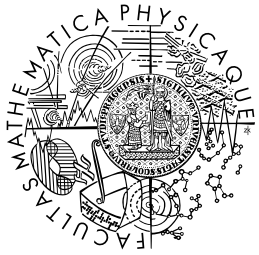


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Devyanshu Koirala

Artificial Intelligence for Quoridor game

Department of Software and Computer Science Education (KSVI)

Supervisor of the bachelor thesis: Mgr. Klára Pešková, Ph.D.

Study programme: Computer Science

Study branch: Artificial Intelligence (AI)

Prague 2024

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

Dedication.

Title: Artificial Intelligence for Quoridor game

Author: Devyanshu Koirala

Department: Department of Software and Computer Science Education (KSVI)

Supervisor: Mgr. Klára Pešková, Ph.D., KSVI

Abstract: Quoridor presents a challenging terrain for strategic decision-making, making it a suitable testing ground for various Artificial Intelligence (AI) algorithms. This thesis explores the implementation of generic interfaces for AI agents and finally evaluation the AI agents in the game, namely Minimax, Monte Carlo Tree Search (MCTS) and A-star within the realm of Quoridor gameplay. In the thesis, we develop the AI interface that can be easily integrated into any other game.

The research begins with a comprehensive overview of the Quoridor game, its rules and strategies. Subsequently, we delve into the theoretical foundations and practical implementation details of the aforementioned AI algorithms and conduct a thorough evaluation in an effort to determine the best one in this context.

Keywords: Quoridor Artificial Intelligence AI Board Game

Contents

Chapter 1

Introduction

In recent years, Artificial Intelligence (AI) is becoming an integral part of many elements of modern world including gaming [Skinner and Walmsley, 2019], pushing the boundaries of what's achievable in both single and multiplayer gaming experiences. AI-driven games now offer users the opportunity to hone and enhance their skills, providing varying difficulty levels and offering optimal moves to guide players through each step if desired. AI has also taken a center stage in gaming with its remarkable accomplishments in age-old strategy games such as Chess, Go, and many others.

Strategy games are a genre of gaming that require planning, often involving various tactics, decision making and execution under various resource constraints. Some examples of strategy games include Shogi, Starcraft and Quoridor. They are unique compared to other genres as they require a selection of an optimal move among multiple possible moves based on a certain strategy. In many scenarios, the number of possible moves depends on the game tree, simulation and prediction of the player's and the opponent's moves, all while managing resources efficiently.

AI, due to its suitability of solving complex decision making problems factoring in multiple variable and constraints, has been particularly effective in playing the strategy games. The history of AI in strategy gaming dates few decades. One of the oldest marked impact of AI in the strategy gaming came in 1997 when IBM's Deep Blue [Campbell et al., 2002] defeated World Chess Champion Garry Kasparov. The influence was more prominent with the success of AI on real-time strategy (RTS) games such as Warcraft and StarCraft [Robertson and Watson, 2014] and strategy games such as Go [Huang et al., 2011]. Recently, DeepMind's Alpha Go for Go, Alpha Zero for Chess [Silver et al., 2017] and AlphaStar for StarCraft [Team, 2019] have widened the gap between the AI and human intelligence even further.

In this thesis, we have chosen Quoridor as the strategy game to implement our interfaces on, and analyze results. Designed by Mirko Marchesi, Quoridor stands out as an engaging strategic board game that is played between two or four players. The game is played on a square grid board where the objective of this game is for each player to move their pawn to the opposite side of the board. This game introduces a fascinating twist where a player, in addition to trying to move their pawn through the square grid, additionally has an option to place walls on the grid locations strategically to obstruct the opponent's path. This strategy compels the player to think of their traversal strategy while predicting

the opponents strategy as well. Despite its seemingly simple rules, Quoridor demands a unique blend of strategic foresight and the ability to anticipate the moves of opponents and outmaneuver the opponent.

The objective of this thesis is to construct a well-structured framework and user-friendly interfaces that seamlessly integrates AI algorithms into the Quoridor game. The adaptation of our generic interfaces on Quoridor's rule-set will not only enhance our understanding of the game's intricate nuances but also facilitate the creation of an intuitive interface for simulating these AI agents. Furthermore, a comprehensive evaluation will be conducted to identify the suitable parameters for different agents, and several head-to-head games between these agents will help us identify the top-performing one from the set of implemented agents including the Minimax agent, Monte-Carlo tree search (MCTS) agent and the A-star agent. In addition to this, the project will encompass the creation of a user-friendly interface that empowers players to engage with an AI opponent of their choice visually, thereby bolstering the game's accessibility and inclusivity.

The structure of this thesis is as follows. In Chapter 2, we will introduce the formal notations of the game and based on it, formally explain the rules of the game. In Chapter 4, we will perform a game analysis from the perspective of game complexity including the state-space complexity and the game tree complexity. In Chapter 5, we will explain the background and the algorithm behind the implemented AI agents. In Chapter 6, we will explain the implementation of the game interface and the AI agents, and provide examples on how one can integrate the said agents in different games. In Chapter 7, we will simulate the results of the game between the implemented AI agents and finally conclude the thesis in Chapter 8.

Chapter 2

Quoridor: Game description

Quoridor is a 2, or 4 player game played in an NxN chess-like board where each player has a pawn and a set number of walls, where, typically the dimension of N is an odd number with 9x9 board being the most popular. In the game, each square represents a potential position for the pawn pieces. The board also contains grooves bordering each squares on the board where the player can place the walls. With 9x9 board, the total number of walls available to the player is 10.

The game starts with the pawn pieces of the players on the opposite side of the NxN square board. The starting position of the pawn is the center of the edge of the board associated with the side of the player. Each player, additionally, can place a wall on the grooves of the board between any two squares. The main objective is to be the first player to traverse with the pawn through the board to any one of the N squares on the opposite edge of the board. A player, in their each move, can either move a pawn or place a wall. The idea is that placing the wall between the squares means that the opponent cannot pass through the squares and potentially many need to take a longer route to reach the opposite edge of the board. Hence, the strategy of a player in the game is to be the first player to move their pawn to the row of squares on the opposite edge of the game board avoiding any walls deterring its path to the goal while strategically placing walls to deter opponents from reaching their goal squares.

The Quoridor game is played in turns between the opponents where in each turn the player can either move a pawn or place a wall. Then the move shifts to the opponent player(s) where they can do the same in their turn.

Walls are a fundamental element of the game, allowing players to strategically block their opponent's path and influence the course of the game. Each wall spans across two squares on the board and occupies exactly four squares either horizontally or vertically, effectively creating a barrier between them. At the beginning, each player starts with a set number of walls that they can use during their turn.

As seen in Figure 2.1, in this thesis, we consider a two-player Quoridor game with player A and player B. The pawn of player A is represented by the letter 'A' and that of player B is represented by the letter 'B'. In the figure, we display a 5x5 board with the squares on the edge of the board highlighted in pink belonging to player B whereas the squares on the opposite edge of the board belonging to player A. The board in the figure also shows the starting position of the two

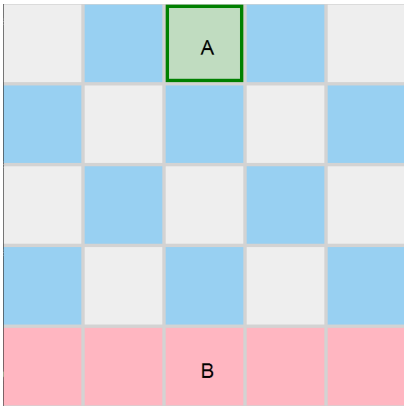


Figure 2.1: A screenshot of a 5x5 game board from our implementation

players with the pawn at the center of their respective edges. The goal of player A is to move its pawn through the board to one of the squares on the edge belonging to player B that has been highlighted in pink. The gray areas between the squares on the board represents the grooves of the board where walls can be placed. As mentioned earlier, the length of the walls is equal to the length of two squares. Hence, placing a wall on the board touches four squares where two squares are on one side of the wall whereas the two other squares are on the opposite side of the wall. The player cannot cross the squares through the wall.

In order to formalize the game, we have to define the notations and use it to formally define the game rules and the player movement and wall placement rules.

2.1 Notation

In this subsection, we will first consider the notations popular in the literature and the Quoridor community and then define our own notation, especially for the wall placement, while analyzing the differences between them. This definition of our own notation is motivation by the ambiguity that we present existing in the current notation.

There are no official notations for this game. However, some popular ones recognized by the Quoridor community include **Glendenning's Notation** ([Lisa Glendenning, 2002]) and the **Quoridor Strat's Notation** ([Quoridor Strats, 2014]). We first consider the notations in these two references below:

First, defining the notation of the squares of the board, consider a Quoridor board of dimension 9x9 as an example. Let $\mathbb{K} = \{a, ..., i\}$ and $\mathbb{R} = \{1, ..., 9\}$ and C denote the 2D Quoridor board with $C_{i,j}$ representing the i-th row and j-th column position of the cell. Both the Glendenning's and Quoridor Strat's notations follow the same principle of labelling each cell or a square by $C_{i,j}$ where $i \in \mathbb{R}$ and $j \in \mathbb{K}$. Hence, in this approach the board is labelled by a combination of alphabetical letters (e.g., $\mathbb{K}$) denoting the columns and the positive integers (e.g., $\mathbb{R}$) representing the rows. This follows the similar style of notation for labelling the squares in chess.

This notation can be extended to a Quoridor board of any dimension NxN by considering the dimension of both $\mathbb{K}$ and $\mathbb{R}$ as N where $\mathbb{K}$ is the set of first N

alphabetical characters in an ascending order and $\mathbb{R}$ is the set of first N positive integers in a ascending order.

Considering an example in Figure 2.1, with these notations, the pawn A is in the position $C_{1,c}$ and the pawn B is in the position $C_{5,c}$.

After representing the Quoridor board, the next step for defining notations for the game is to define the moves. Let us represent the move for a player as M.

$$M(C_{ij}) = ji, \text{ where, } j \in \mathbb{K} \text{ and, } i \in \mathbb{R} \tag{2.1}$$

Unlike in chess, in Quoridor, there is only one pawn for each side. Hence, for the side, for moving the pawn, the starting position of the pawn is not required. Hence, as mention in Equation (2.1), the movement of the pawn to the cell $C_{j,i}$ can simply be defined by the notation ji. The movement of the pawn A, in Figure 2.1 from its position from cell $C_{1,c}$ to the position of B in $C_{5,c}$ through the cells $C_{2,c}$, $C_{3,c}$ and $C_{4,c}$ can be defined with moves $c2$, $c3$, $c4$ and $c4$ requiring a total of 4 moves.

Additionally, as described earlier, the Quoridor game is played between multiple players where each player in a turn can make a move. Hence, the turns are represented numerically, for e.g., 1., 2., 3. represents the first, second and third move respectively. Hence considering the scenario where player A moves its pawn in Figure 2.1 from $C_{1,c}$ to $C_{2,c}$ and then to $C_{3,c}$ and player B moves its pawn from $C_{5,c}$ to $C_{4,c}$ to $C_{4,b}$, the notations can be represented as: $1.C2C4$, $2.C3B4$. Here the numbers 1. and 2. represents the two moves of each players and $1.C2c4$ represents the movement of the players A's pawn to C2 and then player B's pawn to C4. The order of the player's movement can be dependent on the player that moves first. For example, in the example, player A moves first to $C4$, hence it is notated before player B's move. This order has to remain constant throughout the game.

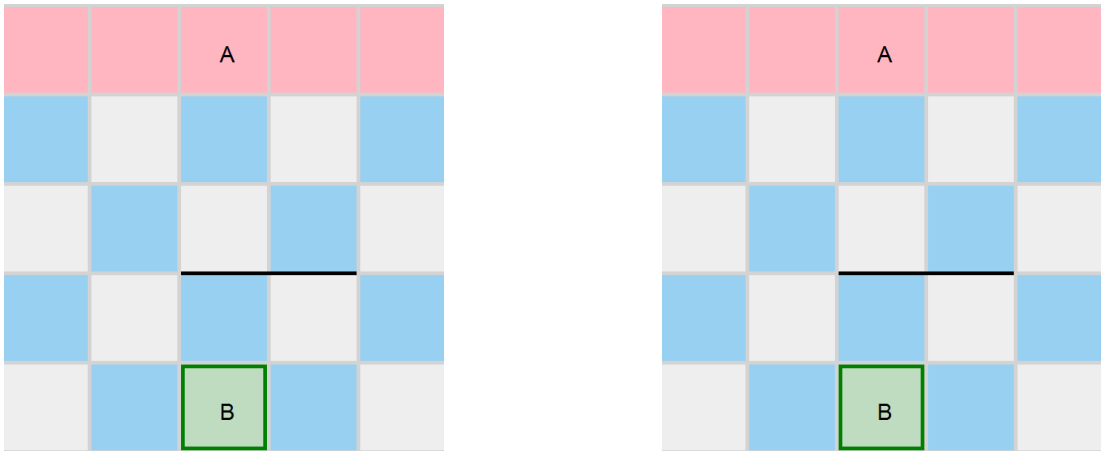
Similarly, after notation of the board and the movement, the notation for wall placement is another important component for Quoridor. The notation for wall placement is defined by the following equations:

$$W(C_{i,j}) = jiD \text{ , where, } j \in \mathbb{K}, i \in \mathbb{R} \text{ , and } D \in \{h, v\} \tag{2.2}$$

In the above equations, the wall is represented with respect to a reference cell. In the above equation, the reference cell is $C_{i,j}$. As described earlier, the length of the wall is equivalent to the length of two cells. Hence, the reference point of the cell determines the starting point of the wall placement that covers two cell lengths. Likewise, D in Equation (2.2) defines the arrangement of the wall. The wall from a reference point can be placed vertically or horizontally. This is represented in the equation by the characters 'h' and 'v' respectively.

