AI for playing classical strategy games - Quoridor

Ace Shyjan

registration: 100390438

1. Introduction

1.1. Quoridor Rules

Quoridor is a two-player abstract strategy game that is typically played on a *nine* by *nine* square grid. It feature simple actions where each player controls a single pawn. The primary objective is to navigate the player's respective pawn to reach the opponent's starting edge of the board before the opponent reaches theirs. On each turn, a player has two choices: to move their pawn or to place a fence, however these two simple choices can lead to advanced strategic plays.

1.1.1. Pawn Movement

Pawns can move one tile at a time, which can be in a forward or backwards direction, both horizontally and vertically. Pawns are typically not able to be moved diagonally; however, in different implementations, mechanics may vary. For example, in Glendenning et al. (2005), diagonal movement is allowed. When one pawn is positioned directly adjacent to the opponent's pawn, they are able to "jump" over their opponent's pawn on their turn, effectively moving two tiles in one go. This rule is not in effect if there is a fence positioned directly behind the opposing pawn.

1.1.2. Fence Placement

Both players have access to a limited supply of ten fences. Once placed on the board, these fences cannot be moved or picked back up. A fence spans two tiles and can be placed either horizontally or vertically; however, they cannot be positioned in a manner that creates an impossible route for either pawn. The fences can be used for many strategic plays, including causing your opponent to backtrack.

1.2. Development Process

1.2.1. Godot Engine

For this project, I will be working within the **Godot Engine**. It's an open-source community driven game engine with thorough support for 2D and 3D. It provide a lightweight interface for creating applications and supports two languages: **GDScript** and **C#**. Godot has been recognised more and more recently, even being the primary focus in project by Salmela (2022). Godot has some unique features such as its **Signal System** allowing scripts to send data from one to another without wasting resources as well as a **Debugger** which allows for the user to go through the current stack frame when debugging.

1.2.2. GDScript

I will be using GDScript to manage the general functionality of gameplay within the Godot Engine. This includes handling the user interface, the game of Quoridor for local-multiplayer as these are relatively lightweight algorithms. GDScript is an object-oriented language designed for Godot, offering seamless integration with the engine's core features. Built-in functions, such as node access, and their own classes have been optimised with GDScript in mind. It's syntax is quite similar to Python's, as it uses indentation to organise the code. In addition, it supports optional static typing of parameters as well as function return types which helps improves performance and reduces errors during development as it enables type hints for code competition and error checking.

1.2.3. C#

C# will be used for the primary algorithms associated with the AI components of the game, particularly for implementing game trees, which are essential for evaluating various possible game states. C# excels in heavy computational tasks, allowing for faster evaluations of leaf nodes, which is crucial for algorithms like the Minimax Algorithm that determine optimal moves. Its static typing enables compiler optimisations, while features like strong typing and generics can reduce memory overhead. One feature of the general game of Quoridor that won't be using GDScript is the illegal fence placement check, which will be using an algorithm such as a graph-traversal, which will be quite computationally expensive.

1.2.4. Other Tools

Aseprite is a lightweight graphical sprite editor that will be used for creating and modifying any sprites and textures used for the game's interface. I will be using GitHub for version control, which will help keep track of progress. It will also be useful as I move between my PC and laptop. Trello will be used to organize my tasks and meetings, as well as to store any potential ideas.

1.3. Objectives

For this project, my main aim is to create a functioning game of Quoridor in 2D. The user should be able to play against a computer. Some potential features include allowing the user to adjust their gameplay experience. Ideas in mind are adjusting the border size, computer difficulty, number of players, and adding multiplayer functionality. Some research will need to be conducted on suitable algorithms for the computer to use.

The primary aim of this project is to develop a functional 2D implementation of Quoridor, where the user can play against a computer opponent. Additional features may include options for users to customize their gameplay experience by adjusting variables such as the **board size**, the computer's **difficulty**, the **number of players**, and possibly incorporating **multiplayer** functionality. A key aspect of this project will involve researching and implementing suitable algorithms for the computer to effectively simulate intelligent gameplay as well as measuring their performance in game, as multiple players and a higher board size will increase the total calculation time.

