CSE472s: Artificial Intelligence Course

Quoridor Game Implementation

Submitted By:

Name	ID
Ahmed Haitham Ismail El-Ebidy	2101629
Mohamed Khaled Elsayed Goda	2100675
Marwan Ahmed Hassan Ali	2100902

December 17, 2025

Contents

1 Introduction

1.1 Project Overview

The objective of this project is to design and implement a fully functional version of the board game **Quoridor**, complete with a Graphical User Interface (GUI) and an Artificial Intelligence (AI) opponent. The project serves as a practical application of concepts covered in the CSE472s Artificial Intelligence course, specifically focusing on adversarial search, heuristic evaluation, state-space representation, and software design patterns.

The resulting software allows for Human vs. Human gameplay on a local machine as well as Human vs. Computer gameplay with varying difficulty levels. It features robust game mechanics including path validation, save/load functionality, and an undo system.

1.2 Game Description

Quoridor is an abstract strategy game designed by Mirko Marchesi and published by Gigamic. It is played on a 9×9 grid. Each player begins with a pawn in the center of their baseline (row 0 for Player 1, row 8 for Player 2).

1.2.1 Objective

The goal is to be the first player to move their pawn to any square on the opposite side of the board. For Player 1, this means reaching row 8; for Player 2, reaching row 0.

1.2.2 Components

- **Board:** A 9x9 grid of cells separated by grooves where walls can be placed.
- **Pawns:** Tokens representing the players' current positions.
- **Walls:** Flat pieces that block movement. Each player starts with 10 walls in a 2-player game. Walls span two squares in length.

1.3 Assumptions Made During Development

To scope the project within the term limits and define the AI's behavior, the following assumptions were made:

- **Rational Opponent:** The Minimax algorithm operates under the assumption that the opponent is playing optimally (Zero-Sum Game). The AI calculates its moves expecting the human player to always choose the move that minimizes the AI's advantage.
- **Heuristic Correlation:** We assume that the "Manhattan Distance" to the goal is a sufficiently accurate heuristic for estimating the "true cost" to win, even though walls may force longer paths. We assume that calculating the exact shortest path for every possible future state is computationally infeasible.
- **Deterministic Environment:** The game environment is assumed to be fully deterministic and static. The board state only changes based on the agents' valid moves, with no stochastic elements (luck) or external interference.

2 Game Rules and Mechanics

Correct implementation of the rules is critical for the validity of the AI search state.

2.1 Movement

On a turn, a player may choose to move their pawn or place a wall.

1. **Pawn Movement:** A pawn moves one square horizontally or vertically. Diagonal movement is not allowed unless jumping.
2. **Jumping:** If a pawn is adjacent to an opponent's pawn, it may jump over the opponent to the square immediately behind them.
3. **Diagonal Jump:** If a wall or the edge of the board blocks a straight jump, the pawn may move diagonally to a square adjacent to the opponent.

2.2 Wall Placement

Walls are placed in the grooves between cells.

1. **Alignment:** Walls can be placed horizontally or vertically.
2. **Overlap:** Walls cannot overlap with previously placed walls or intersect them.
3. **Path Guarantee:** A wall cannot be placed if it completely cuts off the only remaining path to the goal for either player. A valid path must always exist.