In the notations defined as per Glendenning's notations and Quoridor Strat's notations, there is a difference when it comes to the wall representation even though the mathematical formulation is the same as presented in Equation (2.2).

As seen in Figure 2.2, the difference lies in the exact position of the reference point of the walls of the wall. In the reference square, there are 4 corners. The reference point for horizontal wall placement is always the left corners of the cell. However, the difference in the notations lie in fact that whether the reference left corner is the left upper corner or the left lower corner. In the Quoridor Strats



(a) Glendenning's Notation: **c3h** (b) Quoridor Strats Notation: **c4h**

Figure 2.2: Notation differences

Notation, the reference starting point of the wall is defined by the lower-left corner of the sqaure whereas in the Glendenning's Notation, each wall starts from the upper-left corner of the square. This difference is exemplified in the figure 2.2. In Glendenning's Notation, the wall placement with reference to the square $C_{3,c}$ i.e., c3h is defined with respect to the lower left corer of the square $C_{3.c}$. In contrary, in the Quoridor Strats Notation, the wall placement with reference to the $C_{4,c}$ i.e., c4h, is defined with respect to the upper left corner of the cell $C_{4,c}$. Since the lower left corner of the cell $C_{3,c}$ and the upper left corner of the cell $C_{4,c}$ are the same, the wall placement due to the notation difference were the same as seen in the figure.

Even though there notations are widely used, they are very easy to get confused with since they have the same wall representations, and unless specified explicitly, it is difficult to tell which representation is being used. This ambiguity in notation necessitates a new notations, in particular for the wall placement to remove any confusion. For this purpose, in this thesis, we introduce a new notation for the purpose, particularly, of wall placement representation as follows:

$$W(C_{ij}) = jiD \text{ , where, } i \in \mathbb{R}, j \in \mathbb{K} \text{ ,and, } D \in \{N, S, E, W\} \tag{2.3}$$

In this new notation, we increase the size of the set D from horizontal and vertical representation to north, south, east and west representations indicated by the characters 'N', 'S', 'E' and 'W'. This ensures that instead of an ambiguous vertical and horizontal wall representation with respect to a cell, we can now define the direction explicitly. This wall additionally also implies that the wall in the 'N' and 'S' direction covers the reference cell, i.e., $C_{i,j}$ and the cell right to the reference cell, i.e., $C_{i,j+1}$ where $j+1$-th column represents the column on the right of the j-th column. Similarly, the wall in the 'E' and 'W' directions implies that the wall covers the reference cell $C_{i,j}$ and the cell below the reference cell, i.e., $C_{i+1,j}$ where $i+1$-row represents the row below the i-th row.

Looking back at Figure 2.2a, the walls can now be represented by **c4N**, i.e. a Northern wall from the cell $C_{4,c}$.

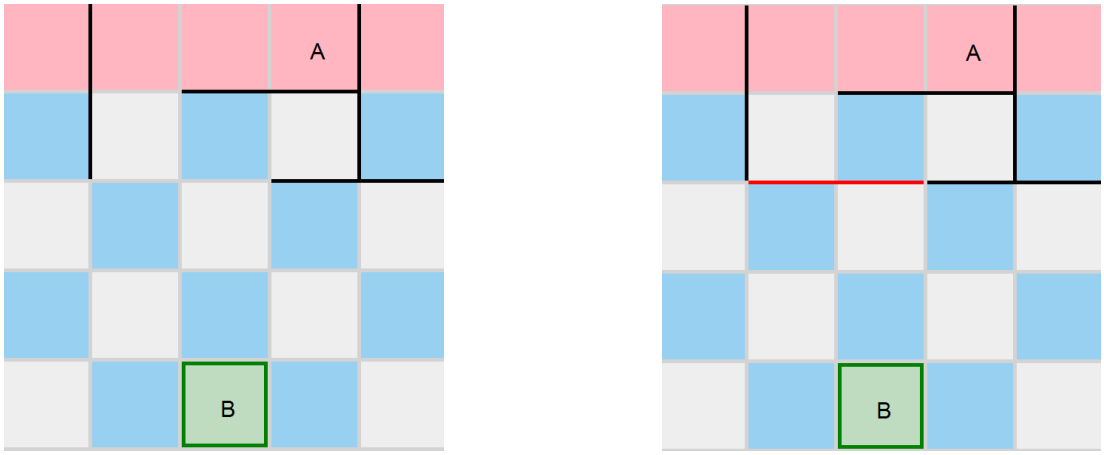
This notation provides an additional flexibility to represent the wall placement in the game and removes the ambiguity that may be present as we saw earlier.

2.2 Rules

In this section, we formally define the rules of the game Quoridor, in particular for the wall placement and player movement.

2.2.1 Wall placement rules

- The walls have to be placed in either vertical or horizontal manner and they cannot be placed diagonally.
- A placed wall must not completely block any player's path to victory. Each player must have at least one path to victory. For example, in Figure 2.3a, the walls are placed in such a way that player A has a valid path towards the edge of the player B, in particular, to the cells $C_{5,a}$, $C_{5,b}$, $C_{5,c}$, $C_{5,d}$ and $C_{5,e}$. However, the wall placement in Figure 2.3, due to the placement of the wall **c3N**, the player A does not have a valid path towards the edge cells of player B. Hence, the move **c3N** would be classified as an invalid move.
- A placed wall cannot intersect any of the previously placed walls. For example, in Figure 2.3a, a walls has been placed with the moves **a1W**. A subsequent move **a2N** would be an invalid move in the game due as it requires the walls to intersect.
- Walls cannot be placed along the edges of the board. Walls must be placed to create a barrier for exactly 4 cells. For example, in Figure 2.3a, a wall cannot be placed with the move **a1E** as it would only touch 2 cells $C_{1,a}$ and $C_{2,a}$ as it is on the edge of the board.
- Every player possesses a limited supply of walls, and once they exhaust these walls, they are unable to place any additional ones. Consequently, in such situation the player is only allowed to maneuver their pawn on the board.



(a) Valid game state

(b) Invalid wall **b2S**

Figure 2.3: Example of an invalid wall placement

The game state represented by Figure 2.3a shows the situation after 5 turns, with it currently being player B's turn to move. Since both **A** and **B** have viable paths to their respective goal rows and all walls have been placed according to the rules (see *Section 2.2.1*), the game state shown in Figure 2.3a is considered valid.

However, player B disrupts the rules by placing the red wall, violating the specified wall-placement rules (see *Section 2.2.1*), consequently rendering the game state represented by Figure 2.3b invalid.

2.2.2 Player movement rules

- Players are allowed to move their pawn one cell at a time in the North, South, East, or West directions during their turn given that there is a cell and the cell is empty in the direction. Diagonal movements are not allowed. For example, a pawn in a cell $C_{2,c}$ can move to either in the north direction (i.e., $C_{1,c}$), the south direction (i.e., $C_{3,c}$), the west direction (i.e., $C_{2,d}$) or in the east direction $C_{2,b}$ given that the adjacent cells or squares empty. However, a pawn in the cell $C_{1,a}$ can only move in the west (i.e., $C_{1,b}$) and the south (i.e., $C_{2,a}$) direction as there are no squares on the east or the north of the cell.

- **Jump**

  - If an opponent is at to the cell a player intends to move in the same direction of the opponent, the player can jump over the opponent provided there is no wall between the opponent or behind the opponent they intend to jump over. For example, in the first picture in figure 2.4 the player B can move to either of the squared highlighted in green. The player can move to east, west or south direction or can jump in the north direction over the player A from square $C_{4,c}$ to the square $C_{2,c}$ as there are no walls between the squares $C_{4,c}$, $C_{3,c}$ or $C_{2,c}$.
  - If there is a wall between the opponent and the jumping square, the player can jump to a cell on either side of the opponent's cell, given the cell is accessible from the opponent's cell (i.e., there are no walls). In the section picture from left in the figure, for example, the player B cannot jump over player A to the cell $C_{2,c}$ due to the wall placement **c3N**. The player in this case can jump over to the cells on either the east side (i.e., cell $C_{3,b}$) or the west side (i.e., cell $C_{3,d}$) of the player A given there are no walls in between $C_{3,c}$ and $C_{3,b}$, and $C_{3,c}$ and $C_{3,d}$ respectively.
  - Players cannot jump over walls. If a player is jumping from a cell $C_{1,*}$ to cell $C_{3,*}$, this is only possible if there are no walls in between them.

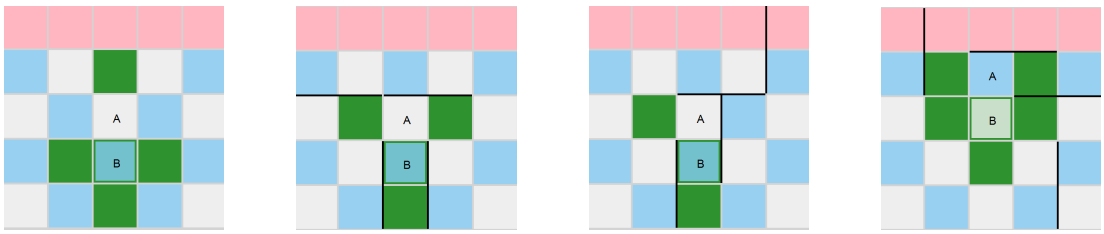


Figure 2.4: Examples of possible moves (marked green) for player B in different game states

Chapter 3

Related Works

Compared to some other strategy games such as Chess and Go, Quoridor has not been extensively studied in the literature. The authors in [Brenner, 2015] developed an MCTS approach for Quoridor. Recently, in [Iwanaga and Sakamoto, 2022], the authors performed an analysis of the game for a miniature 5 by 5 board.

For this thesis, we consider the following works as our inspiration:

- **Mastering Quoridor [Lisa Glendenning, 2002]**
  The author of the thesis paper and assesses various algorithms like Negamax, Alpha-beta Negamax among others. Additionally, they utilized a genetic algorithm to refine the weights within a linear weighted evaluation function, employing 10 distinct features suggested by the author, some of which include player's position towards their goal side, the opponent's position towards their respective goal, the remaining count of walls available to the player, etc.

- **A Quoridor-playing Agent [Mertens, 2006]**
  The author of this paper delves deep into the theoretical aspects of Quoridor, providing an upper-bound on the state-space and the game-tree complexities, which we use as a foundation in Chapter 4. Furthermore, they also develop a Quoridor playing agent based on the Minimax algorithm.

In sharp contrast to the aforementioned works, our thesis takes a distinctly different path by delving deeply into the architectural aspects of AI. Our approach emphasizes abstraction to the greatest extent possible, with an eye on facilitating seamless integration into a broad spectrum of games. We prioritize creating an interface that is adaptable to diverse game environments, setting our research apart from the game-specific focus of the prior works.

Chapter 4

Game Analysis

In Chapter 2, we explored the rules and gameplay mechanics of Quoridor. As we progress, this chapter aims to deepen our understanding by analyzing Quoridor from a theoretical and computational perspective.

In this chapter, we will classify Quoridor within the realm of strategic games, examine its game tree, state-space and tree complexity, and explore the implications of these factors on gameplay and artificial intelligence application.

4.1 Classification of Quoridor

Quoridor can be characterized as a discrete, deterministic, zero-sum, sequential, game with perfect information [Lisa Glendenning, 2002], and therefore, a combinatorial game [Thomas S Ferguson, 2020].

4.1.1 Discrete

In every turn of the game, each player has a finite number of moves and wall placements. These are limited by the game state (already placed walls and moved pawns) and the rules of the game. The game-tree of Quoridor has finite number of nodes (e.g Figure 4.1).

4.1.2 Deterministic

Quoridor has no random elements or chance involved in the gameplay. Every outcome and situation is a direct result of the players' decisions and strategies. There's no dice rolling, card drawing, or any other mechanism that introduces randomness. Hence, this is a deterministic game.

4.1.3 Zero-sum

In Quoridor, when a player makes a move that brings them closer to winning (like advancing their pawn or placing a wall effectively), it inherently puts the opponent at a disadvantage. Therefore, any positive progress for a player translates into a negative impact for their opponent. This reciprocal relationship of gain and loss between the players is what characterizes Quoridor as a **zero-sum** game.

4.1.4 Perfect Information

Every aspect of its gameplay are completely visible and known to all players at all times. This means that the positions of the pawns and the placements of the walls on the board are always in full view, allowing players to make strategic decisions based on the entire state of the game. Hence, this property classifies the game as a game of perfect information.

4.2 Game-Tree

A game tree for an abstract-strategy game (discrete games with perfect information) is a comprehensive graph representing every possible game states and sequence of moves. The nodes of a game tree represent game states, and the edges represent actions/moves.

Game trees are integral to the framework of adversarial search problems, where they are employed to systematically explore and evaluate the possible outcomes of different strategies, and forecast future states of the game based on current and potential moves.

