School of Computer Science

FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

# Final Report

## Computerised Quoridor

**Matthew Bowler**

**Submitted in accordance with the requirements for the degree of**

BSc Computer Science

**2024/25**

**COMP3931 Individual Project**

The candidate confirms that the following have been submitted*:*

| **Items** | **Format** | **Recipient(s) and Date** |
|---|---|---|
| *Final Report* | *PDF file* | *Uploaded to Minerva (26/04/25)* |
| *Link to online code repository* | *URL* | *Sent to supervisor and assessor (26/04/25)* |

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student) [illegible]

## Summary

This project focuses on the development of a software implementation for the 2-player version of the board game Quoridor, that includes an AI playing agent. Extensive background research is performed into relevant topics regarding the concepts required for the game implementation and game playing AI. Existing literature and implementations of Quoridor playing agents are reviewed and criticised. A game implementation is delivered with a graphical user interface that allows for human versus human, human versus AI and AI versus AI games. A minimax agent using a search depth of 2 is developed. The agent's strength is optimised using a simplistic genetic algorithm. User playtesting is used to collect qualitative and quantitative data on the implementation of the game and AI. AI playtesting is used to collect quantitative data on the strength of the AI agent. Results of these playtests are investigated and discussed. Finally, an overview of the project is presented and ideas for future developments and research are discussed.

# Acknowledgements

I would like to thank my supervisor Dr. Haiko Muller for proposing this project, along with his support and guidance throughout the projects development all of which shaped the project into the success it was. I would also like to thank the participants of the user testing for their time and insightful criticisms of the final software product.

# Table of Contents

## Contents

# Chapter 1
# Introduction and Background Research

## 1.1 Introduction

Board games have long been a source of entertainment and intellectual challenge dating back to 3500 BC [1]. Traditionally board games have existed in a physical format however, in recent years, the digital transformation of board games has become increasingly popular. This rise in digitalised implementations has allowed board games to become more accessible than ever. Additionally, recent developments in both mathematics and computer science have allowed for deeper analysis on board game strategy and the development of artificial intelligence (AI) playing agents. Such AI agents can enhance user experience by providing a challenging opponent and the opportunity to learn and play without another human present.

This project focuses on the design and development of a digital implementation of the board game Quoridor, with an integrated AI opponent. The development process followed an iterative approach, beginning with just the core game implementation before evolving into AI integration and refinement.

Quoridor is a strategy game developed in 1975 by Phillip Slater and is based on an earlier game named Blockade [2]. The objective of the game is for the players to navigate their pawn across a board while placing walls to obstruct their opponent’s progress. The games relatively simple rules make it easy for beginners to understand however, the deep strategic nature of the game makes it difficult to master. This complexity makes it an interesting subject for both strategic and computational analysis.

While this dissertation investigates implementing a challenging Quoridor playing agent, its focus is delivering a performant and user-friendly software implementation of the game Quoridor.

The following sections will detail the background research into the game of Quoridor, the requirements of developing a computerised Quoridor game, board game AI and existing literature and implementations of Quoridor playing agents.

## 1.2 Quoridor

Quoridor is a 2 or 4 player game typically played on a 9x9 chess-like board. The game consists of 2 major components: pawns and walls. Each player controls a single pawn, and their objective is to move it to the other side of the board to win. Each player also has 10 walls, which can be placed between 2 adjacent squares to obstruct opponent's movement. This project and subsequent sections focus entirely on the 2-player version of the game.

### 1.2.1 Notation

The notation I will be using throughout this report is a variation of the algebraic standard notation used for chess [3].

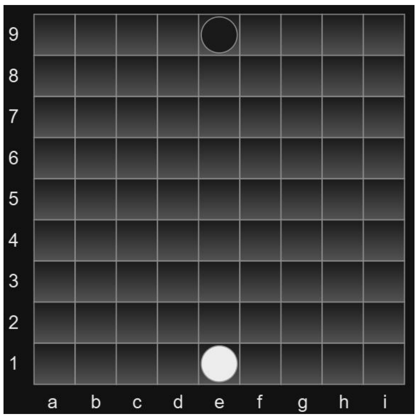


**Figure 1 - Initial Quoridor board**

Square positions are denoted in a coordinate-like fashion, with the combination of both row and column. Pawn positions will simply be denoted by which square they currently occupy; in Figure 1 the white pawn's position is e1. Wall positions will be denoted by the square position it begins at and either "h" or "v" for horizontal or vertical orientation respectively. Walls will always begin in the bottom left corner of the square they occupy and extend 2 square lengths in their respective orientations.

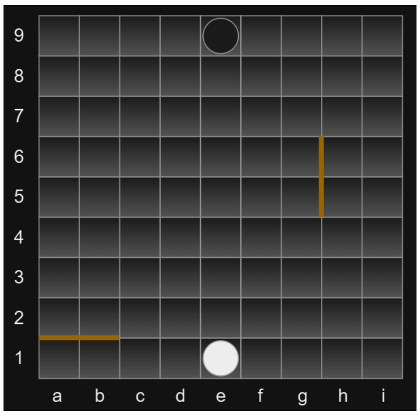


**Figure 2 - Walls placed at a2h and h5v**

### 1.2.2 Setup

Initially, the board is empty with no walls placed between squares. The white pawn is placed halfway between the bottom row and the black pawn is placed halfway between the top row. In the 9x9 version of the game these positions are e1 and e9 respectively. The initial board state of a 9x9 game of Quoridor can be seen in Figure 1.

### 1.2.3 Rules

White always moves first and players take turns either moving their pawn or placing walls following the rules described below.

#### 1.2.3.1 Pawn movement

A player can move their pawn a distance of 1 square in 1 of the 4 cardinal directions (up, down, left or right) on the board, given they are not blocked by a wall.

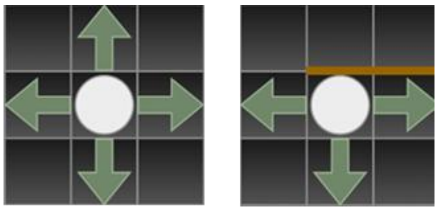


**Figure 3 - Valid pawn movement**

If 2 pawns are adjacent to each other, the player whose move it is may jump over the opposing pawn in the direction they are adjacent to.

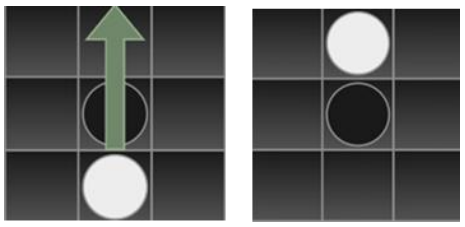


**Figure 4 - Pawn jump move**

If the jump is blocked by a wall, the pawn can instead move diagonally to the squares beside the opponent, given they aren't blocked by walls.

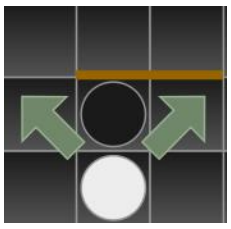


**Figure 5 - Pawn diagonal movement**

One edge case without an official rule is whether the edges of the board are considered walls and whether diagonal movement should be allowed with respect to the edges of the board (see Figure 6). In the implementation described in this report this type of move is allowed.

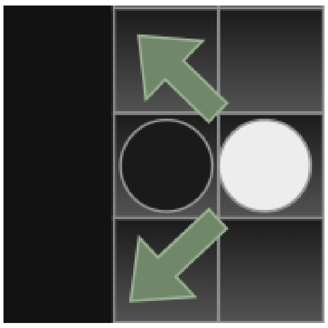


**Figure 6 - Pawn diagonal move at the edge of the board**

**1.2.3.2 Wall placement**

Walls can be placed between 2 squares and must fully cover the squares they are placed between; they cannot partially block squares. Walls cannot overlap, cross or be stacked on top of one another. Walls cannot be placed along the edges of the board (e.g. placements such as a1h or a1v are invalid).

Walls must be placed in a way such that there always exists at least 1 path from each pawn to their goal row. In Figure 7, the wall g1v would completely block the white pawn from progressing and therefore is invalid.