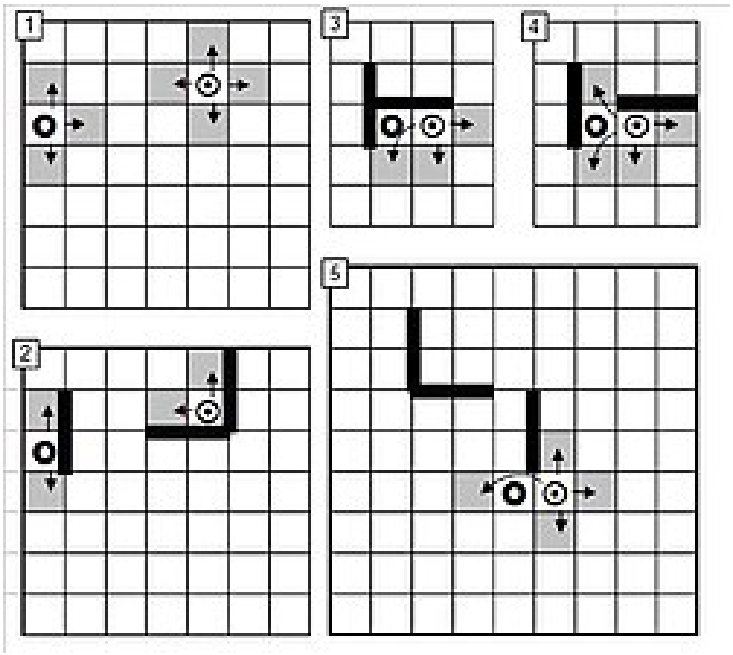

Figure 1: Diagram illustrating valid pawn moves and wall placement constraints.

3 System Architecture

The software is constructed using a modular architecture to separate the game logic, the rendering engine, and the decision-making AI. This ensures maintainability and allows for easier debugging.

3.1 Class Structure

3.1.1 1. QuoridorGame (logic.py)

This class acts as the "Model" in the MVC pattern. It encapsulates the complete state of the game.

- **State Variables:**
  - `p1_pos`, `p2_pos`: Tuples representing (x, y) coordinates of pawns.
  - `walls`: A list of placed walls, stored as $((x, y), \text{orientation})$.
  - `turn`: Integer (1 or 2) indicating the current player.
  - `winner`: Stores the ID of the winner if the game has ended.
- **Key Methods:**
  - `get_valid_pawn_moves(player_id)`: Returns a list of legal coordinates.
  - `is_valid_wall(x, y, orientation)`: Checks boundaries, overlaps, and importantly, calls the pathfinder to verify connectivity.
  - `apply_move(move)`: Updates the state based on an input dictionary.

3.1.2 2. QuoridorGUI (gui.py)

This class handles the "View" and "Controller" aspects using the Pygame library.

- **Rendering:** Responsible for drawing the grid, pawns, walls, and UI buttons.
- **Input Handling:** Captures mouse clicks for movement and wall placement, and keyboard events for shortcuts.
- **Asset Management:** Loads images and audio files dynamically from the `assets/` directory.
- **Responsiveness:** It includes logic to re-calculate layout coordinates when the window is resized, switching between "Bottom Layout" (portrait) and "Side Layout" (landscape).

3.1.3 3. AI (ai.py)

This class contains the intelligence of the computer opponent.

- **Configuration:** Accepts a difficulty parameter to adjust search depth.
- **Search Engine:** Implements the Minimax algorithm with Alpha-Beta pruning.
- **Evaluation:** Contains the heuristic function to score non-terminal board states.

3.1.4 4. Config (config.py)

A static configuration file holding constants for:

- **Theme Colors:** Definitions for "Classic Wood" and UI elements.
- **Game Settings:** AI depths, winning scores, and memory limits.
- **Layout Dimensions:** Dynamic variables for screen size calculations.

4 Algorithms and Logic

4.1 Pathfinding (Connectivity Check)

One of the most computationally expensive rules in Quoridor is ensuring a wall does not block the path to the goal. We utilized the **A* Search Algorithm** due to its efficiency.

4.1.1 Heuristic for A*

We used the Manhattan Distance as the heuristic $h(n)$:

$$h(n) = |n.y - goal\_row|$$

Since pawns cannot move diagonally (without jumping), Manhattan distance provides an admissible and consistent heuristic.