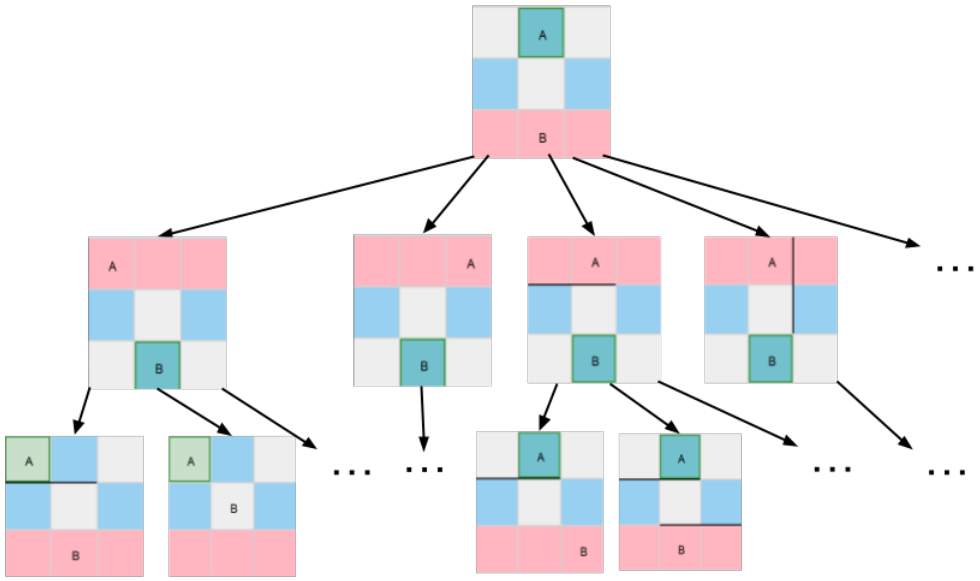


Figure 4.1: A partial game tree for a 3x3 game board.

As depicted in Figure 4.1, the root node of the game tree consists of player A located in the cell $C_{1,b}$ making a choice for the first move which may be one of pawn movement or wall placement. The nodes at a depth of 1 represent all possible game states as a result of moves made by player A and so on.

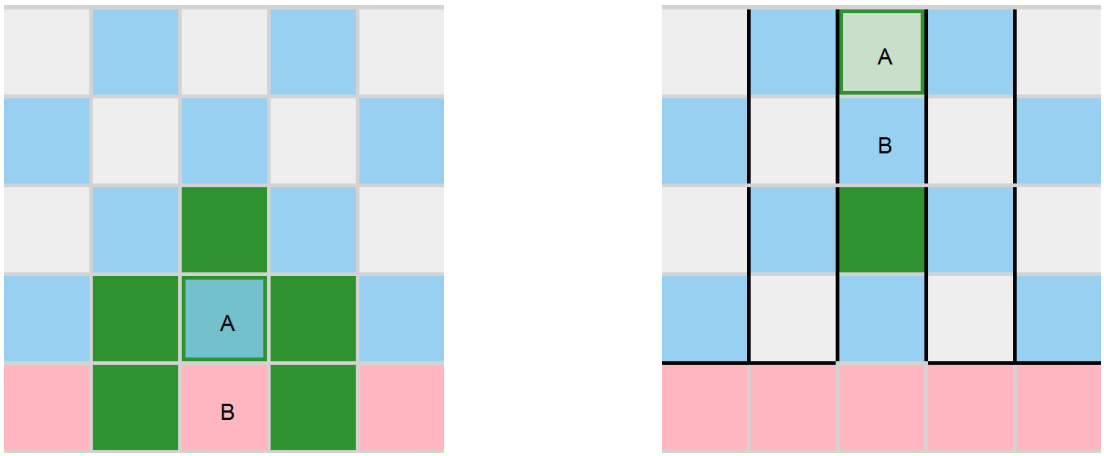
4.2.1 Branching Factor

The branching factor of a Game-tree is the number of child nodes of each node, or in other words, the number of possible moves a player at their turn can make, given the game state.

In Figure 4.1, player A makes the first move. A has **3** places to move their pawn to and **8** places to put one of their walls at. So, the root node has a branching factor of **11**.

The branching factor is greatly influenced by the state of the board in Quoridor, i.e the location of the pawn of the player, the location of the pawn of the opponent and the walls placed on the board.

As an example, the figure below represents a game states with the maximum and minimum branching factors:



(a) Maximum : 37 (b) Minimum : 1

Figure 4.2: Branching factor differences

As depicted by Figure 4.2a, player A has 5 possible places to move their pawn to. No walls have been placed so far, so player A can also place one of their walls in any groove.

For a board of size NxN with no walls placed, $N-1$ walls can be placed in each row (since each wall occupies 2 cell lengths), and there are $N-1$ rows for correct horizontal wall placements. Hence, there are $(N-1)^2$ slots for horizontal wall placements, and since the board is NxN, the total slots for both horizontal and vertical wall placements is given by the equation:

$$2(N-1)^2 \tag{4.1}$$

Coming back to Figure 4.2a, since the board has no walls placed, we now see that A has $5 + 2(5-1)^2 = 37$ possible moves they can perform, ergo, the branching factor of the game state represented by Figure 4.2a is **37**, which is also the maximum branching factor for board sized 5x5.

However, in Figure 4.2b, player A has no available slot for wall-placement, and the already-placed walls block A from moving anywhere except for cell $C_{3,c}$. Hence, the branching factor for the game state represented by Figure 4.2b is **1**.

Average Branching Factor

In Sub-Section 4.2.1, we saw that the branching factor is not uniform due to factors like wall-placements and positioning of players greatly influencing it.

We, therefore, would like to estimate an average branching factor for boards of different dimensions to see if varying board dimension has any effect in the average branching factor.

We already know from Equation 4.1 that the maximum branching factor of the game tree is exponential in order of N and from Figure 4.2b, we can deduce that the minimum branching factor is 1 (since we can replicate a similar game state for any dimension).

To find an estimate of the average branching factor B_{avg}, we propose Algorithm 1, which runs N simulations between 2 agents, keeping a track of the total game states encountered and the total moves made by agents, and averaging their values.

Algorithm 1: Average branching factor

```
Function AvgBranchingFactor(agent1, agent2, simulations):
    input  : Two agents and number of simulations
    output: Average branching factor
    SumOfAverages ← 0
    for i ← 1 to simulations do
        State ← Initialize()
        GamePossibleMoves ← 0
        GameMovesMade ← 0
        Agents ← [agent1, agent2]
        AgentIndex ← 0
        while State is not Terminal do
            Agent ← Agents[AgentIndex]
            AgentIndex ← (AgentIndex + 1) % 2
            GamePossibleMoves ← GamePossibleMoves +
             Length(State.PossibleMoves())
            Move ← Agent.GetMove(State)
            State ← State.Apply(Move)
            GameMovesMade ← GameMovesMade + 1
        end
        GameAverage ← GamePossibleMoves / GameMovesMade
        SumOfAverages ← SumOfAverages + GameAverage
    end
    return SumOfAverages / simulations
```

We then simulate **1000** games between **Minimax** and **Semi-random** agents, each for boards of dimensions 3, 5, 7 and 9, and for depths 1, 2 and 3, and the results of the average branching factor can be seen in Figure 4.3.

From Figure 4.3, like with the maximum branching factor, we can see a similar exponential trend of the average branching factors in order of N. Furthermore, based on only the four dimensions and their averages and maximum branching factors, the average branching factor seems to be almost half the maximum branching factor.

The average branching factor for board of dimension 9x9 was about **61**, which is very close to the value proposed by [Lisa Glendenning, 2002] which was 60.

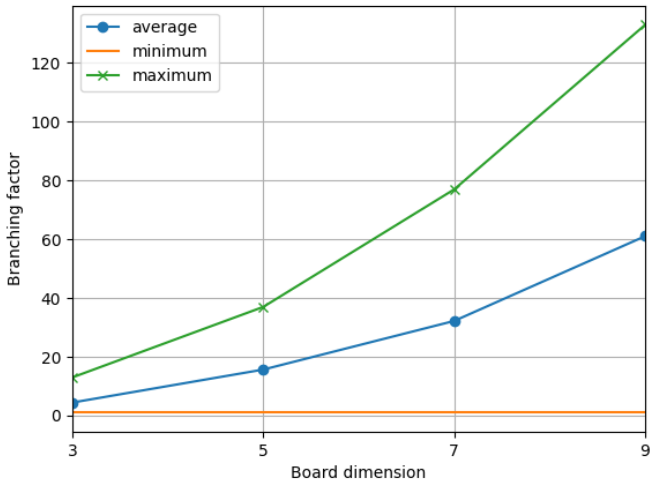


Figure 4.3: Branching factor for boards of different sizes

4.3 State space complexity

The state space complexity is a measure of a game complexity and refers to the total number of possible states in a game. For example, for illustration with the game of Tic-Tac-Toe, there are 9 squares, each being one of X, O or empty. The state space complexity for Tic-Tac-Toe can be written as $3^9 = 19,638$. Of course, this takes into account the illegal positions as well such as all Xs or all Os and hence can be considered as an upper bound on the state space complexity. However, especially in complex games such as Chess and Quoridor with many possible states, rules and illegal positions, exact state space complexity is difficult to estimate and a way to define the complexity is through an upper bound.

In Quoridor, this includes all the possible positions of both players' pawns on the board and all the possible configurations of walls. It is extremely difficult to find an exact value because we also need to account for wall placement rules 2.2.1 and pawn movement rules 2.2.2. Because of this, we loosen the regulations to come up with an upper-bound.

In [Mertens, 2006], the author has determined the state space complexity for a specific board of dimension 9x9. In this thesis, we generalize the complexity evaluation to a general board of size NxN.

For a board of Dimension NxN, the first pawn can be placed in N^2 possible locations. After the first pawn is placed in the board, the second pawn now has $N^2 - 1$ possible locations to be placed into. So, the total number of ways for pawn placement, S_p is given by

$$S_p = N^2(N^2 - 1) \tag{4.2}$$

As for walls, from Equation 4.1, we know that there are $2(N-1)^2$ ways of placing walls on the board, both horizontally and vertically. Since we know that each wall occupies 2 cells, we will use assumption made by the author of [Mertens, 2006] that placing the wall anywhere in the board takes away the possibility of placing walls at 4 different places. As each placed wall takes two cells, placing a wall, for e.g., horizontally, means that further walls cannot be placed in the

location of the wall, walls starting from the cell left and right to the cell of the wall and a wall placed vertically through the placed wall. Every placed wall hence takes away 4 spaces for wall placement. The same example is also valid for a vertically placed wall. Hence assuming that j walls are placed from a given game state, the total number of branches of the game state for wall placement may be $2(N-1)^2 - 4j$,

Furthermore, for a given board dimension (e.g., NxN), there is a fixed number of walls in the game N_w. For example, for the standard 9x9 board with 81 total squares, $N_W = 20$. We can extrapolate the number of walls that may be available for boards of other dimensions too based on the number of squares. For example, for 3x3 board, $N_W = 2$, for 5x5 board $N_w = 8$, for 7x7 board $N_W = 12$.

In the game tree, there may be a path where a total of N_W walls are used. In such scenario, the total number of branches L_{N_w} in the tree can be defined by the following equation:

$$L_{N_w} = \prod_{j=0}^{N_w} (2(N-1)^2 - 4j), \tag{4.3}$$

where, each component in the product defines the total number of states in the subsequent turn following a wall placement.

However, in the game, all the walls may not be necessarily placed. Hence, in such case, the total number of walls used may be variable and hence the total possibilities for wall placement S_w is defined by the following equation:

$$S_w = \sum_{i=0}^{N_w} \prod_{j=0}^{i} (2(N-1)^2 - 4j). \tag{4.4}$$

And thus, since the game tree consists of the possibilities of both pawn movement and wall placement, the total state space complexity is given by $S = S_w * S_p$. [Mertens, 2006]

$$S = N^2(N^2-1) \times \sum_{i=0}^{N_w} \prod_{j=0}^{i} (2(N-1)^2 - 4j). \tag{4.5}$$

In Figure 4.4, we can see the log plot of the state space complexity of wall placement and the game when varied with the dimension of the game (i.e., 3x3, 5x5, 7x7 and 9x9). In the figure, we can see the state space of the game grows exponentially with the increase in the board dimension. Moreover, the state space complexity is dominated by the complexity of the wall placement as seen by the difference between the two curves.

4.4 Game tree complexity

The game-tree complexity of a game, as defined in [Allis et al., 1994], is another measure of a game complexity alongside the state-space complexity. The authors define a game tree complexity as "*the number of leaf nodes in the solution search tree of the initial position(s) of the game*".