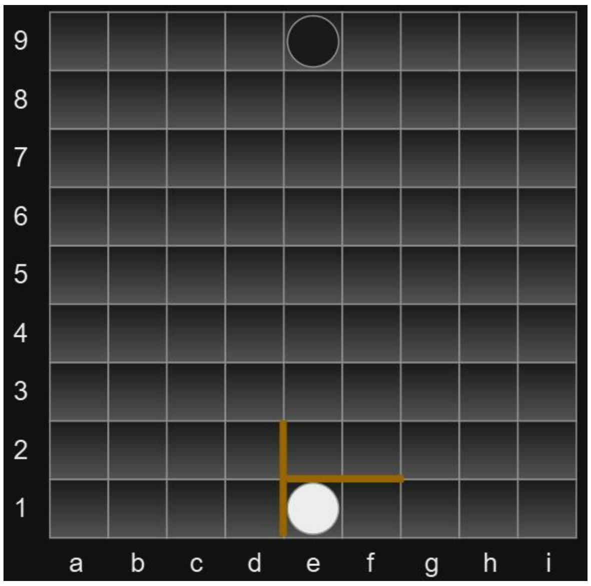


**Figure 7 - Wall placements such wall g1v would be an illegal move**

## 1.3 Pathfinding Algorithms

Now the rules governing wall placement have been established, it is essential to define how the game will determine valid paths for each player. Additionally, finding the shortest path between players and their goal is an essential feature of the evaluation function proposed in Section 3.2.3. These are classic pathfinding problems which are typically solved using search algorithms. A search algorithm explores a set of possible solutions to determine the most optimal path.

Pathfinding algorithms often rely on properties such as heuristics and cost functions. A heuristic provides an estimate of distance to the goal whereas a cost function defines the effort required to traverse a given path.

Several well-known search algorithms are applicable to satisfy the needs of Quoridor:

- Breadth-First search (BFS) – An uninformed search algorithm that explores all possible moves layer-by-layer. BFS will always return the shortest path, however, its uninformed layer-by-layer approach can have significant performance concerns.
- A* search – An informed search algorithm that combines both the use of a heuristic and cost function to improve search efficiency. A* always returns the shortest path and its focus on efficiency often makes it a more favourable choice over BFS.
- Bi-directional search – A technique that runs 2 simultaneous searches, one from the starting position and one from the goal and ends when the 2 searches meet, significantly reducing the number of explored nodes. This technique can be used in conjunction with A* search to produce an efficient pathfinding algorithm that will always return the most optimal path.

There exist many other search algorithms and techniques each suited to different type of problems. Some of these do not always return the most optimal path, however, for the purposes of Quoridor, returning the shortest path is essential. For deeper exploration into these methods *Search Methods in Artificial Intelligence* [4] is a great resource for further reading into this topic.

## 1.4 Quoridor’s Computational Complexity

Despite being seemingly a straightforward game, Quoridor holds a computational complexity greater than Chess [5]. This complexity arises from two factors:

- State-space complexity – The total number of unique positions that may exist in a game. Mertens [6] estimates the state-space complexity of Quoridor to be roughly $3.9905 \times 10^{42}$.
- Game-tree size – The total number of possible games that can be played. Mertens [6] estimates the game-tree size of Quoridor to be roughly $1.7884 \times 10^{162}$.

Given the significance of these figures, techniques such as minimax alone are infeasible for solving the game exhaustively. This illustrates a significant challenge in developing a powerful AI agent capable of making strong moves in a reasonable time frame. However, Mertens suggests that minimax search is not worthless and highlights the importance of alpha beta pruning and limiting the depth of search to address these challenges [6].

## 1.5 Artificial Intelligence Techniques

### 1.5.1 Minimax

Minimax is a decision-making algorithm that, in the context of the board game Quoridor, works by evaluating all possible future board states in a decision tree. A decision tree is a rooted, directed tree $T = (V, E)$ where each node $v \in V$ represents a unique board state and each directed edge $e \in E$ corresponds to a legal move or action that transforms one board state to another. The root of the tree represents the current game state, and the children of a node represent all possible legal states reachable in 1 move.

In 2-player games, it is clear to see that each level of the decision tree represents a turn alternating between the 2 players (see Figure 8). Therefore, we can label these levels MIN and MAX such that:

- MAX is the player whose value we want to maximise, in terms of game playing AI, MAX would be the AI agent.
- MIN is the player whose value we want to minimise, in terms of game playing AI, this would be the AI’s opponent.

A game-tree is the complete decision-tree of a game, starting from its initial state and branching to all final states of the game, represented by the terminal nodes. The minimax algorithm works by recursively searching the decision tree, starting at the terminal nodes it evaluates the board state by determining whether it resulted in a win, draw or loss and assigns that node a value accordingly. For instance, in Figure 8, -1 represents a loss, 0 represents a draw and 1 represents a win. Once all terminal nodes are evaluated, their values are propagated up to their parents:

- At MAX nodes the algorithm chooses the maximum value from its child nodes.
- At MIN nodes the algorithm chooses the minimum value from its child nodes.

This recursive evaluation continues until the root of the tree (the initial game state) is reached which gives the most optimal move for the MAX player. The minimax algorithm therefore assumes perfect play, that is the MIN player will always pick the move that minimises the MAX player's advantage.

Generally, decision-trees used in minimax are subsets of game-trees since often game-trees are too large to be fully analysed.

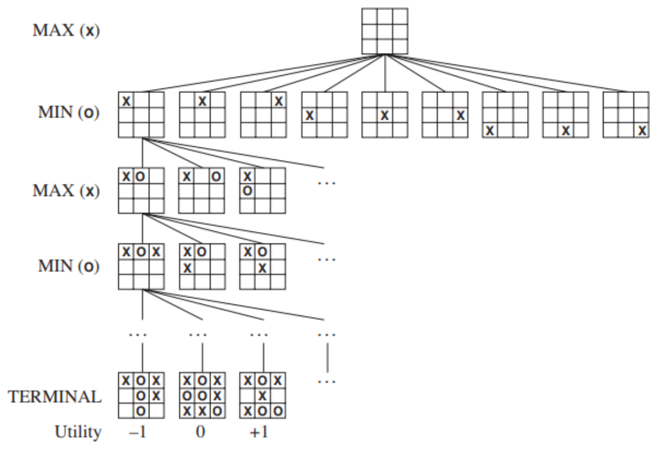


**Figure 8 - A partial game tree for the game of tic-tac-toe [7, p.163]**

Figure 8, shows a partial game tree for the game tic-tac-toe. Due to tic-tac-toe's minimal complexity, it is possible to create a decision-tree that represents the whole game-tree. Since optimal play from any position can be determined using this tree, tic-tac-toe is considered a strongly solved game, [8] meaning the outcome of the game can be correctly predicted from any position. A strongly solved game is one where an optimal strategy for both players is known for every possible state [9].

As mentioned previously, the complexity of Quoridor is too large and therefore, currently no winning strategy for Quoridor is known. As a result, the minimax approach must be adapted to limit the depth of the decision trees considered. To do this we must define an evaluation function that, given a board state, heuristically evaluates its value.

### 1.5.2 Evaluation Function

Since in a depth limited search whether a search path results in a win or loss can no longer always be determined, there needs to be some way to evaluate the board states at the depth being searched to. This is called an evaluation function and is defined as:

$$\boldsymbol{Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)}$$

**Equation 1 - Evaluation function equation [7, p.172]**

This function consists of the sum of weighted features where each $w_i$ is a weight and each $f_i$ is a feature of the position. As mentioned by Mertens, [6] not much research has been done into well performing evaluation features for Quoridor, and this still holds true today. The implementation described in this report proposes its own set of features (see Section 3.2.3). Some of these features are inspired by the features proposed by Glendenning [10], mainly features regarding shortest path and number of walls remaining.

### 1.5.3 Alpha-Beta pruning

Along with limiting the depth of search using an evaluation function, alpha-beta pruning can be applied to the minimax algorithm to reduce the search space, improving computational efficiency without affecting the correctness of the decision-making process. Alpha-beta pruning is a technique that eliminates redundant branches that do not need to be explored, thereby reducing the number of board states that must be evaluated.

It defines two values:

- Alpha ($\alpha$) – The best value that MAX can guarantee itself.
- Beta ($\beta$) – The best value that MIN can force MAX to accept.

As the algorithm explores the decision tree, it updates $\alpha$ and $\beta$ at each node:

- At MAX nodes, the algorithm selects the highest value among its children and updates $\alpha$.
- At MIN nodes, the algorithm selects the lowest value among its children and updates $\beta$.

Pruning occurs when $\beta \leq \alpha$, which indicates that a node's evaluation cannot influence the final decision, allowing the algorithm to safely ignore that subtree.