2. Critical Review

This critical review will explore different aspects of creating Quoridor. The main focus of the research will be on AI algorithms that can effectively work with Quoridor's dual-objective nature of placing fences and moving the pawn. Some AI algorithms I will investigate include the Minimax Algorithm, Monte Carlo Tree Search (MCTS), and Genetic Algorithms. Another area that will be covered is pathfinding algorithms, which will be used for both the AI algorithms and the base Quoridor game, particularly for the illegal-fence placement checks.

2.1. Evaluation Functions

Evaluation functions are used to calculate a score to indicate how favourable an outcome is. In Mertens (2006), the evaluation function employed was simply the number of columns away the pawn was from the target line. Meanwhile, in Jose et al. (2022), the evaluation function used was the difference in distance for the pawns to their winning side, using a Breadth-First Search, which was found to be more effective than a random or greedy player. It was noted in Strong (2011) that one might expect the opponent to play a move that the evaluation function would score highly for (and low for us), and that the scoring function is unlikely to be perfect. This is because the opponent may use moves that appear desirable but could later prove advantageous. This is why a higher depth/ply is preferred, as there is more information available for predicting future game states. The issue with evaluation functions is their speed and precision. As noted in Brenner (2015), there is a trade-off between speed and accuracy in evaluation functions: a precise evaluation may be slow, while a faster one risks losing precision. So finding an evaluation function which is both fast, and doesn't alter the outcome is essential.

2.1.1. Manhattan Distance

The Manhattan distance is the sum of the absolute differences between the coordinates of two points. In Jose et al. (2022), it was used for their reward function due to its

efficiency over a Breadth-First Search, and this reward function also ignored fences. Similarly, in Glendenning et al. (2005), Manhattan Distance was implemented using two points—the pawn and the goal— and they too ignored any fences. The issue faced with this formula is that a common strategy can involve creating a route using fences, which forces a pawn to move back rather than forward.

2.2. Illegal Fence Placement Check and Pathfinding

For Quoridor, we need to check if the fence a player is about to place is illegal or not. To do this, we need to check if both pawns are able to travel from their current position to the target column if the fence were to be placed. There are different pathfinding algorithms available each with their own benefits and drawbacks.

2.2.1. Breadth-First Search

One method of checking for possible routes is to use a Breadth-First Search (BFS), which is a graph-traversal algorithm that uses a queue data structure. It works by visiting all nodes in a graph at the present depth before moving on to the nodes at the next depth level. In Respall et al. (2018), the authors used a BFS algorithm to simulate the effect of placing a fence in Quoridor. After simulating the fence placement, they ran a BFS to check that each pawn still had at least one valid path to reach its goal. This approach prevented scenarios where a pawn would be completely blocked from the target line. A BFS was also considered in the paper Korf (1999); however, the authors stated that the main drawback was its memory requirements. The memory required for a BFS is proportional to the number of nodes stored in the graph, which was a significant issue back in 1999. However, nowadays, this problem should not be as influential, as shown in Jose et al. (2022), where it was used as one of the evaluation functions for their genetic algorithm.

2.2.2. Depth-First Search and Iterative Deepening

Depth-First Search (DFS) is another graph-traversal algorithm that explores paths in a graph or tree by delving as deep as possible along each branch before backtracking. Unlike a BFS, which can be memory-intensive, DFS is typically more memory-efficient as it leverages a stack data structure, either explicitly or implicitly through recursion, to keep track of nodes. As described in Korf (1985), DFS "works by always generating a descendant of the most recently expanded node", prioritizing depth and exploring paths until it reaches the end of each branch. This approach makes DFS particularly well-suited for scenarios where memory usage is a concern. A key limitation of DFS, however, is that it is time-bound rather than space-bound, in contrast to BFS, as noted by Korf (1985).