The game tree complexity refers to the number of possible games that can be played. Unlike the state space complexity, which measures the total number of

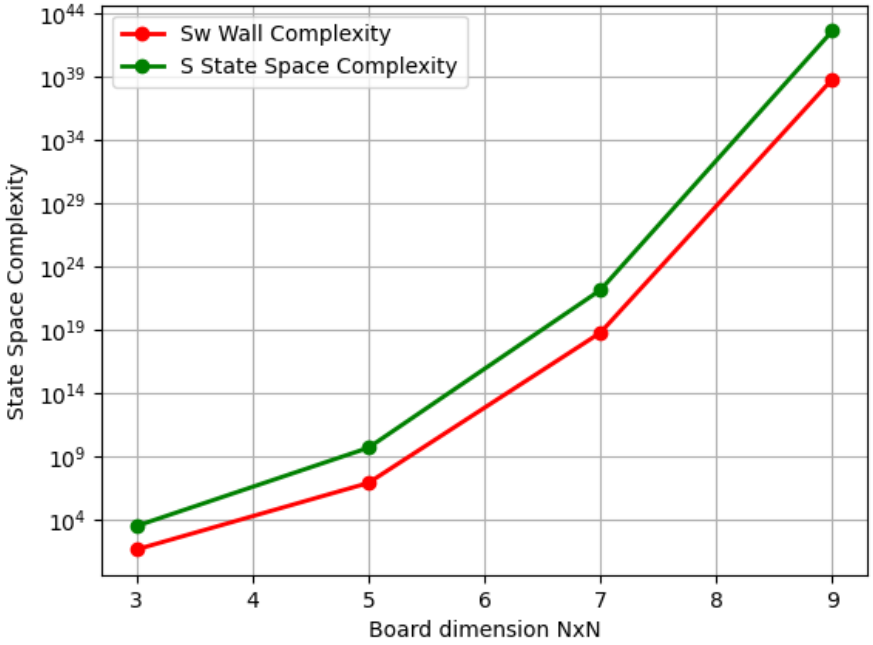


Figure 4.4: Complexity vs Board Dimension

possible states of the game from the starting position, the game tree complexity measures the size of the game tree. The size of the game tree or the game-tree complexity typically is larger than the state space complexity because a game state (counted only once in the state space complexity) can arise through multiple games (counted multiple times in the game tree complexity).

An example of the game tree complexity can be illustrated through the game of tic-tac-toe. In the game tree, player A has 9 positions to put the mark (e.g., X or 0) on. Subsequently, in the second move, player B can put the mark on 8 spaces as one has been taken away in the first move and so on. Hence, the game tree complexity of tic-tac-toe can be defined as $9 \times 8 \times \cdots \times 1 = 9! = 362380$. In comparison to the state space complexity of tic-tac-toe, the game tree complexity is higher as a position arising through different orders of play is counted as one state with the state space complexity and multiple game tree positions with the game tree complexity.

In complex games like Quoridor, the game-tree complexity G can be estimated by raising the average branching factor B_{avg} to a power of the total number of moves by the players D_{avg} [Mertens, 2006], which is given by the following equation:

$$G = B_{avg}^{D_{avg}} \tag{4.6}$$

The average depth D_{avg} simulated for different dimensions of the game can

be found below:

$$D_{avg,3x3} = 11$$
$$D_{avg,5x5} = 47$$
$$D_{avg,7x7} = 72$$
$$D_{avg,9x9} = 97$$

Subsequently, we can now determine the game tree complexity based on the determined B_{avg} and D_{avg}. The game tree complexity for different dimensions can be written as:

$$G_{3x3} = 8.58 * 10^{9}$$
$$G_{5x5} = 9.94 * 10^{58}$$
$$G_{7x7} = 5.56 * 10^{113}$$
$$G_{9x9} = 1.50 * 10^{173}$$

From this, we can infer that the game state complexity for Quoridor, like the state space complexity, also increases exponentially with the increase in the dimension of the game.

4.5 Comparison with other games

	$\log_{10}(S)$	$\log_{10}(G)$	B_{avg}
Tic-tac-toe	3	5	4
Quoridor 3x3	3	10	8
Connect-four	13	21	4
Quoridor 5x5	10	59	18
Quoridor 7x7	22	113	38
Chess	44	123	35
Quoridor 9x9	42	173	61
Go	170	360	250

Table 4.1: State space, game tree and branching factor comparison between well-known games

In the Table 4.1, we present the logarithm of the state space complexity ($\log(S)$), the game state complexity ($\log(G)$) and the average branching factor (B_avg) of some popular games from the literature [Mertens, 2006].

As we can see from the above table, Quoridor with board dimension 3x3 has the state space complexity and the game tree complexity similar to that of Tic-tac-toe. It can also be inferred that the Quoridor game with dimension 9x9 has similar complexity compared to chess.

Chapter 5

Background

In this chapter, we give a brief overview of different AI techniques that have been considered for implementing the agent for Quoridor in this thesis including the Minimax algorithm, MCTS and A-star algorithm.

5.0.1 Minimax algorithm

Minimax algorithm, first proven by John von Neumann in 1928 in his paper *Zur Theorie Der Gesellschaftsspiele* [v. Neumann, 1928], is a very popular algorithm employed in many decision-making scenarios for e.g., in decision theory, game theory and even philosophy. As suggested by the name minimax, the idea of the algorithm is to minimize the player's loss when the opponent makes a decision that gives the player the maximum loss. This algorithm has been implemented in many multi-player strategy games such as Tic-Tac-Toe [Savelli and de Beauclair Seixas, 2008].

Mathematically, a minimax algorithm can be defined by the following equation:

$$\bar{v}_i = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}) \tag{5.1}$$

where,

$$i, -i = \text{index of the player of interest, opponent respectively} \tag{5.2a}$$
$$a_i, a_{-i} = \text{action of the player of interest, opponent respectively} \tag{5.2b}$$
$$v_i = \text{value function of player i} \tag{5.2c}$$
$$\bar{v}_i = \text{minimax value of the player of interest} \tag{5.2d}$$

As defined in the Equation (5.1), the minimax algorithm comprises of two parts. The first part is maximizing part where the player chooses an action from a set of possible actions to maximize the evaluation of the game. Then, the player determines the subsequent action of the opponent to minimize the evaluation of the game. This is clarified further with the following example.

The Figure 5.1 consists of nodes that define either player's or the opponent's actions. In our minimax implementation, we have considered an evaluation function as a linear combination of multiple features such as shortest distance to the goal and remaining number of walls. This is explained in detail in Section 6.4. The numbers for each node Figure 5.1 represent the evaluation of this function. For each action, there can be an evaluation where higher evaluation may mean it

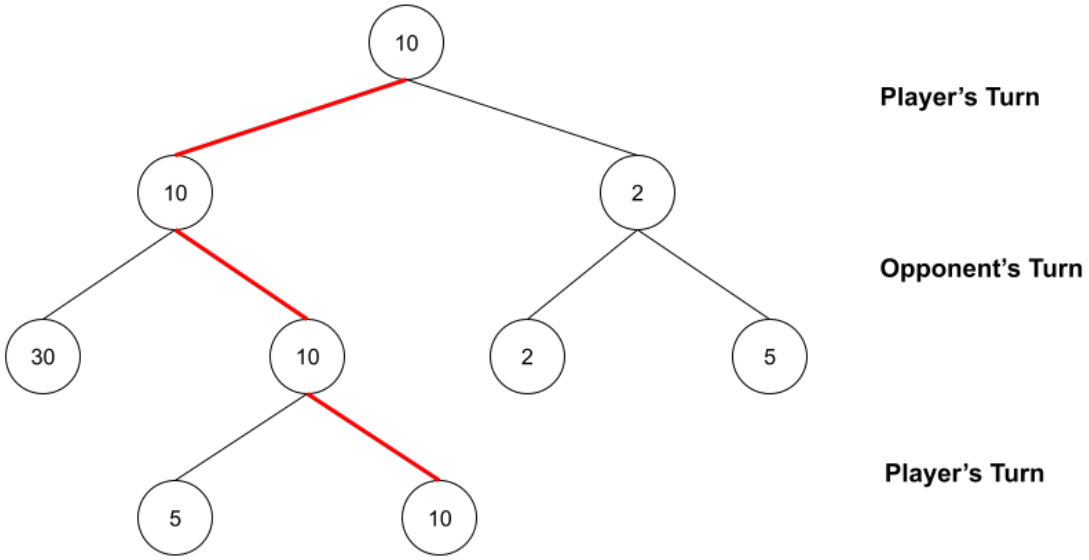


Figure 5.1: Figure illustrating an exemplary game tree and decision based on minimax algorithm

is favouring the player. For example, an action of the player causing two possible evaluations of 10 and 20 would mean the player would have higher advantage choosing the action leading to evaluation of 20.

The minimax algorithm starts with a game tree with possible moves of the player and opponent. The evaluation of the leaf nodes positions are assigned, for example 5 and 10 in the figure. The player chooses a move to maximize the evaluation on their turn whereas the opponent wants to minimize the evaluation. The player chooses evaluation of 10 (between 10 and 5) on its turn and subsequently evaluation of 10 (between 10 and 30) in the opponent's turn. The red line shows the path the player determines as optimal leading to a decision choosing the the action labelled with evaluation of 10.

The minimax algorithm involves in the player performing an exhaustive search on the game tree to determine a sequence of maximizing and minimizing moves. The complexity of such algorithm in large game tree often means such search is often impossible due to limited computational resources. To limit this complexity, further techniques such as depth-limited minimax, alpha-beta pruning and parallel minimax algorithm can be used.

Alpha-beta pruning

Alpha-beta pruning is one way to reduce the compelxity of the minimax algorithm without affecting the performance of it.

In many cases, there may be the actions of the player in the search tree that can be evaluated worse than another action already evaluated. In such cases, where the better move or action has already been determined, it may not be useful to further evaluate the subsequent moves of the player and the opponent. Hence, the alpha-beta pruning reduces the game complexity by not further evaluating the branches of the node with evaluation worse than what has already been determined with another node.

As its name suggests, the alpha beta algorithm maintains two values, alpha

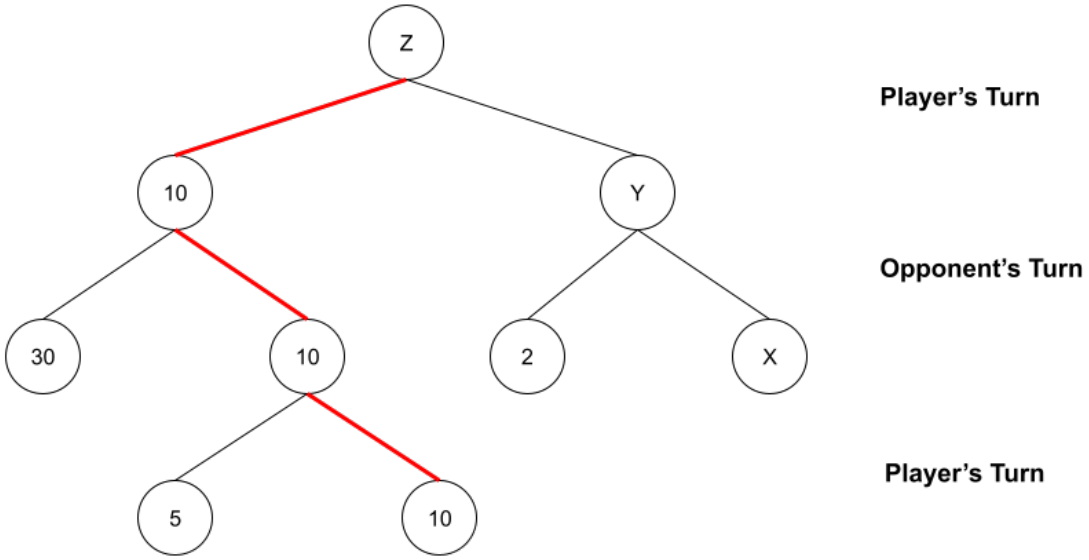


Figure 5.2: Exemplary figure illustrating the alpha beta pruing for minimax algorithm

and beta values. The alpha value stores the minimum score the player is assured to get while the beta value records the maximum score the opponent is assured to get. Whenever the evaluation of the minimum score of the player is higher than the maximum score after the subsequent move of the opponent (or in other words alpha > beta), the alpha-beta pruning algorithm stops evaluating the opponents position. This reduces the number of nodes in the search tree and hence reduces the complexity of the minimax algorithm.

In Figure 5.2, if the player determines that

$$Z = max(10, Y) = max(10, min(2, X)), \tag{5.3}$$

the value of X does not influence the value of Y or Z as $Y = min(2, X) \leq 2$ and hence $Z = max(10, \leq 2) = 10$. In this case, the player may not evaluate the branch X or branch Y further reducing the game tree and hence the computational complexity of the algorithm.

Parallel minimax

Another way to improve the time complexity of the minimax algorithm is to parallelize the algorithm. The minimax algorithm involves in evaluating multiple nodes of the game tree. The way to parallelize such algorithm is to run different processes, in this case, evaluation of the position associated with different nodes, in different threads. This ensures that even though computational complexity may remain the same, the time complexity of the algorithm is distributed over multiple threads and possibly multiple processors.

5.0.2 Monte-Carlo Tree Search

MCTS [Coulom, 2006] is a heuristic tree search algorithm popular in decision-making processes, mostly popular in strategic games where the game tree space

is too large to traverse. One problem with the minimax algorithm is that it requires a robust and accurate evaluation function to evaluate a given position in the game. This problem can be even more relevant when the game tree space is too large making it difficult to find the evaluation of a position. The basic idea of the MCTS algorithm is that is narrows down on certain areas of the game tree, such that the exhaustive traversal and search of the tree is not required. The algorithm achieves this by taking random samples in the tree space and building a search tree based on it.