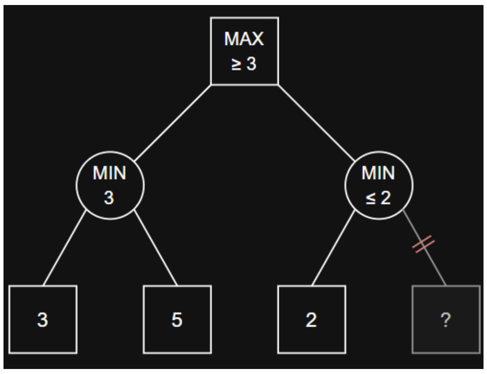


**Figure 9 - Alpha-beta pruning example**

The example in Figure 9 shows this technique as follows:

1. MAX evaluates the left MIN subtree first:
    a. MIN selects the minimum of its children, MIN = 3.
    b. MAX now knows its best guaranteed value so far, $\alpha = 3$.
2. MAX now moves to the right MIN subtree:
    a. The first child has a value of 2.
    b. Since MIN always picks the lowest value, MIN might return $\leq 2$, $\beta = 2$.
    c. But since 2 is already less than $\alpha = 3$, MAX will never choose this branch.
    d. Therefore, the second child of this MIN node does not need to be evaluated.

While this example (Figure 9), illustrates a small decision tree, in larger search spaces, alpha-beta pruning can eliminate entire subtrees from evaluation. This results in a reduction of the branching factor, enabling deeper searches and reducing computational cost.

Russel and Norvig [7] highlight the importance of move ordering when it comes to the effectiveness of alpha-beta pruning. That is, moves should be ordered such that the best moves are evaluated first. This approach is explored in the implementation of this project, however, ultimately ruled out due to performance issues as discussed in Section 3.2.2.

## 1.6 Genetic algorithms

Genetic algorithms are an optimisation technique that are inspired by the principle of natural selection and genetics. They operate on a population of potential solutions, evolving them over multiple generations to converge on an optimal solution.

With each successive generation three key genetic operations are applied:

- Selection – The process of choosing the fittest individuals from the current population to pass their traits to the next generation.

- Crossover – The process of combining aspects from two selected parent solutions to create new offspring.
- Mutation – The process of introducing small random changes to the traits of the offspring introducing diversity. This diversity is required to prevent the algorithm from converging on a suboptimal local minima.

For the Quoridor agent described in this report a simple genetic algorithm approach is used to train the weights of the features, of the evaluation function. While Glendenning, [10] has conducted extensive research into using genetic algorithms for training Quoridor playing agents, the approach taken in this project is more limited in scope due to time constraints. The primary goal of this project is to deliver a software implementation of Quoridor that includes a reasonably competitive AI agent, rather than a fully optimised learning system.

## 1.7 Other Approaches

This section will mention some of the other possible approaches to building a Quoridor playing agent that were not explored during the development of this project.

### 1.7.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is another search algorithm that is commonly used for decision-making in board games. Instead of exploring all possible moves and assuming perfect play, like minimax, it relies on randomised simulations to estimate the best possible moves. It starts at a root node representing the current state of the board and completes rounds of the search to decide its next move. Each round of MCTS consists of four main stages [11]:

1. Selection – The algorithm selects successive child nodes balancing exploration (trying new moves) and exploitation (focusing on strong known moves) until a leaf node is reached.
2. Expansion – Unless the leaf node is a final state (i.e. the game is over), one or more child nodes are created, and one is chosen for further exploration.
3. Simulation (Rollout) – From the chosen node a playout is performed where moves are chosen randomly or based on a heuristic until the game reaches a terminal state.
4. Backpropagation – The result of the simulation is used to update the values along the path taken informing the next round of the search.

The strength of the algorithm improves with the number of simulations performed. The algorithm further differs from minimax due to it not requiring an evaluation function. Since moves are explored through random or heuristically guided simulations an explicit board evaluation function is not required and therefore, no game playing knowledge or experience is required to implement MCTS. MCTS can therefore provide a balance between computation time and explored depth whereas, minimax, in the context of Quoridor, is limited in the depth it can explore due to a high branching factor.

This made MCTS a strong competitor for the agent implemented in this report however, minimax was ultimately chosen due to practical considerations along with the desire to learn and become more experienced with the game through the design of an evaluation function.

### 1.7.2 Machine Learning

Machine learning techniques such as deep neural networks have been successfully applied to board games such as chess. A key example is the Stockfish engine, which in 2024 removed its hand-crafted evaluation function and transitioned to a fully neural network-based approach [12]. This allows the engine to evaluate board states using a trained neural network.

Unlike chess, there is limited research on machine learning based Quoridor agents, meaning there is little existing work to build on. Additionally, Quoridor lacks publicly available datasets of high-quality games, making training an effective neural network infeasible within the time constraints of this project. Furthermore, the creation of such an agent would require significant computational resources and time, making it impractical for this study.

Due to these challenges, this report focuses on traditional search-based methods using a hand-crafted evaluation function. While the parameters of the evaluation function are trained using machine learning techniques, the development of a fully machine learning driven Quoridor agent is a topic for future research.

## 1.8 Past Research and Implementations

Whilst research into the board game Quoridor is still fairly limited, there are a few notable papers and implementations that helped guide this project.

*Mastering Quoridor* [10]

This highly cited paper proposes an AI agent that utilises a minimax algorithm and evaluates board states using a set of hand-crafted features. A subset of the features proposed in the paper were taken into consideration and repurposed with the evaluation function described in Section 3.2.3.

However, the study concludes that the proposed agent was not strong enough to consistently defeat human players. One possible reason for this is the complexity of the evaluation function. A function with too many features may introduce noise, making it harder for the agent to generalise efficiently. Therefore, this project explores using a simplified evaluation function that prioritises features that seem significant at all stages of the game. A simpler evaluation function also enhances the performance of the genetic algorithm, as it doesn't need to consider as many parameters, making convergence on a strong set of weights achievable within a feasible time frame.

*Monte Carlo Tree Search for Quoridor* [13]

This paper proposes an AI agent that utilises Monte Carlo Tree Search (MCTS), resulting in an agent that outperforms the minimax agent proposed in *Mastering Quoridor* [10]. Given this, both implementations will serve as benchmarks to evaluate the strength of the agent developed in this project. As part of the evaluation (see Section 4.2), the proposed agent will compete against both existing agents to assess its relative performance. This will provide insights into the strength of the evaluation function and the difference in the search-based approaches introduced in this project.

*Quoridor AI based on Monte Carlo Tree Search* [14]

This implementation proposes a set of heuristics based on wall placements to reduce the branching factor and improve the search efficiency. These heuristics help prioritise the most promising moves, ensuring the search remains computationally feasible. The application of these heuristics is detailed in Section 3.2.2 and is one of the key optimisation components of the agent developed in this project. Additionally, this implementation was also chosen as a benchmark to evaluate the strength of the agent developed in this project.

While existing research and implementations of Quoridor and AI agents have primarily focused on algorithmic development and performance benchmarks, they generally lack user testing and evaluation. This project, however, aims to deliver a software product that provides value not only through a competitive AI agent but also considering user experience.

# Chapter 2
# Methodology

## 2.1 Initial Project Decisions

Given the evolving nature of this project, an agile methodology was used to divide the work into sprints, where the results from prior sprints would guide the next. This allowed for a great deal of flexibility during the project's lifespan. One of the earliest priorities of the project was delivering the core game implementation first before considering additional extensions. These extensions included networking (remote play), Artificial Intelligence (AI) playing agents and game variations (differing board shapes or sizes, differing player counts). Once the game implementation was finished, AI was chosen as the primary area of focus for the remainder of the project.

Despite the flexibility of this approach, some key initial decisions still had to be established to facilitate the development of this project, within the time frame provided. One key decision, was picking the language this project would be developed in. After evaluating several options, Python was selected as the primary language for this project.

Python was chosen due to its extensive collection of libraries such as *Pygame*, *Pathfinding*, *NumPy* and *Pytest* all of which were essential to the development of this project which is covered in Appendix B. These libraries, along with Python's ease of development allowed for quick prototyping and efficient iterative development.

Other languages were considered, including C++ and JavaScript. C++ was an attractive option due to its significant performance benefits. However, the increased complexity of the language and longer development time outweighed these performance benefits. JavaScript was also considered due to its ability to integrate within HTML and CSS allowing for the easy creation of a dynamic and visually appealing user interface. Additionally, the ease of deploying the project to the web would have made the project a lot more accessible, which would have been beneficial for user testing. However, due to Python's ease of use and a lack of powerful libraries in the other languages, Python was ultimately chosen.