Depth-First Iterative Deepening (DFID) is "a search algorithm which suffers neither the drawbacks of breadth nor depth-first search" - Korf (1985). DFID starts by performing a DFS with a depth level of one, discarding any nodes generated, and repeats this process until the goal state is reached. DFID expands all the nodes at the current depth before continuing with the next depth, thereby guaranteeing a solution of the shortest length. The disadvantage of this approach is that it performs extra computations before reaching its depth goal; however, analysis by Korf (1985) shows that these computations do not significantly affect the asymptotic growth of the run time for exponential trees. They noted that "almost all of the work is done at the deepest level of the search."

2.2.3. Dijkstra's Short Path Algorithm

This algorithm finds the shortest path between a start node and a destination node in a weighted graph. Each weight on the graph represents the cost of traversing from one node to another. It works by visiting all unvisited nodes, starting with the node with the lowest cost. It visits each of its neighbours and calculates the total distance through the current node and updates it, if the total distance is smaller than the current value. It repeats this process, until all nodes are visited. In Lawande et al. (2022), they said how this algorithm requires a lot of memory and that it cannot be used for values that have a negative cost. Whilst Quoridor will not be using negative cost values, high memory usage might be a potential issue in the future.

2.2.4. A-Star Algorithm

This algorithm is an extended and advanced version of Dijkstra's algorithm. As described by Glendenning et al. (2005), it "makes use of a **heuristic function** to predict the shortest path" by calculating the total cost as the sum of the exact cost from the start node to the current node and the estimated cost from the current node to the goal. The pathfinding process begins by adding the starting node to an open set and repeatedly selecting the node with the lowest total cost within the open set for exploration. Each selected node is then moved to a closed set, and its neighbouring nodes are evaluated, updating their total costs. The process stops when the target node is reached or the open set becomes empty. The A* algorithm is most optimal when the heuristic function is admissible, meaning the actual cost of reaching the goal is never overestimated. This implies that performance can be affected depending on the heuristic chosen.

2.3. Minimax Algorithm

A Minimax algorithm, such as covered in Plaat et al. (1996) and Strong (2011) finds the best next move for the current player. Its aim is to minimize the possible loss for a worst-case scenario whilst maximizing the player's own gain (hence *minimax*). The

minimax algorithm builds a **Game Tree** which is used to represent the current game state as the root node. Each node in the game tree represents a possible game state. For each edge/connection between a parent and a child means a possible move from one game state to a future game state. In Quoridor, the child node to any parent, would be the parent's opponent's turn.

The issue with a minimax algorithm for games like Quoridor and Chess, is that due to its complex nature, it has no terminal solution meaning a game tree would be too large and complex to compute in a reasonable time frame. There are some optimisations which are done to improve the algorithms performance and efficiency. In Jose et al. (2022), they made use of a limited depth of two for the tree. This mean they could calculate only one set of moves for the opponent and player. A low depth of two would be that the algorithm couldn't think far ahead into the future and play like a human, but it allowed for a starting point for future work.

2.4. Negamax Algorithm

This approach was also used by Glendenning et al. (2005) and Respall et al. (2018), who adapted the standard Minimax algorithm by utilizing only the maximization step, a variation known as *Negamax*. In Negamax, the algorithm is simplified by evaluating all moves from the perspective of a maximizing player. To achieve this, it recursively calculates values for each move and then negates them, effectively treating the opponent's optimal move as a minimization in the context of the maximizer's perspective. This structure means every node is evaluated as if it were from the "max" player's perspective, which reduces the need for separate min and max functions, streamlining the algorithm while preserving accuracy in competitive decision-making.

2.5. Alpha-Beta Pruning

Another optimisation technique that is commonly paired with minimax algorithms is the Alpha-Beta Pruning algorithm. This algorithm improves the efficiency of the game tree's search by eliminating branches of the game tree that are irrelevant to the final move selection. By maintaining two values—alpha, the maximum score for the maximizing player, and beta, the minimum score for the minimizing player — the algorithm can prune branches when it encounters nodes that fall outside these thresholds. This significantly reduces the number of evaluations needed, resulting in faster search times and improved AI responsiveness.

2.6. Genetic Algorithms

"Genetic Algorithms are a family of computational models inspired by evolution" - Mathew (2012). They can be used when the solution space is vast, such as for Quoridor and work by simulating the process of evolution by generating a population of possible solutions and evolving them over time to find the best/most optimal solution. The process begins by generating a population of potential solutions (**chromosomes**, which represents a different possible solution / move that the player can do. Each chromosome undergoes evaluation through a **fitness function** which states how well it meets a criteria - this reflects the chromosomes overall effectiveness.