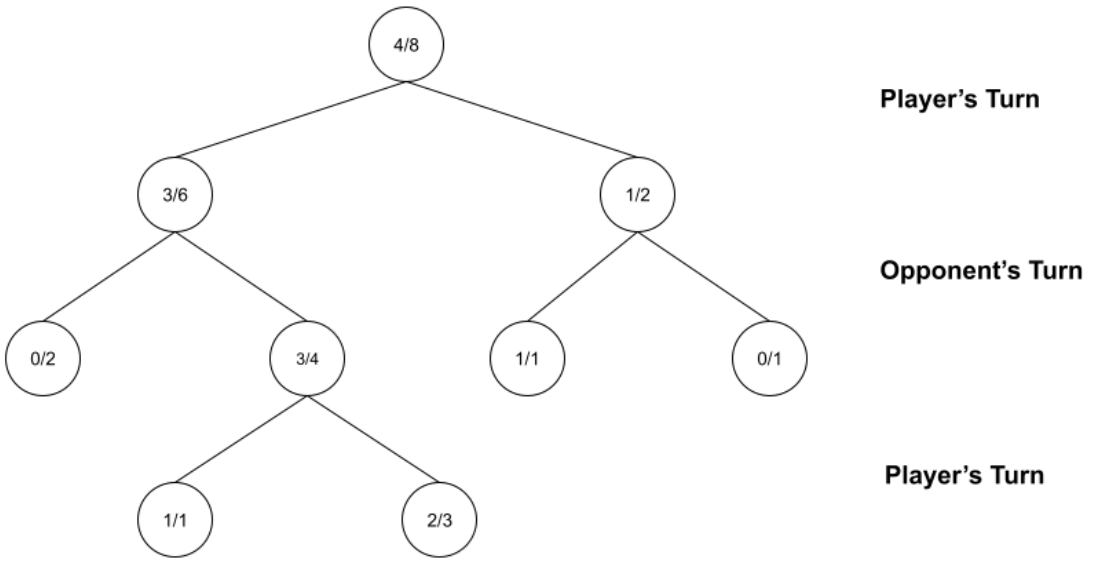


Figure 5.3: Figure illustrating an iteration of the MCTS algorithm

As shown in Figure 5.3, MCTS algorithm determines the best path to the destination node in the game tree based on multiple trial runs through the game tree. In the figure, each node is labelled with a fraction where the denominator represents the total number of game runs through the node and the numerator represents the total number of wins for the player. For example, the game is simulated a total of 8 times in the example where the player wins 4 of the runs. The player wins 3/6 when taking an action while 1/2 when taking another action and so on.

The MCTS algorithm builds upon the following framework:

1. **Selection:** This step involves the algorithm choosing a move based on either a good move determined in previous iterations or a exploratory new move. The algorithm uses the Upper Confidence Bound for Trees **(UCT)** to guide this decision, balancing the tradeoff between exploration of uncharted nodes and exploitation of nodes with a high success rate.

   Mathematically,

   $$UCT(n) = \overline{X}(n) + cU(n) \tag{5.4}$$

   where,

   $$\overline{X}(n) = \text{average win rate of node n} \tag{5.5a}$$

   $$U(n) = \sqrt{\frac{\ln N(P(n))}{N(n)}} \tag{5.5b}$$

2. **Expansion:** This step involves the algorithm to add a new node to the game tree determined during the selection process. A node can simply be a valid move starting from the node from where no simulation step has been played out. In Figure 5.4, an un-evaluated option (e.g., node inside the dotted box) is explored and simulated.

3. **Simulation:** This step involves the agent playing out the game using policy. One of the policies can simply be a random policy (e.g., choosing a move based on certain distribution).

4. **Back propagation:** Finally, based on the simulation step, this step involves in the algorithm updating the nodes. In Figure 5.4, the red arrows show the back propagation step as a result of exploration and simulation step where the probabilities or the weights of the nodes are updated as a result of exploration and simulation steps.

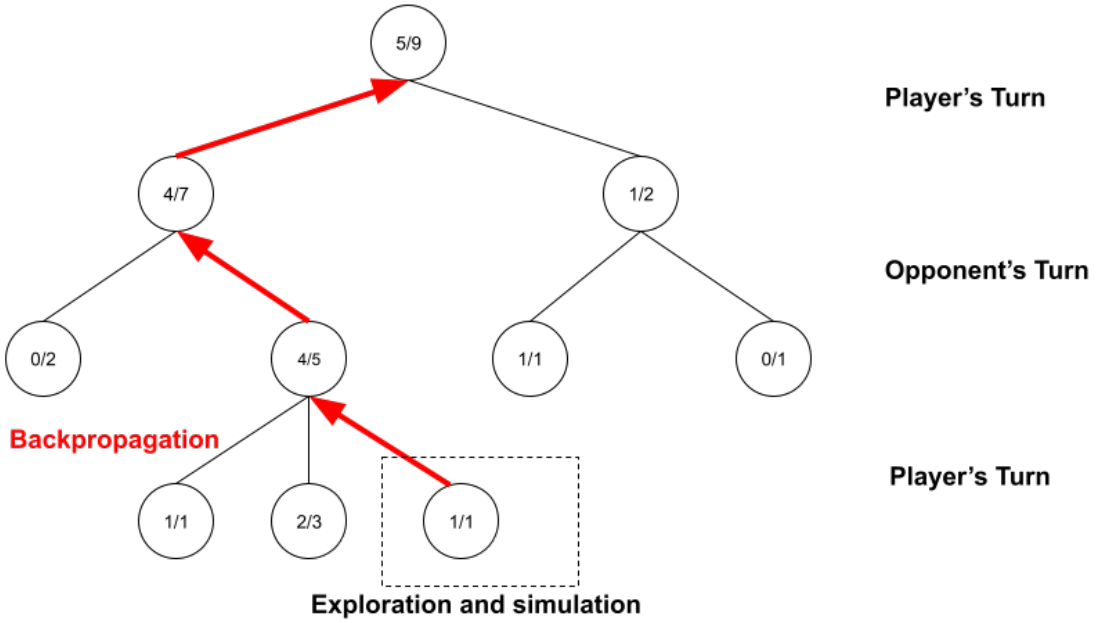


Figure 5.4: Figure illustrating steps 2, 3, and 4 of the MCTS algorithm

The advantage of MCTS algorithm over the minimax algorithm is that the MCTS algorithm does not require any evaluation function as the weights are determined based on multiple simulation runs. On the contrary, the MCTS algorithm requires multiple runs through the game to determine the weights.

5.0.3 A-star algorithm

A star is a popular algorithm [Hart et al., 1968] used mainly for graph search and traversal problems. The main aim of the A-star algorithm is to find a path between a starting node and an destination node with the least cost. The cost function of the algorithm comprises of two components, the distance from the starting node to the current node and the estimated heuristic from the current node to the destination node. As the distance to the destination node may not be exactly known, one may use the estimate such as Manhattan distance and

Euclidean distance. In the context of Quoridor gameplay, for example, we use the *Manhattan* heuristic (see Equations 6.2 and 6.3).

The cost function, or "f" score, for a node depends on two components "g" value and "h" value. "g" value represents the distance from the source node to the node and "h" value represents the cost of the path from the node to the destination node, often defined by a heuristic. Mathematically,

$$f = g + h \tag{5.6}$$

Chapter 6

Implementation

In this chapter, we delve into the practical aspects of creating AI agents capable of tackling abstract strategy games. The focus is on constructing adaptable interfaces that can be applied to a variety of games, ranging from the simplicity of Tic-tac-toe to the profound strategic depths of Go, with our primary case study being the game of Quoridor.

C Sharp (C#), is selected as the language of choice, mainly for its robustness, versatility, and strong support for object-oriented programming paradigms. C#'s rich feature set makes it an excellent tool for developing sophisticated AI frameworks that require a blend of performance, maintainability and readability.

6.1 Interfaces

The architecture of our implementation leverages interfaces, which specify a set of methods relevant to game mechanics. They form the backbone of our AI systems, ensuring that the algorithms are versatile and can be adapted to new games with minimal changes to the underlying codebase. An example of such interface usage can be seen in Section 6.3

Through the use of generic parameters **TPlayer**, **TMove**, and **TGame**, these interfaces offer a framework that is adaptable to various game entities such as players, moves, and game states.

Before describing the game-specific interfaces, we first introduce the generic interface **IAIStrategy<TMove, TGame, TPlayer>**, which acts as a common interface for all our AI agents.

```
interface IAIStrategy<TMove, TGame, TPlayer>
{
    string Name { get; }

    AIStrategyResult<TMove> BestMove(
        TGame game, TPlayer player);
}

class AIStrategyResult<TMove>
{
    double Value { get; set; }
    TMove BestMove { get; set; }
```

```
}
```

This interface provides a common property **Name**, a `string` representing the name of the agent, and a method **BestMove** that takes in a game and the current player in the game, and returns a tuple - the first item labelled as **BestMove** (parametrized over generic type **TMove**) being the best move for the current player in the given game state and the second item of the tuple labelled as **Value**, a `double` type representing the value corresponding to the best move selected by the AI agent. For example, for the Minimax algorithm, the Value corresponding to the best move would be the state with the highest payoff function, which could be used for debugging purposes.
The following interfaces are used within the **BestMove** method for each AI agent, and are meant to be defined explicitly by the user within their game.

IValidMoves

```
interface IValidMoves<TMove>
{
    IEnumerable<TMove> GetValidMoves();
}
```

The **IValidMoves<TMove>** interface is the most fundamental interface used by (almost) all our AI systems. Parametrized over the generic type **TMove**, this interface provides access to all valid moves (of a user-defined **TMove** type), e.g from the current game state, which our implemented AI algorithms evaluate and find the best move.

IPlayer and IOpponent

```
interface IPlayer<TPlayer>
{
    TPlayer CurrentPlayer { get; }
}

interface IOpponent<TPlayer>
{
    TPlayer Opponent { get; }
}
```

Since we are focused on abstract two-player games, we also provide the **IPlayer<TPlayer>** and **IOpponent<TPlayer>** interfaces, both parametrized over the generic **TPlayer** type. The MCTS algorithm, for example, uses these interfaces during the simulation and back-propagation phases.

IDeepCopy

```
interface IDeepCopy<T>
{
    T DeepCopy();
}
```

Although C# has the **ICloneable<T>** interface provides a **T Clone()** method that we could use instead of our defined **IDeepCopy<T>** interface, the method we define leaves no room for ambiguity on shallow vs deep copying of relevant objects. This method is used by e.g the Parallel Minimax algorithm in order to ensure the original game state don't get changed by any means while exploring the game tree by making several moves.

ITerminal

```
interface ITerminal
{
    bool HasFinished { get; }
}
```

Another important information for a game is whether the game reached the terminal state (win, loss, draw). This information helps update evaluations of game states and also notifies the algorithms to stop looking further. For example, the Simulation step of the MCTS algorithm performs moves until the game reaches a terminal state.

IStaticEvaluation

```
interface IStaticEvaluation
{
    public double Evaluate(bool currentMaximizer);
}
```

The **IStaticEvaluation** interface computes the heuristic evaluation of the current game state, indicating the desirability of the state for the player who is currently maximizing or minimizing the game value. This is used by the depth-limited **Minimax** algorithm to statically evaluate game states instead of exploring them when the specified depth has been reached.

IMove

```
interface IMove<TMove>
{
    void Move(TMove move);

    void UndoMove(TMove move);
}
```

The **IMove<TMove>** interface is yet another fundamental interface that allows games to progress further. To further evaluate game states, AI algorithms must make different moves from current game state, then run any evaluation function and make decisions based on that. The **Move** method provides access to do exactly that.
The **UndoMove** method complements the **Move** method in that the algorithms can track back to the original game state using the **UndoMove** method before returning the best move it found based on different game state evaluation.

INeighbors

```
interface INeighbors<TMove>
{
    IEnumerable<TMove> Neighbors(TMove pos);
}
```

The **INeighbors<TMove>** interface yields the neighboring positions or states from a given position (parametrized using the generic **TMove** type) crucial for determining potential player actions. This is used by the **A*** algorithm to find the shortest path to the goal.

IRandomizableMoves

```
interface IRandomizableMoves<TMove>
{
    public IEnumerable<TMove> RandomizableMoves();
}
```

The **IRandomizableMoves<TMove>** interface returns all the valid moves (that guide the current game state towards the terminal state) that can be made from the current state and can be used by the random agent.

In context of the Quoridor game, if we move randomly on each turn, there's a possibility of never reaching the goal row. So, we would instead like to have an option to place wall randomly, but move towards the goal. So, we define the **IRandomizableMoves** interface and return all possible walls that can be placed in the board, and use a different approach for pawn movements.

6.2 Agents implementation

In this section, we discuss the generic AI agent (algorithm) implementation, namely Random, Semi-Random, Minimax, A-Star, Monte Carlo Tree search, and describe how the aforementioned interfaces are used as building blocks for the algorithm.

6.2.1 Random Agent implementation

We consider a random agent as a baseline for comparing the performance of the other implemented agents.