To accelerate the initial implementation, a YouTube series covering the creation of a checkers game with Minimax in Pygame [15] [16] was used as a baseline for this project. Checkers' similarity with Quoridor, regarding its use of a board and moving pieces, along with the tutorial's modular approach, made it easy to modify it to the needs of Quoridor and a solid foundation for the project.

Version control through Git and GitHub was also used throughout the project to ensure the project was sufficiently backed up: https://github.com/Matt-Bowler/Individual-Project.

By making these key decisions early on the project was able to progress efficiently while providing flexibility for future decisions and enhancements. With these initial decisions established, the next sections will outline each phase of the project's development.

## 2.2 Sprint 1 – Initial game implementation

The goal for the first sprint was to implement the core game functionality and create a basic graphical user interface (GUI) to show the board state. The sprint was divided into two main parts: the first half focused on setting up the GUI and implementing basic pawn movement, while the second half focused on adding the wall placement mechanics.

In the first half of the sprint, the primary objective was to create the GUI to display the board and pieces. The movement logic was then developed and integrated with the GUI to allow for the selection of pieces and the ability to move them to valid locations. The highlighting of valid moves once a piece was selected was added for better usability by ensuring that players could clearly see where their pawn could move.

The second half of the sprint, focused on implementing the wall placement logic and updating the GUI to facilitate this. This included creating the mechanics for players to place walls on the board, as well as visually representing where walls could be placed by adding dividers between the squares. Valid wall position highlighting through hovering over the dividers was added to the GUI to ensure players could easily identify legal wall positions.

By the end of Sprint 1, the core game interface was implemented with all move and wall placement logic considered. This laid the foundation for the next sprints where the AI playing agent would be developed.

## 2.3 Sprint 2 – AI development and initial evaluation function

Now the core game and rules had been implemented, focus shifted on developing the AI playing agent for the game. The primary objective of this sprint, was to implement the minimax algorithm and then apply various optimisation techniques (see Section 3.2.2) to reduce the computation time. Then a simple evaluation function was defined to ensure that the AI was making logical moves.

By the end of Sprint 2, a working AI agent had been implemented that was capable of making decisions in a reasonable time frame (around 4 seconds per move). This focus on optimisation first was crucial for the next sprint which relies on a self-play approach to improve the AI’s decision making. Without these early optimisations, self-play, where the AI plays against itself to refine its evaluation function, would have been infeasible within a reasonable time frame. Additionally, excessive computation times could have limited the number of self-play iterations possible, hindering the AI’s ability to improve.

## 2.4 Sprint 3 – Evaluation function training

Now with a functional AI agent, the focus of this sprint was to improve its decision-making by refining the evaluation function. The first half of this sprint, focused on exploring a variety of features by using previous experience with the game. This was done through many iterations of manual adjusting and playing against the AI until a solid set of concise features were defined.

Once the features of the evaluation function were implemented, they could begin to be weighted and trained through a genetic algorithm. The second half of this sprint was focused on implementing this genetic algorithm and then conducting series of self-play games and picking the weights with the highest win rates to use in the next generations.

By the end of Sprint 3, the evaluation function had been significantly refined which resulted in a stronger AI capable of making more strategic moves. This marked the completion of the AI’s development, with the final phase of development focusing on integrating all components together.

## 2.4 Sprint 4 – Final GUI design

With the AI’s development complete, the final sprint focused on adding the final GUI components including a main menu for selecting between different game modes (Human vs Human, AI vs Human, AI vs AI) and a game over screen, with options to restart exit. Additional GUI features included a wall counter for each player and a progress bar for AI moves, providing visual feedback and preventing users from mistakenly assuming the program was unresponsive when the AI would make long decisions.

By the end of Sprint 4, the GUI was fully integrated with the underlying game logic, marking the completion of the software deliverable. This allowed the remainder of the projects time to be focused on evaluating the product. This involved user testing and assessing the AI’s performance compared to other existing agents, as discussed in Chapter 4.

# Chapter 3
# Implementation and Validation

## 3.1 Game implementation

### 3.1.1 Board Class

The board class is responsible for keeping track of the board state during the game and enforcing the rules of the game. The board is represented as a 2D array stored in the board class. Each cell of the board array either contains a Piece object or a 0 which represents an empty space. The Piece class is responsible for tracking a piece's position (row and column) and its colour.

Two sets are defined for storing horizontal and vertical walls. These are sets of tuples representing the row and column the walls occupy. Sets are used instead of arrays where possible due to their faster lookup times. The board class also contains a set called valid_walls consisting of Wall objects. The Wall class is responsible for tracking a wall's position (row and column) and its orientation (horizontal or vertical). The valid_walls set is precomputed to store all valid walls when the board object is created. Once a wall is placed on the board, this set is updated to remove the walls invalidated by this wall placement (overlapping walls and crossing walls). However, it is not so easy to determine which walls in this set become invalid due to blocking a player's path to their goal and therefore, further validation is required.

The board class also contains methods used for move and wall validation. The method get_valid_moves(piece) returns a set of the valid positions a piece can move to, based on the game rules. The method is_valid_wall(wall) first checks whether the given Wall object is in the valid_walls set. If the wall is in the set, then it is temporarily placed on the board, and a pathfinding algorithm is used to verify that each player still has at least one unobstructed path to their goal. If a path exists for both players, the wall placement is considered legal. These validation functions are used in the Game and AI classes to validate player and AI moves.

The board class also contains the evaluation function used in the minimax algorithm. This function takes the current board state and evaluates its strength.

### 3.1.2 Game Class

The game class is responsible for managing the overall state of the game and serves as the main point of interaction for the players. It keeps track of whose turn it is and handles win conditions. The class is initialised containing an instance of the Board class which it uses to access and update the current state of the board and validate moves.

The class can also be initialised with a Pygame window, allowing it to serve a GUI and contains methods for updating this GUI to render updated board states.

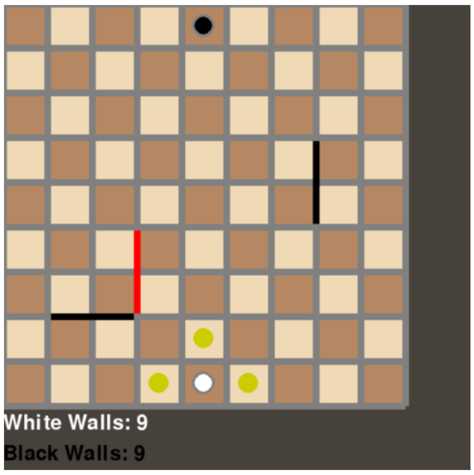


**Figure 10 - Example of the games GUI**

Figure 10, shows an example of this GUI. The yellow circles appear when a piece has been selected and shows the places where the piece can move to. The red rectangle appears when hovering over a divider position where a valid wall can be placed.

### 3.1.3 Pathfinding Module

The pathfinding module is built using the external *pathfinding* Python library, which provides a variety of standard pathfinding algorithms such as BFS, A* and Dijkstra's. To integrate these algorithms into the game implementation, the library's Grid class was extended to create a new class called QuoridorGrid. Specifically, the neighbors(node) method was overridden to account for placed walls, where neighbours of a node were removed given, there was a wall between them.

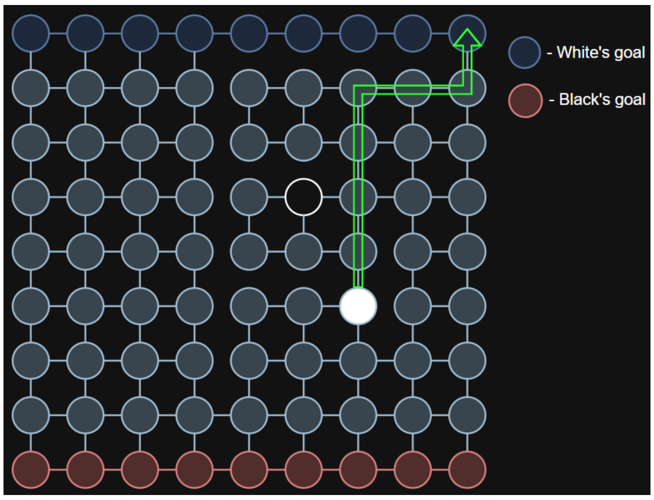


**Figure 11 - Grid representation of the board.**

This allowed the board to be represented as a grid such as the example shown in Figure 11. Nodes represent valid piece positions, while edges represent valid moves with respect to wall placements. The green arrow highlights the shortest path for the white player in this configuration. Now one of the library's pathfinding algorithms could be applied to the grid. After various testing, the algorithm chosen from the library, was the bi-directional A* algorithm as this was found to be the most efficient for this application. The algorithm was used to create two methods for the pathfinding module:

- path_exists(board) – Verifies whether a path exists for either player to reach their goal row and is used in is_valid_wall() to enforce that rule.
- shortest_path(piece) – Finds the shortest path for a piece to any of its goal nodes and is used in the evaluation function (see Section 3.2.3) to inform AI decision making.