Algorithm 1 Shortest Path Check (A*)

```
function ShortestPath(start_pos, goal_row, walls)
    open_set ← PriorityQueue()
    open_set.add((0, start_pos))
    g_score[start_pos] ← 0
    while open_set is not empty do
        current ← open_set.pop()
        if current.y == goal_row then return ReconstructPath(current)
        end if
        for neighbor in GetNeighbors(current, walls) do
            tentative_g ← g_score[current] + 1
            if tentative_g < g_score[neighbor] then
                g_score[neighbor] ← tentative_g
                f_score ← tentative_g + |neighbor.y − goal_row|
                open_set.add((f_score, neighbor))
            end if
        end for
    end while
    return ∅                                              ▷ No path exists
end function
```

4.2 Adversarial Search (Minimax)

The AI decision-making is powered by the Minimax algorithm. Given that Quoridor is a zero-sum game (one player's gain is the other's loss), Minimax is the ideal choice. However, the branching factor of Quoridor is high. On an empty board:

- Pawn moves: ≈ 4
- Wall placements: 8×8 intersections $\times$ 2 orientations $= 128$

A naive Minimax search would be too slow. We implemented **Alpha-Beta Pruning** to eliminate branches that do not affect the final decision.

4.2.1 Evaluation Function

Since the search cannot reach the end of the game (terminal state) within a reasonable time, we use a heuristic evaluation function at the leaf nodes. The score is calculated from the perspective of the AI (maximizing player).

$$\text{Score} = (20 - \text{Dist}_{AI}) \times 10 - (20 - \text{Dist}_{Opp}) \times 10 + (\text{Walls}_{AI} - \text{Walls}_{Opp}) \times 5$$

Rationale:

1. **Distance (Weight 10):** We want to minimize our distance to the goal and maximize the opponent's distance. This is the primary driver of behavior.
2. **Walls (Weight 5):** Having more walls than the opponent is a strategic advantage, allowing for future blocking maneuvers.

4.3 Optimizations

To enable deeper searches (Depth 3-4), several optimizations were required.

4.3.1 1. Transposition Tables

We cache the evaluation of board states. Quoridor has many transpositions (e.g., moving Pawn A then Pawn B results in the same state as moving Pawn B then Pawn A). We hash the board state (pawn positions + sorted wall list) to store and retrieve values, preventing redundant calculations.

4.3.2 2. Move Ordering

The efficiency of Alpha-Beta pruning depends heavily on the order in which moves are visited. We sort moves such that:

- Pawn moves are evaluated first (low branching factor, immediate distance change).
- Wall moves are evaluated second.

This allows the algorithm to establish a decent "alpha" value quickly using pawn moves, potentially pruning the vast number of wall moves that follow.

4.3.3 3. Selective Wall Generation

Instead of generating all 128 possible wall moves, the AI only considers "relevant" walls. A relevant wall is defined as one that is placed within a radius of 1 cell from the opponent's current shortest path. This heuristic drastically reduces the branching factor from ≈ 130 to ≈ 20, making the search significantly faster while still playing strategically.

5 Implementation Details and Challenges

5.1 The "Pickle Recursion" Memory Leak

One of the most significant challenges encountered was a memory leak during the implementation of the Undo system.

The Problem: We utilized Python's `pickle` module to serialize the `QuoridorGame` object into a history stack.

```
self.history.append(pickle.dumps(self))
```

However, the `self` object contained the history list itself. By pickling `self`, we were pickling the current state *plus* the entire history of previous states.

- Turn 1 saves State 1.
- Turn 2 saves State 2 (which contains a copy of State 1).
- Turn 3 saves State 3 (which contains copies of State 2 and 1).

This resulted in $O(N^2)$ memory growth. By turn 25, the game would consume gigabytes of RAM and crash.

The Solution: We implemented a custom `save_state` method that creates a shallow copy of the object's dictionary and explicitly removes the history attribute before serialization. This ensures constant memory usage per snapshot.

```
def save_state(self):
    clean_state = self.__dict__.copy()
    if 'history' in clean_state:
        del clean_state['history'] # Break recursion
    return pickle.dumps(clean_state)
```

5.2 In-Place State Modification

Standard Minimax implementations often clone the game board for every simulated move. In Python, `deepcopy` is slow. To optimize the AI, we implemented `apply_move_fast` and `undo_move_fast`.

- **Apply:** Modifies the board variables directly and returns a lightweight "delta" object containing only the data needed to reverse the move (e.g., previous position).
- **Undo:** Uses the delta object to restore the variables.

This approach eliminated the overhead of object creation during the search phase.