The random agent uses the **IValidMoves<TMove>** interface to get a list of all valid moves, and then picks move at random. The class definition for the Random Strategy is as follows:

```
public class RandomStrategy<TMove, TGame, TPlayer>(int
   seed) :
    IAIStrategy<TMove, TGame, TPlayer>
        where TGame : IValidMoves<TMove>
```

As the Random Strategy implements the **IAIStrategy<TMove, TGame, TPlayer>** interface, it has a property **Name** of `string` type and the **BestMove(TGame game, TPlayer player)** method that returns the best move

and the value that accompanies with it is the seed provided while initializing the random nubmer generator. The pseudocode for the Random agent is given below.

```
public string Name => "Random";

public AIStrategyResult<TMove> BestMove(
    TGame game, TPlayer player)
{
    //IValidMoves<TMove>
    var validMoves = game.GetValidMoves();

    var randIndex = random number between 0 and
       validMoves.Count();
    return { BestMove = validMoves[randIndex], Value =
       seed };
}
```

As described above, the Random Agent picks a move from the set of valid moves based on the **IValidMoves<TMove>** interface.

6.2.2 Semi-Random agent implementation

There are cases where we want to return a random move, but we don't want the game to continue forever by the random agent possibly returning move that never end in a terminal state. In this case, we want to guide the random agent to produce a random move, but also make sure the game will terminate eventually.

The semi-random agent uses the **IRandomizableMoves<TMove>**. It also takes in a strategy to get the best move and then add it to the list of randomizable moves.

```
public class SemiRandomStrategy<TGame, TMove, TPlayer>
    : IAIStrategy<TMove, TGame, TPlayer>
        where TGame : IRandomizableMoves<TMove>
```

Then, in the **BestMove** method, it gets the list of moves that can be randomized, finds the best move from a list of non-randomizable move set and then produces a random move from these two. The pseudocode from the Semi-Random algorithm is given below:

```
public TMove BestMove(TGame game, TPlayer player)
{
    //IRandomizableMoves<TMove>
    //these moves won't result in a possible infinite
       game
    possibleMoves = game.RandomizableMoves();

    //non-randomizable move. This move might create
    //infinite game loop if not used strategically,
    //eg. pawn moves in Quoridor.
    nonRandomizableMove = _strategy.BestMove(game,
       player);
```

```
    //add the non-randomizable move to the
    //list of all moves
    possibleMoves.Add(nonRandomizableMove);

    return random move from possibleMoves
}
```

Adding more context to the Quoridor example in the **IRandomizableMoves<TMove>** interface description in Section 6.1, the Semi-Random algorithm in Quoridor would process and return the best move the following way:

```
unplaced_walls = game.GetRandomizableMoves();
//Shortest path to the goal row
best_pawn_move = AStar.BestMove(game,
   game.CurrentPlayer);

possible_moves = unplaced_walls.Add(best_pawn_move);
random_index = random nubmer from 0 to possible_moves;
return { BestMove = possible_moves[random_index],
   Value = random_seed };
```

This algorithm is especially useful for the Simulation step of the MCTS algorithm as an approach to shorten the game length to reach the terminal state in context of Quoridor game.

6.2.3 Minimax agent implementation

The minimax algorithm uses the **IValidMoves<TMove>** interface to get a list of all moves, **IMove<TMove>** interface to perform moves, get a static evaluation (therefore needing the **IStaticEvaluation** interface) of the game state and undo moves. It also requires the **ITerminal** interface to check if the game reached the terminal state, and an access to the **IPlayer<TPlayer>** interface, especially during the static evaluation in order to know which player to evaluate the board for.

The class definition for Minimax is as follows:

```
public class Minimax<TPlayer, TMove, TGame>
    : IAIStrategy<TMove, TGame, TPlayer>
        where TGame :  ITerminal, IMove<TMove>,
           IStaticEvaluation,
            IValidMoves<TMove>, IPlayer<TPlayer>
```

In our implementation, we consider a depth limited minimax algorithm. Then, in the **BestMove** method, it traverses down the game tree until it reaches a certain depth, in which case it calls for the game to perform static evaluation and returns a state with the best evaluation result.

```
//ITerminal
if (depth limit reached or game.HasFinished)
    //IStaticEvaluation
    return game.Evaluate(maximizingPlayer);
```

```
bestScore = maximizingPlayer ? MinValue : MaxValue;
bestMove = none
//IValidMoves
foreach(var move in game.GetValidMoves())
{
    //IMove
    game.Move(move);
    result = recursive call to minimax with depth-1
    if (maximizingPlayer and result > bestScore) OR
       (!maximizingPlayer and result < bestScore) {
        bestScore = result;
        bestMove = move;
    }
    //IMove
    game.UndoMove(move);
}
return bestMove;
```

As explained in Section 5.0.1, the minimax algorithm involves in evaluating certain positions for the player and the opponent and based on the evaluation. In our implementation, we consider the following evaluation function for the minimax agent.

Assume a Quoridor game instance of 2 players P and Q, with P starting at cell $C_{1,c}$ and Q starting at cell $C_{N,c}$, where N is the dimension of the game board.

Suppose P is at cell $C_{x,y}$ and Q is at cell $C_{u,v}$ in an arbitrary game state G_s, and let S_P be the shortest path from $C_{x,y}$ to P's goal row $C_{N,*}$ and let S_Q be the shortest path from $C_{u,v}$ to Q's goal row $C_{0,*}$.
Let W_P denote the number of walls left for player P and let W_Q denote the number of walls left for player Q at state G_s.

Then, we define our static evaluation function for player P, F_P in game state G_s as:

$$F_P(G_s) = S_P(G_s) - S_Q(G_s) + W_P(G_s) - W_Q(G_s) \tag{6.1}$$

So, for the static evaluation function in our implementation of the Quoridor game, we consider the following 4 features:

- Shortest distance to goal for player P
- Shortest distance to goal for player Q
- Number of walls left for player P
- Number of walls left for player Q

In our implementation, the minimax algorithm is implemented with further optimizations including the alpha-beta pruning and the parallel version of the algorithm as described below:

Alpha-beta pruning implementation

The Alpha-beta pruning variant of Minimax uses the same interface as the aforementioned Minimax implementation. So, we present the pseudocode highlighting the core logic of this variant of Minimax.

```
foreach (var move in game.GetValidMoves())
    game.Move(move);
    result = recursive call to minimax with alpha,
       beta, depth-1
    game.UndoMove(move);
    if (maximizingPlayer)
        if (result > bestScore)
            update bestScore and bestMove
        if (bestScore > beta)
            break;

        alpha = Math.Max(alpha, bestScore);
    else
        if (result < bestScore)
            update bestScore and bestMove
        if (bestScore < alpha)
            break;

        beta = Math.Min(beta, bestScore);
return bestMove;
```

We have implemented a recursive minimax agent with alpha-beta pruning that is depth limited. The algorithm alternates with the variable *maximizingPlayer* being 1 and 0 in each step indicating whether its the player's turn or the opponent's. In each case, the player determines the evaluation of the branch in the variable *result* and sets the node as the best if it is the maximum (if player's turn) or minimum (if opponent's turn).

In the above implementation, the parameter depth is introduced to only consider the depth limited version of the algorithm, and alpha and beta variable limit the search space of the nodes in game tree.

Parallel minimax implementation

In this thesis, in order to reduce the complexity of the minimax algorithm, we have combined it together with the parallel implementation. In the following, we present the implemented pseudocode of the implemented minimax algorithm which is depth limited, run with alpha-beta pruning and with parallel implementation.

```
Parallel.ForEach(game.GetValidMoves(), (move,
   loopState) =>
{
    //IDeepCopy<T>
    var clonedGame = game.DeepCopy();
    clonedGame.Move(move);
```

```
        var result = call minimax recursively with alpha,
           beta, depth-1

        lock
        {
            bestMove = Update(result, bestMove);
        }
    }
    return bestMove;
```

Parallel Minimax additionally requires the **IDeepCopy<T>** interface to ensure the original game state doesn't get altered in any way. The other two variants of Minimax algorithms were only using the **IMove<TMove>** interface since one thread was responsible for changing the game state sequentially so undoing a move would cancel out the applied move effectively.

6.2.4 Monte-Carlo tree search agent implementation

The class definition of MCTS is as follows:

```
class MonteCarloTreeSearch<TMove, TGame, TPlayer>
    : IAIStrategy<TMove, TGame, TPlayer>
        where TGame :  ITerminal, IMove<TMove>,
            IDeepCopy<TGame>, IOpponent<TPlayer>,
            IPlayer<TPlayer>, IWinner<TPlayer>,
               IValidMoves<TMove>
        where TPlayer : IEquatable<TPlayer>
```

We first present the pseudocode for the core part of the MCTS algorithm, which is present inside the **BestMove** method, then show how the interfaces we used in the class definition are relevant for the algorithm.

```
while resources left:
    selectedNode = TreePolicy(root);
    //IDeepCopy
    winner = Simulation(selectedNode.DeepCopy());
    BackPropagation(selectedNode, winner);
return best child of the root node using UCT
```

The **TreePolicy** method combines both the Selection and the Expansion steps. The pseudocode for the tree policy is described below:

```
Input: root node
Output: selected/extracted node

While input node n is non-terminal
    if n has not been fully expanded
        return Expand(n)
    else
        return BestChild(n)
```

The **Expand** method gets the next move from the pool of available moves, creates a child node from the states as a result of applying the said move.

The **BestChild** method returns the best child using Equation 5.4.

We now present the pseudocode for the Simulation step of the MCTS algorithm below. The simulation step takes in a strategy for simulating the game until it is done, in which case the function returns the winner of the simulation. One of Random or Greedy agents is typically used as the simulating agent.

```
//ITerminal
 while(!game.HasFinished)
{
    //IAIAgent, IPlayer
    var move = moveStrategy.BestMove(
        game, game.CurrentPlayer).BestMove;
    //IMove
    game.Move(move);
}
//IWinner
return game.Winner;
```

The Backpropagation step then updates the win count of nodes in the tree using the winner returned by the Simulation phase.

```
input: node (selected during TreePolicy step),
       winner of the simulation
while (node is not null)
{
    node.Count = node.Count + 1
    //IComparable, IOpponent
    if (node.Opponent.Equals(winner))
        node.Wins = node.Wins + 1

    node = node.Parent;
}
```

As described in the above code snippet, the MCTS implementation depends on the four steps including Selection, Expansion, Simulation and Back-propagation, as described in Section 5.0.2.

6.2.5 A-Star agent implementation

In this thesis, we have further implemented the A-star agent for Quoridor, which is presented in the pseudocode below:

We begin by describing the **IPlayer<TPlayer>** interface as follows:

```
public interface IAStarPlayer<TMove>
{
    TMove GetCurrentMove();
    bool IsGoal(TMove move);
    double CalculateHeuristic(TMove move);
}
```

It is parameterized over the generic **TMove** type. The first method **TMove GetCurrentMove()** provides the current user-defined position (of type TMove). This could be e.g position in the game board.
The **IsGoal(TMove move)** method checks if the new move is a goal, i.e the player's goal move. E.g in context of Quoridor, where **Vector2** is used as **TMove**, each player has a **IsGoal(Vector2 pos)** that checks if the position is one of player's goal row.

For the A-star agent, we need to define the heuristic to define the cost function from the current state to the destination state defined by **CalculateHeuristic(TMove move)**. In our implementation of the A-star agent, for example, we use **Manhattan Distance** as the heuristic.

Suppose player P is at at cell $C_{x,y}$ and player Q starts at cell $C_{a,b}$, and let n be the Quoridor board dimension.
Player P's goal is to reach row n regardless of the column it is at, and Player Q's goal is to reach row 1. So, the heuristic function for player P, H_P is given by

$$H_P = |n - x| \tag{6.2}$$

and the heuristic function for player Q, H_Q is given by

$$H_Q = a \tag{6.3}$$

Before we describe the class signature for the **A*** algorithm, we would want the user-defined **TMaze** type to implement the **INeighbors<TMove>** interface to get access to neighboring moves (e.g positions), given a move (or position).

```
public class AStar<TMove, TMaze, TPlayer>
    where TPlayer : IAStarPlayer<TMove>
    where TMaze : INeighbors<TMove>
```

The pseudocode for the **BestMove** method is as follows:

```
openSet = { start }
var closedSet =  { }

while (openSet is not empty)
    nodeWithLowestFscore = node in openset with
        lowest f-score value

    //IAStarPlayer<TMove>
    if player.IsGoal(nodeWithLowestFScore):
        return nodeWithLowestFScore

    closedSet.Add(nodeWithLowestFscore);
    openSet.Remove(nodeWithLowestFscore);

    //INeighbors<TMove>
    foreach (neighbor in
       maze.Neighbors(nodeWithLowestFScore))
    if (!openSet.Contains(neighbor) || neighborNode.G
       < G)
```

```
{
    neighbor.G = G;
    //IAStarPlayer<TMove>
    neighbor.H =
       player.CalculateHeuristic(neighbor);
    neighbor.F = G + neighbor.H;
}
```

The A-star algorithm starts by maintaining two sets *openSet* and *closedSet*. *openSet* consists of nodes with children not yet visited and *closedSet* consists of nodes with children nodes explored already. The node with lowest cost function ("f" score) is explored and marked as the *currentNode* from the *openSet*. Subsequently, the 'f' scores of the neighbours of the *currentNode* is calculated. This algorithm runs until the destination node in the tree is reached.

6.3 Generalization of AI interface to other games

In this section, we demonstrate how seamless it is to integrate the AI agents with our implementation to other games. We will use an example to integrate our interface to the tic-tac-toe game for this purpose.