## 3.2 AI implementation

### 3.2.1 AI class

The AI class contains the minimax algorithm along with some methods for getting valid moves, using the Board class, and filtering them for decision making. To simplify the implementation of the minimax algorithm, a variation called negamax is used. Negamax is more streamlined for 2-player games by removing the need to alternate between maximising and minimizing layers. Instead of evaluating from the perspective of both players separately, negamax assumes the score from one player's perspective is simply the negation of the opponent's score [17].

```
function negamax(board, depth, alpha, beta, color)
    if depth == 0 or board is in terminal state then
        return evaluate(board, color)
    best_move ← null
    best_value ← $-\infty$
    for move in filtered_moves do
        evaluation ← -negamax(move, depth – 1, -beta, -alpha, -color)
        if evaluation > best_value then
            best_value ← evaluation
            best_move ← move
        alpha ← max(alpha, evaluation)
        if beta ≤ alpha then                              /* alpha-beta pruning */
            break out of for loop
    return best_move, best_value
```

**Figure 12 - Negamax function with alpha-beta pruning**

### 3.2.2 Optimisations

The main performance optimisation comes from limiting the depth of the minimax search to a depth of 2. The use of alpha-beta pruning in this search resulted in around an average 75% speed up in decision making. As mentioned in Section 1.5.3 move ordering can improve pruning efficiency. However, when implemented using a simple heuristic to order the moves, the overhead introduced by sorting and evaluating the moves outweighed the potential performance gain. This is largely due to the shallow search depth used, where the benefits of move ordering are minimal. An additional optimisation technique was to heuristically remove wall positions from consideration, considering only those likely to be strategically useful. For each AI turn, potential wall moves were filtered using the following checks.

1. Proximity to players: Walls within a Manhattan distance of 3 squares from either player's current position were considered. This prioritised walls that were more likely impact player movement directly.
2. Adjacency to existing walls: Walls that were adjacent to existing walls (i.e. extending a wall structure) were considered. This prioritised creating meaningful obstructions.

Walls that did not meet these criteria were discarded before evaluation. This significantly reduced the branching factor without noticeably affecting decision quality. However, this approach could potentially overlook less obvious but strategically strong wall placements, which is a trade-off accepted for performance reasons.

For each move considered by the minimax algorithm a copy of the board object had to be created. Originally, this was done using a full deepcopy, but profiling, using Python's cProfile, revealed that this introduced significant performance overhead. Therefore, a method called partial_deepcopy() was added to the AI class. It uses shallow copying where possible, only deep copying the Piece objects within the board array, as these were the only components modified during simulated moves.

Zobrist hashing, [18] is another optimisation technique typically used in minimax chess AI. It is used to implement transposition tables, which is a method to cache previous board evaluations to prevent re-searching. A simple transposition table was explored however, with the limited depth of search, it was found that on average only around 3 cache hits were occurring per move. While the number of cache hits increased at deeper searches, the complexity of implementing a full caching layer using Zobrist hashing, outweighed the minimal performance gain at the chosen depth level. As a result, caching was not added.

With these optimisations implemented, the AI was able to make decisions in approximately 4 seconds on average at a search depth of 2, and around 20 seconds at a search depth of 3. While 20 seconds may still be an acceptable time for decision making, the depth of 2 was ultimately chosen to balance the agent's responsiveness and strength.

### 3.2.3 Evaluation Function

The evaluation function used by this agent consists of 4 main features that provide a heuristic estimate of a board states favourability:

- Path length difference – The difference between the shortest paths to the goal for both players: $Shortest\ path\ of\ opponent - Shortest\ path\ of\ agent$
- Wall bonus – The difference in the number of walls remaining for both players, favouring the player who has more walls left.
- Proximity bonus – Number of walls that are close (within one square) to the opponent's piece as these are more likely to restrict their movement.
- Forward bonus – Difference between how far each player has advanced to their goal (measured by their row position) encouraging forward movement.

This set of simplistic features results in an AI agent that balances progressing to its own goal, obstructing its opponents progress and conserving walls. A small random factor is also added to the function to make it less deterministic and introduce variation in the AI's decisions.

## 3.3 Evaluation Function Training

### 3.3.1 Setup

Given the modularity of the game implementation, creating a separate module for AI training was simple. Three new classes were defined:

- TrainingBoard – An extension of the Board class that overrides the evaluation function to allow weights to be assigned to the features. Additionally, the random factor in the evaluation function is removed for training purposes.
- TrainingGame – An extension of the Game class that takes an array of weights and creates an instance of TrainingBoard with them when initialised.
- TrainingAI – An extension of the AI class that overrides the partial_deepcopy() function to return an instance of TrainingBoard when called.

Then using these training classes, a machine learning module was created to implement the basic genetic algorithm used to train the evaluation function. This module contains the following key functions:

- generate_random_weights() – Uses *NumPy* to generate a random array of 4 weights (one for each feature of the evaluation function) with each weight being a float in the range 0-10.
- self_play() – Simulates a number of games between two AI agents, each using a different set of weights. It returns the number of wins for each agent after all the games have been played.
- evaluate_population() – Runs self_play for each set of weights in a population and returns them as an array sorted by their win rate.
- select_top_performers() – Selects the top percentage of weights based on a specified retention ratio.
- crossover() – Uses *NumPy* to combine two parent weight arrays into a new child array of weights. It selects a random index, weights before the index are taken from one parent, and the remaining weights are taken from the other.
- mutate_weights() – Uses *NumPy* to apply a small mutation to a random weight in an array of weights based on a specified mutation rate.
- generate_next_generation() – Creates the next generation of weight sets by repeatedly selecting parents, applying crossover and mutation until the population reaches the desired population size.

### 3.3.2 Process

The training began with the initial generation being a large population of 50 randomly generated weight sets. This large initial population size was chosen to ensure a high level of diversity was considered at the beginning, allowing the algorithm to explore a broad range of potential solutions. Subsequent generations used a population size of 10, to focus on rapidly improving the most promising candidates from the initial sample.

In each generation, every weight set in the population was evaluated using the evaluate_population() function, where each set of weights would play 10 games against an opponent consisting of randomly generated weights. The reason the agents were evaluated against random opponents as opposed to each other was to prevent the algorithm from converging on a population that only performs well against its peers. The idea was that a candidate that consistently performs well against a wide range of randomised opponents is more likely to generate a strong general play style. Over enough generations, weaker strategies are naturally bred out.

After evaluation, the select_top_perfomers() function is used to select the top 40% of candidates based on their win rates. These top performing weight sets are then used to generate the next generation. This is done through repeated use of the crossover() function and applying occasional mutations using mutate_weights() to introduce variability. A mutation rate of 10% was chosen, meaning each child has a 10% chance that one of its weights will be slightly altered.

This process was repeated for 12 generations. Throughout training, a timeout mechanism used for cases where both agents would get stuck in an infinite loop (see Section 3.4.2). This timeout was set at 2 minutes and if any game exceeded this limit, it was aborted, and a loss was assigned to the candidate. The overall training process took around 24 hours to complete.

### 3.3.3 Results

Following the completion of the training process, the results of the evaluation training are presented below.

| **Feature Name** | Path length difference | Wall Bonus | Proximity Bonus | Forward Bonus |
|---|---|---|---|---|
| **Feature Weight** | 9.175 | 2.337 | 2.483 | 0.548 |

**Table 1 - Results of evaluation function training**

Table 1 shows the final weights determined in the 12th generation. These results were then hardcoded into the evaluation function in the Board class. The clear primary feature of a strong evaluation function was shortest path difference. Agents that prioritised shortening their own path while lengthening their opponents through both wall placements and movement, consistently performed best. Both the wall and proximity bonus were assigned a moderate weight, indicating that placing walls close to the opponent and conserving walls had a secondary but still significant role. The relatively low value assigned to the forward bonus feature implies that forward movement on its own was less important compared to the other features.