2.6.1. Selection, Crossover and Mutation

In the selection stage, chromosomes with higher fitness scores (which are more likely to be chosen) are **selected** to be chosen for reproduction which mimics the natural selection process as the advantage traits are continued on through generations. After selection, **Crossover** occurs. This is where two parent chromosomes combine to create their offspring - this introduces a new combination of traits which allows for greater diversity among solutions. In addition, a mutation process can occur which adds random alterations to some chromosomes which allows the algorithm to explore more possibilities which crossover couldn't cover. These iterative processes of selection, crossover, and mutation are repeated over multiple generations, gradually evolving the population toward higher fitness scores. Each generation refines the pool of candidate solutions, increasing the chances of finding a near-optimal/optimal solution.

In the paper by Jose et al. (2022), they were exploring and comparing possible algorithms for Quoridor AI. They used a minimax algorithm as a baseline and one of the algorithms they looked into was a *Genetic Algorithm*. They adopted the minimax algorithm and established a set of heuristic situation evaluation functions, one for the player and opponent: Manhattan distance, Perpendicular Pawn Distance, Breadth-First Search (BFS), Dijkstra's Shortest Path. They found that the Genetic Algorithm was an effective algorithm as it could evolve potential problems over multiple generations. However, it struggled with more strategic plays, such as wall placements due to Quoridor's complexity.

2.7. Monte-Carlo Tree Search

A study by Respall et al. (2018) focused on the application of the Monte-Carlo Tree Search (MCTS) algorithm specifically for the game of Quoridor. MCTS operates by simulating random play-outs from various possible moves, selecting those that lead to the most successful outcomes. The algorithm constructs a search tree where nodes

represent different game states and edges correspond to the actions taken to transition between these states. MCTS consists of four key stages: **Selection**, **Expansion**, **Simulation**, and **Backpropagation**.

A study by Brenner (2015) explores the application of Monte Carlo Tree Search (MCTS) in comparison to the Minimax algorithm with Alpha-Beta Pruning for decision-making in Quoridor. The researchers found MCTS more suitable due to the game's asymmetrical game tree. This asymmetry, combined with Quoridor's high branching factor, makes traditional Minimax less effective for rapid AI decision-making without substantial pruning efforts. MCTS proved advantageous in this case, as it allows the AI to assess moves without needing a fully expanded game tree.

The paper by Respall et al. (2018) highlights that, given Quoridor's dual-objective nature—where players must both advance their pawn and strategically place walls—the MCTS approach offers a flexible way to navigate its complex decision space. However, in the paper they noted that further improvements in simulation strategies and overall efficiency are necessary in order to achieve optimal performance, suggesting that while MCTS is promising, there remains significant room for refinement in its application to Quoridor and similar games. One problem they did encounter during implementation, was with their MCTS Class. The class itself contains a "huge amount of information and data structures", and which they found that after 5000 simulations their program had run out of data. Their solution for this was storing the move to perform rather than saving the generated board in each node instead.

2.7.1. Selection

In the selection phase, the algorithm traverses the tree, starting from the root node and moving toward the leaf nodes. During this traversal, it balances exploration and exploitation; exploration involves examining less-frequent paths to discover potentially better outcomes, while exploitation focuses on paths known to yield favourable results.

2.7.2. Expansion

Once a leaf node is reached — indicating a point where further exploration is possible, one or more child nodes are added to the tree, provided the current state is not terminal (i.e., the game is not over). This phase increases the number of potential future game states that can be analysed, allowing for a broader range of possible outcomes to be considered in subsequent simulations.

2.7.3. Simulation

In the simulation stage, the algorithm conducts a random play-out from the newly added node(s), simulating the remainder of the game until a terminal state is reached. This involves making a series of random moves, which can help gauge the viability of the current game state. The randomness in this stage allows MCTS to explore a variety of game scenarios.