6 User Interface Design

The Graphical User Interface (GUI) was built using Pygame. Special attention was paid to usability and visual feedback.

6.1 Visual Theme

We moved away from the default abstract colors to a "Classic Wood" theme to emulate the physical board game feel.

- **Board:** Medium Oak (#8B6B4E)
- **Cells:** Light Beige (#C2B280)
- **Players:** Mahogany Red and Forest Green for high contrast.

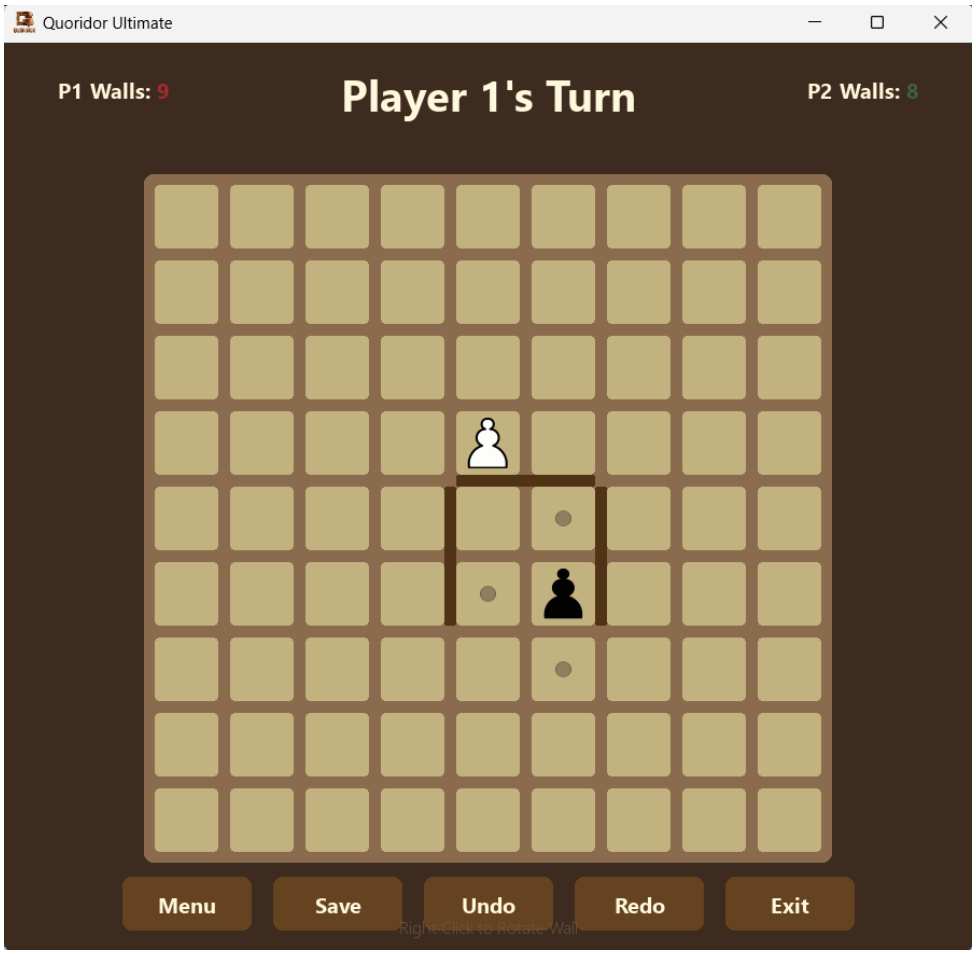

Figure 2: Gameplay screenshot showing the "Classic Wood" theme, pawns, and walls.

6.2 Dynamic Layout

The GUI handles window resizing events (`VIDEORESIZE`). It calculates the layout dynamically, switching between "Bottom Layout" (portrait) and "Side Layout" (landscape). This ensures the board remains maximized and centered regardless of window shape.