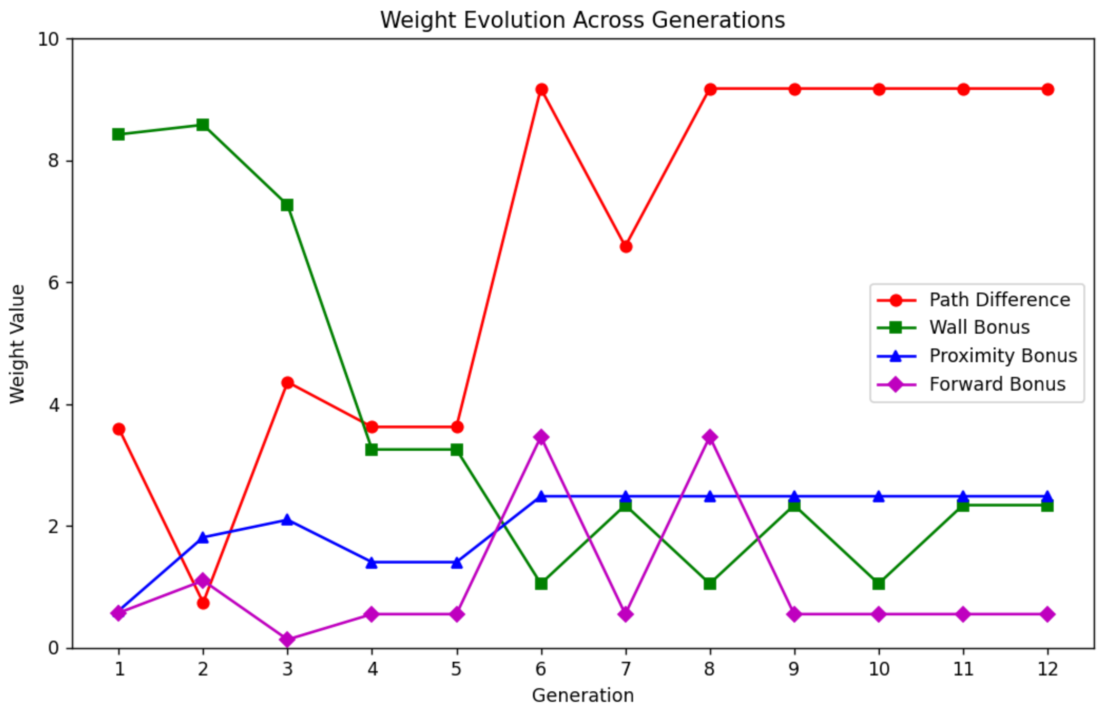


**Figure 13 - Graph of weight evolution across each generation**

Figure 13, shows the weight values of the top performer for each generation. The exploratory nature of the training process can be seen, with the weight values fluctuating significantly until eventually plateauing in the later generations. This can be clearly seen in the path difference and wall bonus features where in early generations, path difference starts with a low weight while wall bonus is relatively high. Over subsequent generations, the weight assigned to the path difference feature steadily increases, reflecting its growing importance, while the weight of the wall bonus gradually decreases. Both eventually stabilise around their optimal values in the later generations.

## 3.4 Validation and Testing

### 3.4.1 Unit testing

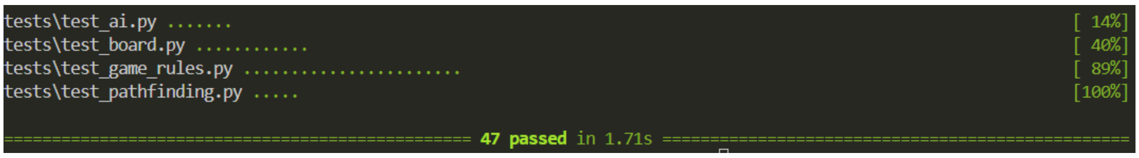
Using the external library *Pytest,* a suite of unit tests was created to validate the individual components of the game. These unit tests ensured that the components were functioning as intended and to prevent regressions during future development.

Board class testing focused on ensuring that the board state is correctly updated following piece moves or wall placements. Additionally, methods such as get_valid_moves() was tested to validate it return the expected moves in various scenarios.

The tests for the Game class focus on turn handling, win condition detection along with testing the game rules, defined in Section 1.2.3, are correctly implemented.

The tests for the AI class validated that the negamax() function returned legal moves. The wall filtering heuristic was tested to ensure it properly discarded irrelevant walls. Additionally, the functions used to simulate moves during decision making were tested to ensure the new copy of the board state reflected the simulated move.

The tests for the pathfinding module included testing the grid representation of the board correctly reflected the board's state. Path detection was tested by simulating wall moves that would block a player's path entirely and validating that the path_exists() function would identify this. The shortest_path() function was also tested to ensure that it returns the expected path lengths in various situations.



```
tests\test_ai.py .......                                                        [ 14%]
tests\test_board.py ............                                                [ 40%]
tests\test_game_rules.py .......................                                [ 89%]
tests\test_pathfinding.py .....                                                 [100%]

============================================= 47 passed in 1.71s =============================================
```

**Figure 14 - Results of the unit testing**

### 3.4.2 AI testing

During the development of the AI agent manual testing was conducted through playing against the AI and observing AI versus AI games. This hands-on approach helped identify behavioural issues not captured through unit testing. A key issue identified was an infinite loop problem, where the AI would repeat the same moves indefinitely without making any progress. To mitigate this a random factor was added to the evaluation function in hopes it would make the agent less likely to repeat identical sequences of moves.

Training the evaluation function, combined with the randomness introduced into decision making, reduced the frequency of this issue, though it still occasionally persists.

# Chapter 4
# Results, Evaluation and Discussion

## 4.1 User Playtesting

### 4.1.1 Method

Once the implementation was finished, user testing was used to evaluate both the game implementation and AI agent. Five participants of varying familiarity with Quoridor and similar strategy games were recruited. At the beginning of the session, the participant was given a copy of the information sheet in Appendix C. After reading the sheet and agreeing to participate in the test, the rules of the game were clearly explained to the participant or briefly summarised in cases where the participant was already familiar with the game.

The participant played a few games with the researcher to become familiar with the interface before facing the AI opponent. The participant played as many games as they chose against the AI opponent. During these games observations were collected regarding the AI’s playstyle against the participant. At the end of each game the winner was noted, along with any verbal feedback from the participant regarding the AI’s strategy and performance. At the end of the session the participant was presented with a questionnaire for them to complete regarding their experience with the software.

### 4.1.2 Results

#### 4.1.2.1 User Feedback

The questionnaire participants completed at the end of the session consisted of linear scale questions (rated from 1 to 10), yes/no questions and space for open ended comments.

| **Question** | **Average Rating** |
|---|---|
| How easy was it for you to work out what moves were available to you? | 8.4 |
| How easy was it to place walls where you intended? | 8.0 |
| How visually clear was the board state at all times? | 9.6 |
| How visually appealing was the games interface? | 7.0 |
| How challenging did you find the AI opponent? | 8.0 |
| Did the AI's moves seem logical and strategic? | 8.0 |
| How would you rate the AI's response time? | 8.0 |
| How enjoyable was the overall experience? | 9.4 |

**Table 2 - Results of the linear scale questions from the questionnaire**

For the yes/no questions 100% of participants voted that they never felt unsure about the game’s rules due to the game’s interface. Additionally, 100% of participants said they would recommend this implementation to a beginner trying to learn the game.

Overall, the feedback from participants was generally positive across all areas of the game. Participants found the game easy to interact with, intuitive and overall enjoyable. The lowest-scoring category, visual appeal (7.0) still received a decent rating, though it suggests some room for visual improvement, with one participant stating, “it didn’t look the best”.

No major issues with the game were identified through the user testing however, some general improvements suggested by participants included:

- Adding an undo/redo functionality to aid learning.
- Introducing a difficulty setting for the AI to suit different skill levels.
- Colour coordinated walls to see which player had placed which walls.
- Adding a move timer option.

The AI agent was also rated positively with participants finding the AI to be challenging, strategic and responsive. Participants noted that the AI was very strong in the start to mid game, often catching participants of guard and trapping them in unfavourable positions. The AI was very effective at creating positions which forced a longer path for the opponent while also ensuring its own path was short and difficult for the participant to extend.