The interfaces with our implementation as described earlier are parametrized over 3 generic types, namely TGame, TMove and TPlayer. For Tic-tac-toe, we will use the *int* type for TMove and TPlayer parameteters. For TGame, we will use the **TicTacToe** class type.

```
public class TicTacToe :
    ITerminal, //used by Minimax, used by MCTS
    IValidMoves<int>, //Minimax, MCTS
    IMove<int>, //Minimax, MCTS
    IPlayer<int>, //Minimax, MCTS
    IOpponent<int>, //MCTS
    IDeepCopy<TicTacToe>, //MCTS
    IWinner<int>, //MCTS
    IStaticEvaluation //Minimax
```

For the Tic-Tac-Toe game, we will need a 3x3 array representing the game board, and a property turn that represents which player's turn it currently is. Turn therefore will have 2 values, 1 and 2 representing player 1 and player 2 respectively.

```
private int[,] Cells = new int[3, 3];
private int turn = 1; // 1 -> p1, 2 -> p2
```

The game board, represented by `Cells` property, is initially all zeros. Over the course of the game, it will contain values 0, 1 or 2.

We will now implement all the interfaces above. We start by implementing the **IValidMoves<int>** interface. To get all the valid locations, i.e Cells marked by 0, we can encode the Cell's i and j position by the following equation

$$Move(C_{ij}) = i + 3 * j \tag{6.4}$$

As an example, consider the cell $C_{1,2}$. From Equation 6.4, we have that $Move(C_{1,2}) = 1 + 3 * 2 = 7$.

```
public IEnumerable<int> GetValidMoves() {
    for(int i =  0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (Cells[i, j] == 0)
                yield return i + 3 * j;
}
```

We now write a method *Place* that takes in two arguments, move and mark, move being an integral value represented by Equation 6.4, and mark being one of 0, 1, 2. 1 and 2. On each placement, we can also retrieve the winner(if any) and get information on whether the game terminated, so we implement both the **ITerminal.HasFinished** and **IWinner.Winner** properties. To check if the game has finished we check if 3 adjacent sides of the game board are filled by the same player. These include diagonals too. We define **Winner** to hold 4 possible values - 1 and 2 indicating player 1 and player 2 victory respectively, 0 indicating a draw and -1 indicating that the game is still in progress. Before all these, we firstly need to decompose the move we encoded by Equation 6.4 to i and j values. To do so from given move $Move(C_{ij})$, we can use the following equations:

$$i = Move(C_{ij}) \mod 3 \tag{6.5}$$

$$j = \frac{Move(C_{ij})}{3} \tag{6.6}$$

We then place one of 'X' or 'O' signs, (or remove them if we want to undo the last action), check if any player won, and if not, switch turns.

```
public bool HasFinished => CheckWin();

// 0 draw, 1 -> p1, 2 -> p2, -1 game in progress
private int _winner = -1;

public int Winner => _winner;

private void Place(int move, int item) {
    int i = move % 3;
    int j = move / 3;
    Cells[i, j] = item;
    CheckWin();
    turn = turn % 2 + 1;
}

void CheckWin() {
    //check all 3 consecutive adjecent squares
       (including
    //diagonals), and return true if they're filled by
       the
    //same player.
    // update the _winner variable based on this
```

```
}
```

We can then implement the `Move` and `UndoMove` methods. Both these methods use the `Place` method. We also switch turns after a successful Move/UndoMove operation.

```
 public void Move(int move) {
    Place(move, turn);
}

public void UndoMove(int move) {
    Place(move, 0);
}
```

We also need to implement the `CurrentPlayer` and `Opponent` properties implemented by the **IPlayer** and **IOpponent** interfaces respectively. We simply use the value held by the **turn** variable in our implementation to return it. The **turn** variable holds the index of the current player, so for the opponent, we simply return the value not held by the **turn** variable.

```
public int CurrentPlayer => turn;

public int Opponent => turn % 2 + 1;
```

To have the Tic-Tac-Toe implementation work smoothly with the Minimax algorithm, we also implement the **IStaticEvaluation.Evaluate()** method.

```
public double Evaluate ( bool currentMaximizer ) {
    if ( _winner == CurrentPlayer ) return 1.0;
    if ( _winner == Opponent ) return -1.0;
    return 0.0;
}
```

Finally, we implement the **IDeepCopy** interface. This interface is used by the MCTS algorithm, especially during the simulation phase so as to not change the original game state or properties references in any way possible.

```
public TicTacToe DeepCopy() {
    var t = new TicTacToe();
    //shallow copy of struct(int in our case) is
    //fine since no reference is copied
    t.Cells = (int[,])Cells.Clone();
    t.turn = turn;
    return t;
}
```

We can now use the Minimax, Monte Carlo Tree Search, Minimax Alpha-beta pruning, Parallel Minimax Alpha-beta pruning, Random agents to play the game of tic-tac-toe. For example:

```
var tt = new TicTacToe();

//MinimaxABPruning<TPlayer, TMove, TGame>
var minimaxABagent = new MinimaxABPruning<int, int,
   TicTacToe>(...);
```

```
//MonteCarloTreeSearch<TMove, TGame, TPlayer>
var mctsAgent = new MonteCarloTreeSearch<int,
   TicTacToe, int>(...);

minimaxBestMove = minimaxABAgent.BestMove(tt,
   tt.turn).BestMove;
tt.Move(minimaxBestMove);

mctsBestMove(tt, tt.turn).BestMove;
tt.Move(mctsBestMove);
```

This way, we can play the Tic-Tac-Toe game between 2 smart or trivial agents until the game finishes.

6.4 Quoridor Game Implementation

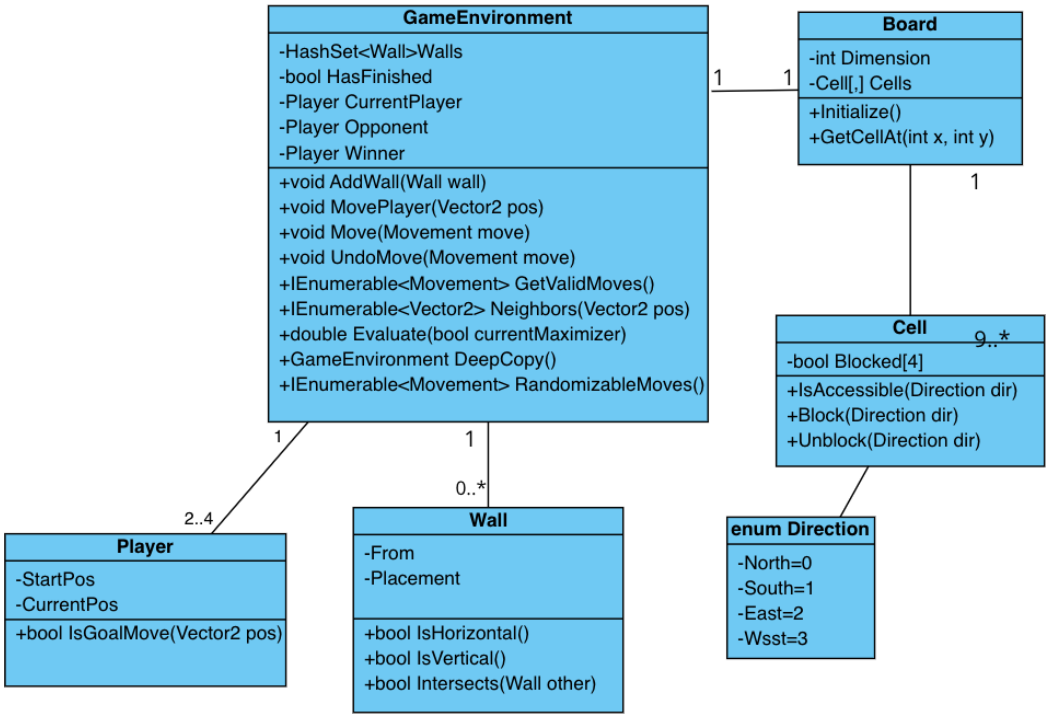


Figure 6.1: A UML diagram depicting relationship in the core library

As seen in Figure 6.1, the `GameEnvironment` class implements all interfaces necessary for AI integration.

In contrast to the **Tic-Tac-Toe** implementation we saw earlier in Section 6.3 where `int` type was used for the `TMove` parameter, we use a reference type `Movement` for the `TMove` parameter, since we need to consider wall placement and player movement, which is difficult to encode and decode as integral values.

```
public abstract class Movement{}

public class Vector2 : Movement{...}
```

```
public class Wall : Movement{...};
```

This approach makes it easier to identify movement types and therefore easily perform Move/Unmove operations on the game and much more.

We will now describe the core elements of the interfaces that we integrated for this game.

```
public IEnumerable<Movement> GetValidMoves() {
    List<Vector2> neighborMoves;
    //Populate the moves based on whether the
       neighboring
    //cells are accessible from the cell the current
       player
    //is in

    List<Wall> possibleWalls;
    //Populate the available wall list. We don't
       include
    //already placed walls/walls intersecting with
       already
    //placed walls

    return neighborMoves.Concat(possibleWalls);
}
```

Also, as described earlier, for the move and unmove operations, we do not need to decipher the movement by a pre-defined rule like we did in Section 6.3. We can easily check the underlying type of the abstract Movement type and do operations accordingly.

```
public void Move(Movement move) {
    if (move is Vector2 v2) {
        MovePlayer(v2);
    }
    if (move is Wall wall) {
        AddWall(wall);
    }
    //change turn
}
```

6.4.1 Project Structure

The solution consists of six fundamental projects, each written in C#.

- **Quoridor.Core**
  This library project contains the core game logic for Quoridor, and implements all interfaces to allow AI algorithms to run.

- **Quoridor.AI**
  This library project includes all the fundamental interfaces and a generic AI algorithms implemented using these interfaces.

- **Quoridor.Common**
  This library project includes all common helpers, such as XML parser helper, logging helper, etc.

- **Quoridor.Tests**
  This NUnit test project includes all unit tests for robust development.

- **Quoridor.ConsoleApp**
  A CLI tool that runs simulations and allows user to play against an opponent with a visual interface.

- **Quoridor.DesktopApp**
  A WinForms application that allows user to play against one another or against various AI.

These projects are inter-connected by references. For example from Figure 6.2, **Quoridor.AI** is a standalone library that contains all the interfaces, which is referenced by all other projects. All project dependency structures are depicted in Figure 6.2.

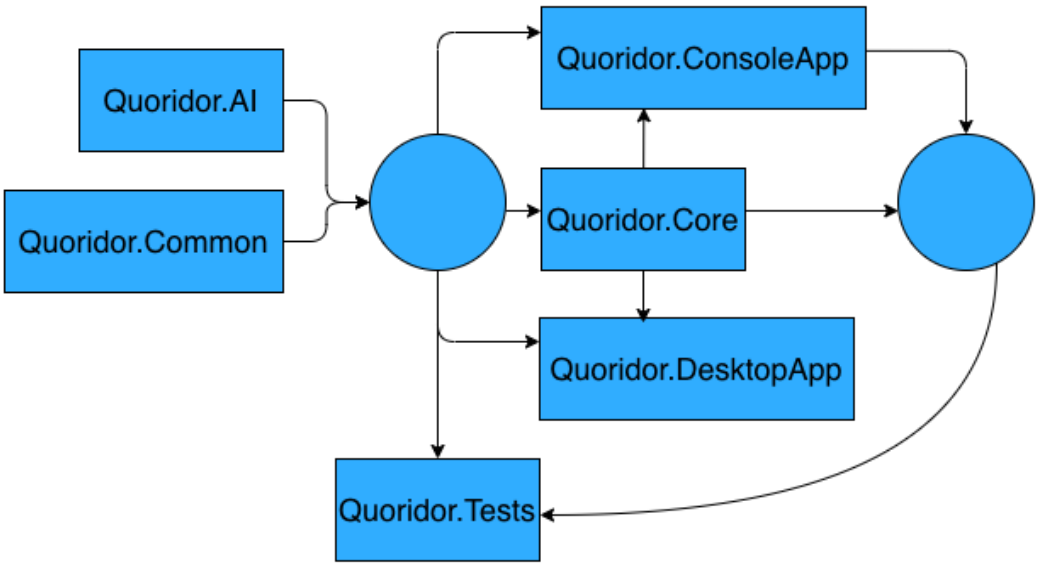


Figure 6.2: Quoridor project dependency diagram

Chapter 7

Experiments

In this chapter, we define our experiment, the parameters involved in it for different agents and implementations and the final results of the simulation based on the implementation. The main focus of this chapter will be to characterize the complexity for different implementations mainly in terms of average time per move with different implementations and the result of tournament considered between the different agents. The main reason for the experiment is to validate our implementation and to compare the implemented agents including the random and semi-random agent baselines.

7.1 Usage

We use the console application to run all our experiments.

As an example, we we provide the following arguments to the console application simulate 100 games between Parallel-Minimax with a depth of 1 and Semi-Random agents on a game board of dimension 5x5:

```
$dotnet run play -s1=parallelminimaxab -s2=semirandom
   -dimension=5 -depth=1 -sim -numsim=100
```

The aforementioned command produces the following output:

```
===Results===
Player A : Parallel MinimaxAlgorithmABPruning won
   99/100 games. Win rate : 99%. Average move time(ms)
   : 3.04
Player B : Semi-Random won 1/100 games. Win rate : 1%.
   Average move time(ms) : 1.05
Toal moves made across 100 games : 1781
Average total moves per game : 17
```

From the result above, we see that Parallel Minimax took **1781** moves across 100 games, with an average time per move of **3.04 ms** per move, and with an average win rate of **99%**.