2.7.4. Backpropagration

Following the simulation, the algorithm enters the backpropagation phase, where it updates the nodes along the path taken during the traversal, all the way back up to the root node. Each node in this path has its visit count incremented, and its total reward is adjusted based on the outcome of the simulation — whether it resulted in a win, loss, or draw. This feedback mechanism helps the algorithm refine its strategy over time, as nodes representing with higher visit counts will receive more attention. This means that MCTS does not "need any specific evaluation function" as stated in Brenner (2015).

2.8. Summary

Across various reports, the minimax algorithm frequently served as a baseline for comparing AI methods in Quoridor. MCTS demonstrated the most promising results from the studies that compared it with Minimax, largely due to its adaptability in handling Quoridor's dual-objective gameplay of both advancing pawns and strategically placing walls. The flexibility of MCTS made it well suited to exploring diverse strategies, offering an advantage over more rigid approaches. In contrast, the Genetic Algorithm struggled with Quoridor's complexity, particularly in developing strategic play and optimal wall placements, as it lacked the strategic planning that a human is able to do. Despite this, implementing the Genetic Algorithm and comparing it could offer some valuable insights and potentially introduce some interesting dynamics.

Most evaluation functions used were fairly simple, being the number of tiles / columns away as this function runs for each leaf node in a search tree. I found that between the pathfinding algorithms, whilst A-Star being the most popular and reliable path-finding algorithm and BFS being quite popular, DFID could have been a good contender, if the games were memory bound, however nowadays that isn't the case, hence why it was used in Jose et al. (2022).

References

Brenner, M. (2015). Artificial intelligence for quoridor board game.

Glendenning, L. et al. (2005). Mastering quoridor. *Bachelor Thesis, Department of Computer Science, The University of New Mexico*.

Jose, C., Kulshrestha, S., Ling, C., Liu, X., and Moskowitz, B. (2022). Quoridor-ai. *ResearchGate*.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

Korf, R. E. (1999). Artificial intelligence search algorithms.

Lawande, S. R., Jasmine, G., Anbarasi, J., and Izhar, L. I. (2022). A systematic review and analysis of intelligence-based pathfinding algorithms in the field of video games. *Applied Sciences*, 12(11):5499.

Mathew, T. V. (2012). Genetic algorithm. *Report submitted at IIT Bombay*, 53.

Mertens, P. J. (2006). A quoridor-playing agent. *Bachelor Thesis, Department of Knowledge Engineering, Maastricht University*.

Plaat, A., Schaeffer, J., Pijls, W., and De Bruin, A. (1996). Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(1-2):255–293.

Respall, V. M., Brown, J. A., and Aslam, H. (2018). Monte carlo tree search for quoridor. In *19th International Conference on Intelligent Games and Simulation, GAME-ON*, pages 5–9.

Salmela, T. (2022). Game development using the open-source godot game engine.

Strong, G. (2011). The minimax algorithm. *Trinity College Dublin*.

A. Risk Analysis

Risk	**Likelihood**	**Impact**	**Mitigation Plan**
Challenges in AI decision-making (dual objectives)	Medium	High	Focus on early AI testing with small-scale prototypes, adjust complexity gradually, and implement simpler fallback logic for AI if needed.
Difficulty in generating effective AI opponents	Low	Medium	Use pre-tested libraries or frameworks for Genetic Algorithm, run extensive test cases to ensure quality of opponents, reduce scope if too much to handle.
Loss of data or project files	Low	High	Use GitHub regularly
Bugs with Godot Engine	Medium	Low	Find a workaround or edit the engine as it is open source
Scope Creep	High	Low	Prioritise each feature and ensure the most important features are added first.