However, every participant identified that the AI was often too aggressive with its wall usage, burning through all its walls early on, often resulting in unfavourable endgames for the AI since it could no longer defend against the participant’s progress. Additionally, the loop issue mentioned in Section 3.4.2 occasionally occurred in some of the endgame positions, often causing the AI to lose positions which they were favoured to win. These behaviours highlight an issue of short-sightedness in the AI’s decision-making. This is most likely due to it only being able to look two moves ahead, which restricts its ability to account for long-term consequences of its actions.

#### 4.1.2.2 Human versus AI

The results of the human versus AI games are shown in the table below.

| Participant Skill Level | Games played | AI wins | AI win percentage |
|---|---|---|---|
| Advanced | 3 | 0 | 0% |
| Beginner | 10 | 7 | 70% |
| Beginner | 4 | 3 | 75% |
| Beginner | 4 | 2 | 50% |
| Advanced | 4 | 2 | 50% |
| **TOTAL** | **25** | **14** | **56%** |

**Table 3 - Results of Human vs AI games**

The AI performed notably better against beginner participants, achieving a win rate of 66% across those games. Against advanced players, however, results were more mixed with one participant winning all games and another resulting in an even split. The more advanced participants were able to exploit the patterns of weakness regarding aggressive wall usage and short-sightedness. They did this by baiting the AI agent to exhaust its walls early while they conserved their own to apply pressure in the endgame.

## 4.2 AI Playtesting

### 4.2.1 Method

AI playtesting was used to benchmark the strength of AI created in this project against existing AI agents for the game Quoridor. The 3 main implementations used for comparison are described in Section 1.8, these being: Glendenning's Minimax agent [10], Respall's MCTS agent [13] and Lee's MCTS agent [14]. For each implementation, human versus AI games were set up independently with one program playing as white being the AI and the other program playing as black being the AI. Then moves were manually mirrored on each implementation until the game was over. The results of these games along with differences in AI playing style were recorded.

For Glendenning's implementation a fixed move time of 5 seconds was used to align with the average decision time of the agent developed in this project. As Glendenning's agent was deterministic, and therefore every game would be identical, only one game was played per weight set. However, to evaluate the performance across different styles, the agent was tested against 3 of the weight sets proposed in the original paper, namely: psi1 (weight set of best fit), digamma2 and omicron1.

Respall's MCTS-based agent was configured to only use 1000 simulations per move since this resulted in a similar average move time as the agent created in this project. The 60k agent was excluded from testing due to it taking on average around 7 minutes per move making it impractical for comparison. However, due to the non-deterministic nature of the MCTS algorithm, games played against this agent differed, therefore 4 games were played against the 1000 simulation agent to benchmark performance.

For Lee's agent, 20,000 simulations were used per move, again due to it matching roughly the decision time of the agent in this project. Five games were played against this 20k agent. Additionally, to see how the agent in this project performs against stronger opponents, 3 games were played against Lee's 60k agent. This was feasible due to Lee's 60k agent only taking approximately 8 seconds per move.

### 4.2.2 Results

The table below shows the results of the games including a column for undecided games, where both AIs got stuck in a loop and the game was ultimately aborted. The loop issue, as discussed in Section 3.4.2, affected all three AI implementations to varying degrees. Despite differences in design and decision-making methods, none of the agents were entirely immune to this loop issue, highlighting a common challenge in Quoridor AI design.

| Opponent Agent | Games Played | Games Won | Undecided Games | Win Percentage |
|---|---|---|---|---|
| Glendenning psi1 | 1 | 1 | 0 | 100% |
| Glendenning digamma2 | 1 | 1 | 0 | 100% |
| Glendenning omicron1 | 1 | 1 | 0 | 100% |
| Respall 1k agent | 4 | 3 | 1 | 75% |
| Lee 20k agent | 5 | 1 | 0 | 20% |
| Lee 60k agent | 3 | 0 | 0 | 0% |
| **TOTAL** | **15** | **7** | **1** | **47%** |

**Table 4 - Results of AI vs AI games**

The AI developed in this project performed strongly against other minimax-based agents, achieving a perfect win rate across the three weight sets tested from Glendenning's implementation. It also held up reasonably well against MCTS agents with limited simulation depth, with a win rate of 75% against Respall's 1k agent.

However, against Lee's 20k MCTS agent, which had a similar decision time to the AI developed in this project, performance dropped significantly, with only 1 win in 5 games. Additionally, against Lee's 60k agent, which took around twice the decision time, it was severely outclassed failing to win a single game. These results reinforce the short-sightedness identified during user playtesting demonstrating that strategies focused on deeper lookahead enabled by MCTS, outperform current minimax-based solutions, as hypothesised in Section 1.7.1.

## 4.3 Conclusions

Overall, this project successfully achieved its objectives to deliver a well-rounded software implementation of Quoridor featuring a user-friendly interface and competitive AI opponent. The delivered software implementation was rated positively by users during user testing. The AI agent showed strong performance in both human and AI benchmarks. While not always having the foresight to win all its games, its moves in general were strong and strategic. All key objectives were met in the timeframe of this project, while laying a solid foundation for future enhancements which are explored in the next section.

## 4.4 Future work

User feedback identified some potential key enhancements to the game implementation such as implementing AI difficulty levels and introducing additional mechanics. Given another sprint of development, these enhancements would be relatively simple to implement. Possible extensions have also been mentioned in the methodology chapter such as, networking and board size or shape options. The introduction of networking would allow the project to be deployed online, enabling remote multiplayer gameplay. Changes in board configurations could introduce fun alternative game modes and open new avenues for research into how different special layouts affect game playing strategy.

The evaluation function developed in this project proved to be effective however, there remains room for improvement. Further research into Quoridor specific features could allow for the development of a minimax agent with strong decision making across all stages of the game. Another key area for investigation is the loop issue observed in this project, which affected all AI agents tested. Research into loop detection and handling could fix a major issue in an otherwise strong AI playing agent.

A particularly promising direction for future work would be the development of a hybrid agent that combines both minimax and MCTS into its decision making. The combination of strong informed decision making through an evaluation function, with the deeper search capabilities of MCTS could produce a more robust agent, especially in end game situations where the current implementation tends to struggle.

Finally, as discussed in Section 1.7.2, the lack of a large dataset of high-level Quoridor games poses a significant challenge to the development of learning-based agents. This project could be easily extended to collect and store gameplay data, which over time could support machine learning approaches trained on real gameplay, opening new opportunities for AI research into Quoridor.

## List of References

1. Wikipedia. *Board games*. [Online]. 2025. [Accessed 27 March 2025]. Available from: https://en.wikipedia.org/w/index.php?title=Board_game&oldid=1281394859
2. Wikipedia. *Quoridor.* [Online]. 2025. [Accessed 27 March 2025]. Available from: https://en.wikipedia.org/w/index.php?title=Quoridor&oldid=1282325848
3. Elkies, N.D., Stanley, R.P. The mathematical knight. *The Mathematical Intelligencer* [Online]. 2003, **25**, pp.22–34 [Accessed 27 March] Available from: https://doi.org/10.1007/BF02985635
4. Khemani, D. *Search methods in Artificial Intelligence*. Cambridge: Cambridge University Press, 2024.
5. Wikipedia. *Game Complexity.* [Online]. 2025. [Accessed 27 March 2025]. Available from: https://en.wikipedia.org/w/index.php?title=Game_complexity&oldid=1268021280
6. Mertens, P.J.C. *A Quoridor-playing Agent.* BSc thesis, Maastricht University, 2006
7. Russel, S. Norvig, P. *Artificial Intelligence A Modern Approach.* Prentice Hall Series in Artificial Intelligence, 3rd Edition. Upper Saddle River: Pearson. 2010.
8. Allis, L.V. *Searching for solutions in games and artificial intelligence.* Doctoral Thesis, Maastricht University, 1994.
9. Wikipedia. *Solved Game.* [Online]. 2025 [Accessed 27 March 2025]. Available from: https://en.wikipedia.org/w/index.php?title=Solved_game&oldid=1281596884
10. Glendenning, L. *Mastering Quoridor.* BSc thesis, The University of New Mexico, 2005.
11. Chaslot, G. Winands, M.H.M, Jaap Van Den Herik, H. Uiterwijk, J.W.H.M. Bouzy, B. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation.* [Online]. 2008. **04**, pp343-357 [Accessed 27 March]. Available from: https://doi.org/10.1142/S1793005708001094
12. Stockfish, Stockfish 16.1. 24 Feb. *Stockfish Blog.* 2024. [Online]. [Accessed 27 March 2025]. Available from: https://stockfishchess.org/blog/2024/stockfish-16-1
13. Respall, V.M. Brown, J.A. Aslam, H. Monte Carlo Tree Search for Quoridor. In: *19th International Conference on Intelligent Games and Simulation, GAME-ON, 18/20 Sep, Dundee.* Ostend: EUROSIS, 2018 pp.5-9.
14. Lee, K. *quoridor-ai.* Github Repository [Online]. 2024. [Accessed 27 March]. Available from: https://github.com/gorisanson/quoridor-ai
15. Tech With Tim. *Python Checkers Tutorial.* [Online]. 2020. [Accessed 16 April]. Available from:

https://www.youtube.com/watch?v=vnd3RfeG3NM&list=PLzMcBGfZo4-lkJr3sqpikNyVzbNZLRiT3

16. Tech With Tim. *Python Checkers AI Tutorial.* [Online]. 2020. [Accessed 16 April]. Available from: https://www.youtube.com/watch?v=RjdrFHEgV2o&list=PLzMcBGfZo4-myY28wdQuJDBi8pCt-GIj6
17. Heineman, G.T. Pollice, G. Selkow, S. *Algorithms in a Nutshell.* California: O’Reilly Media Inc. 2008.
18. Zobist, A.L. *A New Hashing Method With Application For Game Playing.* Wisconsin: The University of Wisconsin. 1970.

