoardController::pegsRemainingChanged,
        boardView, &BoardView::updatePegCount);
```

- **Meta-Object System**: Leveraging Qt's reflection capabilities with the Q_OBJECT macro
- **Event Handling**: Overriding event methods to handle mouse, keyboard, and paint events
- **Custom Painting**: Implementing custom drawing routines with QPainter for the board visualization
- **Layouts**: Creating responsive UIs that adapt to window resizing
- **Widget Management**: Creating, organizing, and managing hierarchical UI components

3.3.4 Memory Management

The project highlighted important aspects of memory management in C++:

- Understanding object ownership models in Qt's parent-child relationship system
- Properly cleaning up resources when objects are no longer needed

- Avoiding memory leaks by ensuring proper object destruction
- Managing dynamically allocated memory for temporary objects (like test boards in the solving algorithm)

3.3.5 Performance Considerations

Writing performance-critical code provided insights into C++ optimization:

- Using pass-by-reference for large objects to avoid expensive copying
- Employing const correctness to communicate intent and enable compiler optimizations
- Understanding when to use different container types based on access patterns
- Minimizing object creation and destruction in tight loops
- Profiling and identifying bottlenecks in the algorithm implementation

Overall, this project served as an excellent practical application of C++ programming concepts, reinforcing theoretical knowledge with hands-on experience in a complex, real-world application.

References

[1] George I. Bell. George's peg solitaire page. Online, 2025.