B. Gantt Chart for Quoridor Development

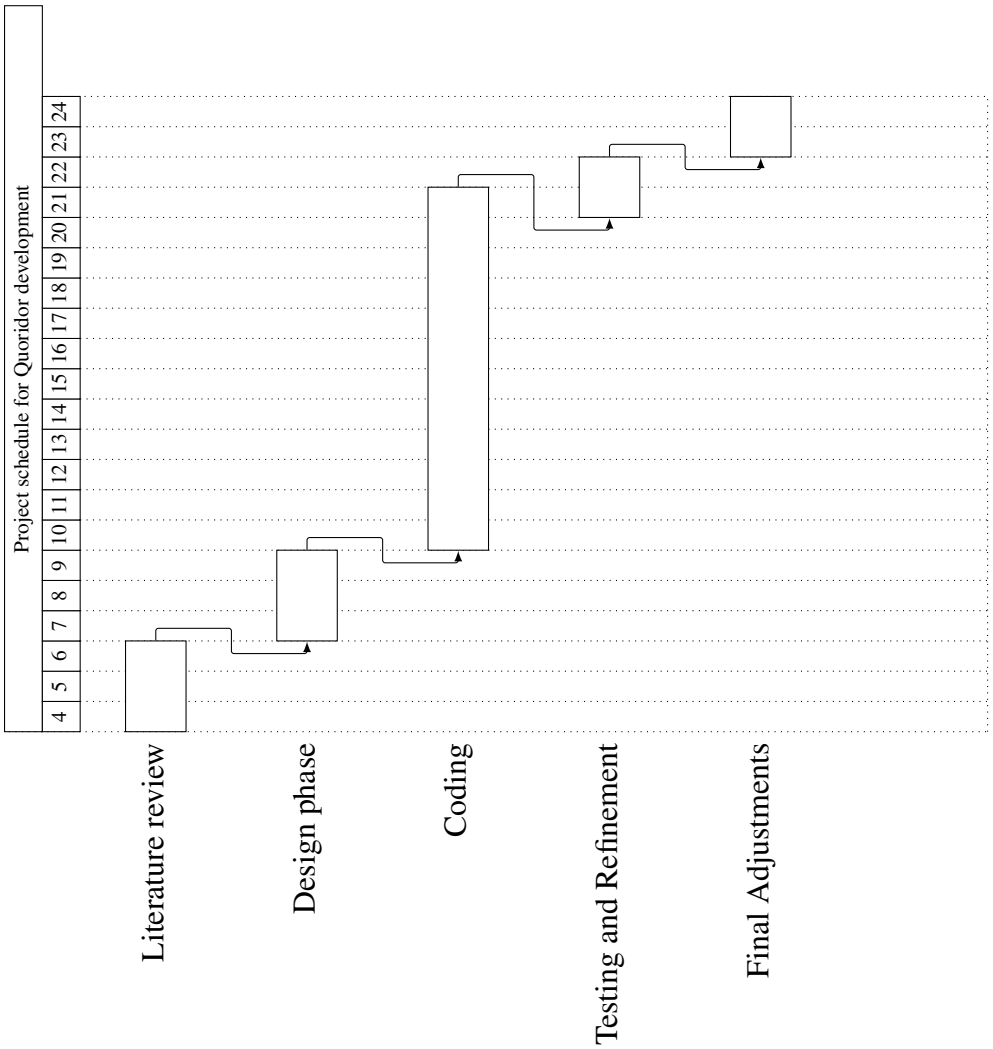

Figure 1: Quoridor Development Gantt Chart

CMP-6013Y **Student registration number: 100390438**

Literature review

Introduction: brief description of project, aims and objectives, areas of knowledge required.	First	2.1	2.2	3	Fail
Discovery of suitable quantity and quality of material.	First	2.1	2.2	3	Fail
Description of key issues and themes relevant to the project.	First	2.1	2.2	3	Fail
Evaluation, analysis and critical review.	First	2.1	2.2	3	Fail

Quality of writing

Clarity, structure and correctness of writing	First	2.1	2.2	3	Fail
References correctly presented, complete adequate (but no excessive) citations.	First	2.1	2.2	3	Fail

Planning

Risks: identification, suitable contingency planning.	First	2.1	2.2	3	Fail
Measurable objectives : appropriate, realistic, timely.	First	2.1	2.2	3	Fail
Gantt chart: legibility, clarity, feasibility of schedule.	First	2.1	2.2	3	Fail

Comments

Supervisor: Dr Michal Mackiewicz

Markers should circle the appropriate level of performance in each section.