# Appendix A
# Self-appraisal

## A.1 Critical self-evaluation

Overall, I would consider this project to be a success, with all its primary objectives delivered. The flexible approach I took to the project allowed me to manage my time efficiently and adapt the scope of the project where necessary. Background research was conducted as necessary with the initial phase of the project being quite sparse and spent exploring existing literature and implementations. This helped gain insight into the feasibility of the project and other approaches taken in the past. Deeper dives into background research were conducted during the development phases of the project. This allowed me to respond to problems as they arose and gain relevant insights as they were needed.

Biweekly meetings with my supervisor Dr Haiko Muller supported the iterative agile approach to development described in the Methodology chapter. These meetings allowed me to present on going work and gain feedback. Then the rest of the meeting be used to plan upcoming tasks. It was here that Dr Muller's domain expertise provided invaluable insights and guidance. These discussions helped by defining achievable goals in two-week periods, keeping the project always moving forward and allowing project time to be managed efficiently.

However, this project did not come without its issues. One such issue was a crucial oversight in the evaluation function training stage that cost around 24 hours of wasted time. Originally the agent would play 5 games against the same random agent in each generation. However, given the deterministic nature of the AI, these 5 games were identical and therefore, only 1 game was necessary to determine strength against another agent. This oversight was only caught once training had complete and could have been avoided with more careful observation during the earlier stages of the training process.

Another area that could have been improved was advanced planning for user testing. While the user testing conducted did provide very valuable insights, recruitment of user testers could have been improved. Participants were only recruited within the final months of the project making arranging playtesting sessions rather difficult. Additionally, if a larger focus on recruiting users earlier on was conducted, highly experienced players could have been involved in the testing. User evaluation could also have been incorporated into the development cycle and if done with highly experienced players, could have provided insights into how the game is played at a higher level. This potentially could have helped develop a more powerful evaluation function.

Lastly, the effort required to produce the final report was underestimated. While the development process was handled well with minimal issues, focus on the report should have been incorporated earlier on. This oversight meant when it was time to plan the report, at the end of the project, difficulties in defining its scope and structure arose. Additionally, during each significant development notes should have been taken since when writing the report, it was difficult to remember exactly what had happened in some of the earlier stages of development.

## A.2 Personal reflection and lessons learned

Altogether I am happy with how I conducted myself during the project's lifespan. Having never tackled a project of this scale in the past, initially I was rather overwhelmed. However, as the project progressed, I became more confident and now by the end feel my hard work has paid off.

One key lesson learned was how to handle a technical report. While report and long form writing usually makes me uncomfortable, some key skills were learned through the completion of this report. When it comes to report writing I often tend to overthink and overexplain, polluting the key points with unnecessary details that weaken their impact. However, the 30-page limit enforced on this report forced me to keep my writing concise and deliberate. This constraint helped develop skills regarding planning, prioritisation of content and clearer communication of my ideas all of which I'll carry forward to future long form writing tasks.

Another key lesson learned was how to conduct thorough academic research. In the beginning I found it difficult to engage with relevant academic sources. However, with completion of the project, I have become more comfortable with finding and understanding relevant research, evaluating the relevance and quality of papers, extracting key information from papers and building upon existing research.

Finally, the successful completion of such a long-term and technically challenging project has given me motivation and inspired me to revisit other personal projects I had previously set aside. The skills developed, along with the accomplishments achieved, over the course of this project have made me more confident in tackling challenging long-term projects.

## A.3 Legal, social, ethical and professional issues

### A.3.1 Legal issues

This project is based on the board game Quoridor, which is a registered trademark of Gigamic. No official assets or proprietary materials from the original board game were used. As the implementation was developed solely for educational purposes and was not distributed commercially or publicly released no legal issues are considered relevant here.

All libraries and external materials used in this project were used in accordance with their licenses and all external materials are appropriately credited in Appendix B.

### A.3.2 Social issues

The application developed during this project was used solely in a private, educational setting and not released to the public therefore presenting no social issues.

### A.3.3 Ethical issues

Ethical issues regarding user testing were considered during this project. Participants were presented with the information sheet in Appendix C which described the purpose of the project, the nature of their involvement and how any data collected would be used. Participation was entirely voluntary, and participants were informed they could withdraw from testing at any point. No personal information on participants was collected or stored.

### A.3.4 Professional issues

This project was completed as part of a British Computing Society (BCS) accredited degree and therefore, sound software development principles were used throughout. An agile methodology was adopted to ensure the projects delivery was on time. Version control was used adequately with a GitHub repository used to track and backup the projects code. Consistent naming conventions were used, and a focus was placed on code readability with comments used where necessary. Unit tests were used to validate the codes correctness and quality.

# Appendix B
# External Materials

- The Pygame library was used to deliver the graphical user interface and handle user inputs.
    - https://www.pygame.org
- The Pytest library was used as the framework to develop the unit tests to validate the software.
    - https://pytest.org
- The Numpy library was used for the numerical processing in the genetic algorithm.
    - https://numpy.org
- The Pathfinding library was used to apply pathfinding algorithms to the board.
    - https://pypi.org/project/pathfinding/
- Code from 2 tutorials was used at initial stages of development as a baseline before being heavily modified and restructured into the final project developed.
    - https://www.youtube.com/watch?v=vnd3RfeG3NM&list=PLzMcBGfZo4-lkJr3sqpikNyVzbNZLRiT3
    - https://www.youtube.com/watch?v=RjdrFHEgV2o&list=PLzMcBGfZo4-myY28wdQuJDBi8pCt-GIj6

# Appendix C
# User Testing Information Sheet

## Project Information Sheet

**Project Title**: *Computerised Quoridor*

You are being invited to take part in a student project. Before you decide, it is important for you to understand the aim of the project and what participation will involve. Please take time to read the following information carefully and discuss it with others if you wish. Ask if there is anything that is not clear or if you would like more information. Take time to decide whether or not you wish to take part. Thank you for reading this.

**Project Aim**:

The aim of this project is to create a computerised version of the board game Quoridor along with a strong artificial intelligent playing agent.

**Do I have to take part?**

It is up to you to decide whether or not to take part. If you do decide to take part you will be given this information sheet to keep and you can still withdraw at any time. You do not have to give a reason.

**What will happen to me if I take part?**

Should you take part in this study you will initially be told the rules of the game, and you will play as many games against myself as you like until you understand the game and the software. Then you will play as many games as you like against the AI. The results of these games will be recorded. After the testing is complete you will be asked to complete a questionnaire.

**Will my taking part in this project be kept confidential?**

No personal information will be collected during the testing.

**What type of information will be sought from me and why is the collection of this information relevant for achieving the project's objectives?**

Only results of the games against the AI along with your answers to the end of session questionnaire will be recorded. This information will be used to assess the strength of the AI agent developed in this study along with the extent to which the software achieves its objectives regarding user experience.

**What will happen to the results of the project?**

The results of this project will be published in a report to be submitted for assessment at the end of the undergraduate module COMP3931 Individual Project in the School of Computing at the University of Leeds.

**Contact for further information**: For further enquiries regarding this study, contact Matthew Bowler via email at sc22mb@leeds.ac.uk

If you decide to participate in this project, you will be given a copy of this information sheet. Thank you very much for taking the time to read this information sheet.