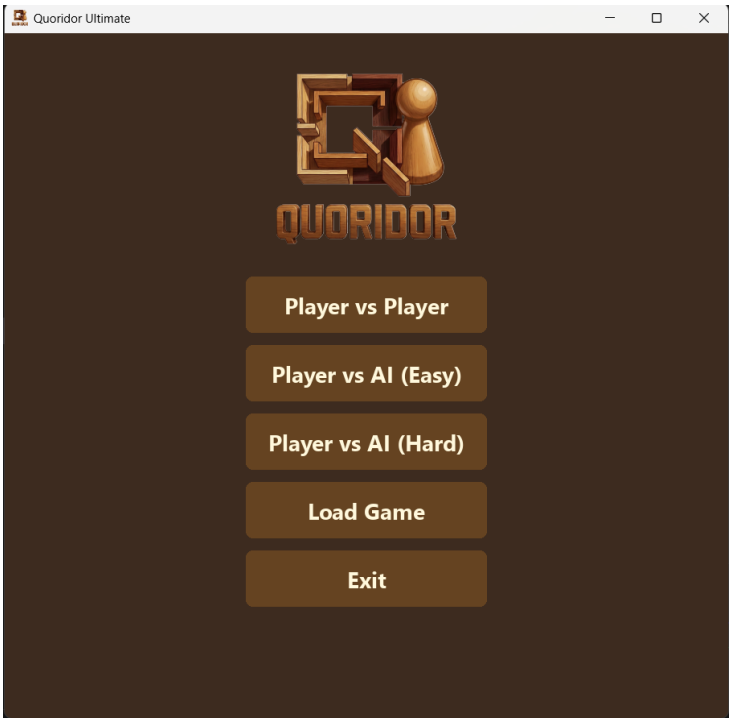

Figure 3: Main Menu showing the options and custom logo.

6.3 Win State and Rematch

When a player reaches the goal:

1. An overlay darkens the board.
2. A "Player X Wins" message is displayed.
3. A "Rematch" button appears, allowing immediate restart with the same settings.
4. A "Menu" button appears to return to the title screen.

7 Testing and Results

7.1 Functional Testing

We verified the implementation against the official Quoridor rules:

- **Path Blocking:** We confirmed that the game forbids placing a wall that blocks the last path.
- **Jump Logic:** We tested edge cases, such as jumping over an opponent when a wall is behind them (forcing a diagonal move).
- **Save/Load:** Verified that games can be saved to disk and loaded later without data loss.

7.2 AI Performance

We benchmarked the AI performance on a standard consumer laptop (Intel Core i7).

Difficulty	**Depth**	**Avg Time/Move**	**Win Rate vs Random**
Easy	2	$< 0.1s$	100%
Hard	4	$1.5s - 3.0s$	100%

Table 1: AI Performance Metrics

The "Hard" AI (Depth 4) plays a competent game. It effectively blocks the human player when they get close to the goal and navigates complex mazes created by walls. The selective wall generation optimization was crucial in keeping the processing time under 3 seconds per turn.

8 Conclusion and Future Work

This project successfully delivered a complete, polished implementation of Quoridor. The application is robust, visually appealing, and provides a challenging AI opponent. The modular design allowed us to overcome significant hurdles like memory leaks and search inefficiencies.

8.1 Future Improvements

- **4-Player Mode:** Extending the logic to support 4 players.
- **Monte Carlo Tree Search (MCTS):** Replacing Minimax with MCTS to potentially improve strategic depth in the late game.
- **Online Multiplayer:** Using Python sockets to allow play over a network.

References

[1] Gigamic. "Quoridor Official Rules." Gigamic Games. Accessed 2025.

[2] Russell, Stuart, and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed., Pearson, 2020 (Chapters on Adversarial Search and Heuristics).

[3] Hart, P. E., Nilsson, N. J., and Raphael, B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, 1968.

[4] Pygame Community. "Pygame Documentation." https://www.pygame.org/docs/.

[5] Python Software Foundation. "pickle — Python object serialization." https://docs.python.org/3/library/pickle.html.

Project Resources

Below are the links to the video demo and GitHub repository:

- **Video Demo:** Click here to watch the demo
- **GitHub Repository:** Click here to visit the repository