In this thesis, the average time per move complexity can be defined as the average time taken by the agent to go through each move. The average is performed with respect to the total number of moves made by a particular agent in a game.

7.2 Minimax depth tuning

We first consider the implementation of the Minimax algorithm with the depth limited version, alpha-beta pruning version and the parallel version for different Quoridor board dimensions and demonstrate the time complexity of the implemented agents in terms of average time per move.

For the following experiment, we run all of the 3 implemented variants of the Minimax algorithm including the alpha-beta and the parallel variant, we use the **Quoridor.ConsoleApp** console application, and simulate 100 games with different board and agent configurations against the semi-random strategy - a total of 36 times (3 variants of minimax x 3 depth x 4 dimensions).

]

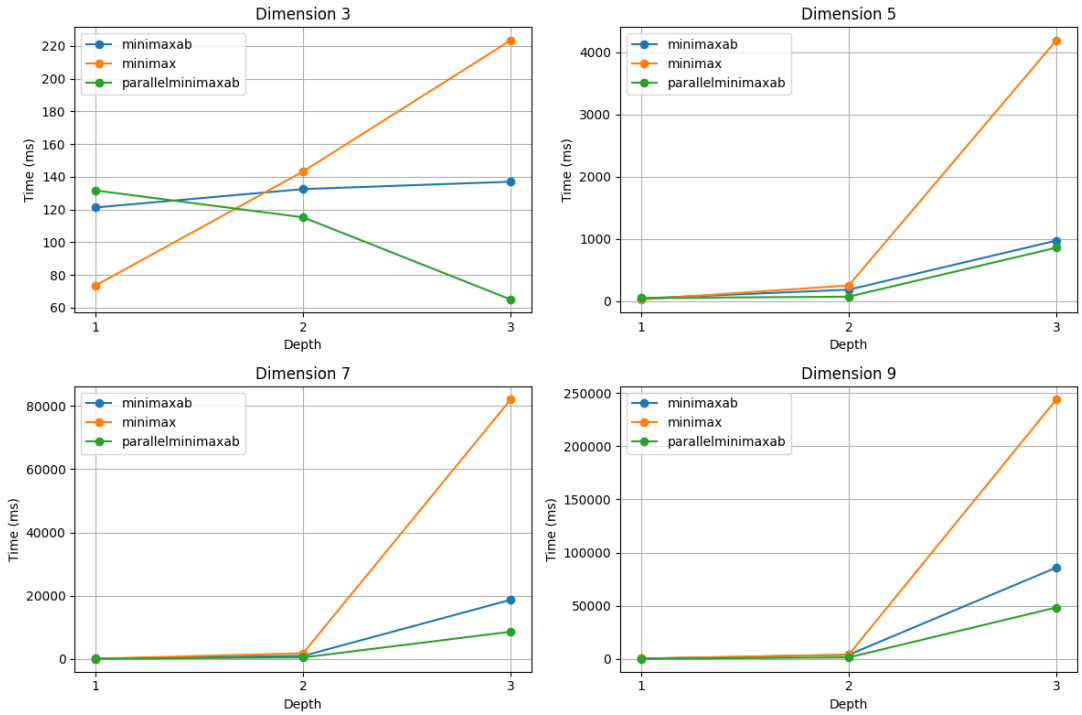


Figure 7.1: Average move time comparison between all 3 Minimax variants for different board sizes

In Figure 7.1, we can see the average time per move of the Minimax algorithm along with its optimized variants on different board dimensions. In the figure, the dimension refers to the board dimension. For e.g., dimension 3 refers to a board size of 3x3 and so on. Considering the board of large dimension (e.g., 7 and 9), without further optimization, the complexity of implementing the algorithm for Quoridor is prohibitively high. For example, for depth 3, the move time complexity for the board with dimension 7 is 82 seconds and that for dimension 9 is 244 seconds.

From the figure, it can be inferred that for depth 1, the minimax algorithm without any further optimizations including alpha-beta or parallel implementation performs the fastest. The reason for this is straightforward, as for depth 1, the algorithm only evaluates one further move. In such case, there is not further improvement with alpha beta as we anyways need to evaluate all the nodes. The

additional condition checking for the alpha beta implementation for this depth increases the average time per move without any gains. Similarly, the complexity of implementing the parallel algorithm outweighs the simple sequential evaluation hence increasing the time complexity per move without any impact on the performance.

However, for depths more than 1, it can be seen that the time complexity for each move considering alpha-beta pruning and further parallel implementation of the algorithm significantly improves as we increase the considered depth. Below, we have a table showing the improvement of the time complexity considering the minimax algorithm without any further optimization as the baseline.

	Depth 1	Depth 2	Depth 3
Dimension: 3x3			
Minimax	1	1	1
Minimax alpha-beta	1.8	0.92	0.61
Minimax alpha-beta parallel	1.79	0.8	0.29
Dimension: 5x5			
Minimax	1	1	1
Minimax alpha-beta	1.82	0.72	0.23
Minimax alpha-beta parallel	2.06	0.27	0.20
Dimension: 7x7			
Minimax	1	1	1
Minimax alpha-beta	1.29	0.53	0.22
Minimax alpha-beta parallel	1.05	0.25	0.10
Dimension: 9x9			
Minimax	1	1	1
Minimax alpha-beta	1.06	1	0.35
Minimax alpha-beta parallel	0.43	0.33	0.19

Table 7.1: Relative move-time complexity compared to the baseline minimax algorithm

In Table 7.1, we present the relative move time complexity of the different algorithms with reference to the minimax algorithm for the same depth. Here, we can see that, especially for depth 2 and depth 3, the time minimax algorithm when using the alpha beta pruning and further parallel implementation reduces significantly. This is especially relevant when considering the large dimension of the Quoridor board as the move time complexity is significantly high. Here, optimization is required for running the algorithm in relatively manageable time.

7.3 Experiment 1 : 3x3 board

In this section, we perform various simulations to find out the best parameters for the MCTS algorithm, and use those to run head-to-head simulations between all agents to find out the best performing one for the 3x3 game board.

7.3.1 MCTS parameter tuning

The main aim of this experiment is to tune the parameters in the MCTS algorithm, namely the number of iterations and the exploration parameter. Given a relatively small size of the board, each move has a significant impact on the result, where a single miscalculation can lead to an immediate loss. For example, Figure 7.2a depicts the initial game state for the 3x3 configuration. Player A makes a move `b2` resulting in the game state depicted by Figure 7.2b. Player B now can immediately win with the move `a2` (it jumps over player A). Considering this, we choose a large number of game iterations, **500** for MCTS. Higher iteration count allows for a comprehensive evaluation of the game states, accounting for the strategic depth required in the tight confines of a 3x3 board where tactical blunders are less forgivable and can lead to a loss, allowing the algorithm to recognize and avoid simplistic heuristics that may overlook such critical missteps as depicted in figure 7.2

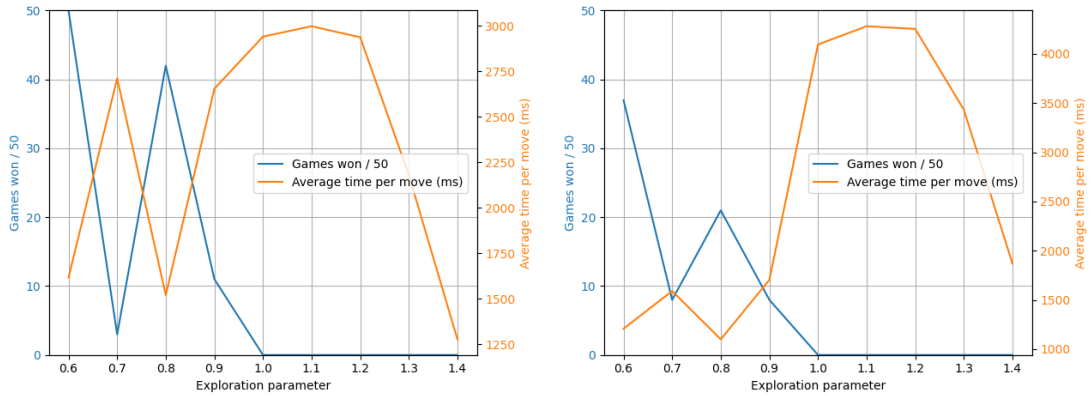
(a) Initial game state

(b) b2 move by player A

Figure 7.2: First bad move by player A leading to a (possible) loss

Using **500** iterations, we conduct an experiment by playing the MCTS agents against two different opponents, A* and Minimax with varying exploration parameters. In this experiment, we consider the minimax algorithm with depth of 2 and simulate 50 games for each exploration parameter. We vary the exploration parameter from 0.6 to 1.4 with a interval of 0.1. As we see on Figure 7.3a, MCTS algorithm with an exploration parameter of **0.6** won **100%** of the games against the **A*** algorithm and an average of **74%** against the **Minimax** algorithm (with a depth of 2).

Furthermore, as depicted by the graphs in Figure 7.3, MCTS algorithm loses all games against the two opponents when the exploration parameter is greater than 1. A parameter greater than 1 excessively prioritizes **exploration** of states with very few simulations, leading to a neglect on promising strategies already discovered by the MCTS algorithm. This can cause the algorithm to overlook immediate threats or miss direct paths to victory, as evidenced by the total losses incurred at higher parameter values on Figure 7.3. In contrast, an exploration parameter less than 1 indicates a balanced approach that weighs the value of **exploiting** well-performing moves against the potential benefit of exploring new moves. This balance is particularly critical on the small 3x3 board, where each decision carries significant weight. Therefore, an exploration parameter value below 1 is preferred for a 3x3 Quoridor board, as it ensures that the algorithm



(a) MCTS vs A*

(b) MCTS vs Minimax

Figure 7.3: Average time per move and win rates of MCTS agent running 500 iterations with varying exploration parameter for board size 3x3

maintains a focus on exploiting successful strategies while still considering any potential novel moves, leading to more consistent and strategic game play.

7.3.2 Results

In Table 7.2, we present the tournament result for the agents for the board dimension 3x3 and present the results. For the simulation, we ran the experiment 1000 times between different agents. For the minimax agent, we considered the depth of 2. For MCTS agent, we considered the determined optimal exploration parameter of 0.6 and 500 MCTS simulations per move.

A value $V_{i,j}$ for agents i and j refers to the average win rate out of 1000 simulations with agent i as player A and agent j as player B.

With these parameters, we can now compare various agents on a board of dimension 3x3.

p2 / p1	Random	Semi-Random	A*	Minimax	MCTS
Random		31.3	6.6	5.3	0
Semi-Random	73.5		11.5	10.2	0
A*	92.2	90.2		0	0
Minimax	92.4	88.9	8.2		0
MCTS	90	74.3	100	74	

Table 7.2: Agent comparison for dimension 3x3. The values in the table represent the total win percentage of the agent in the row of the table when competing against the agent in the column of the table

7.4 Experiment 2 : 5x5 board

Following the results of the 3x3 board experiment, as presented in Table 7.2, we would also like to find out the ideal MCTS parameters for the 5x5 variant and

run a head-to-head tournament betweeen all our implemented agents.

7.4.1 MCTS parameter tuning

For this experiment, we consider a Quoridor board of size 5x5 and play against a Minimax agent with a depth 2 and like the previous scenario, perform 50 experiments per parameter we are interested in.

First, we perform a grid search to find the ideal number iterations for the 5x5 board, using average win-rate as our deciding factor. We iterate over a range of 30 to 240 with an interval of 30 and run 50 simulations, all while using a theoretical optimal exploration parameter of $\sqrt{2}$ [Kocsis and Szepesvári, 2006].

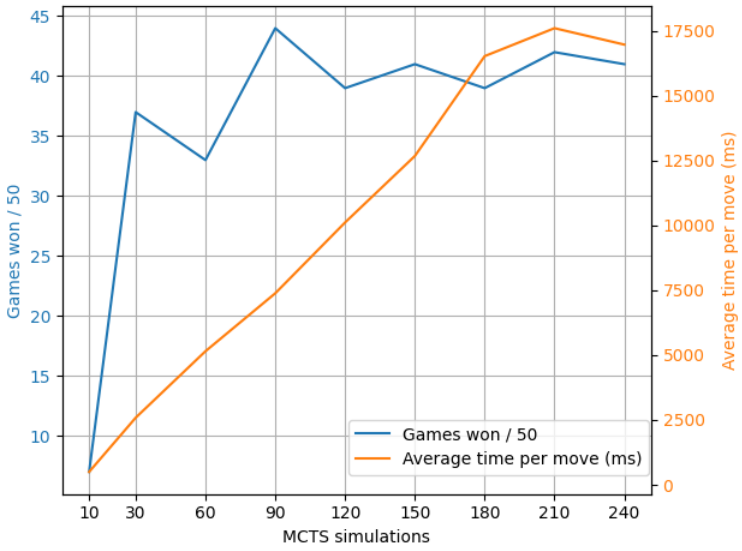


Figure 7.4: Average time per move and win rates of MCTS agent with varying simulations for board size 5x5

In Figure 7.4, we see that after 90 simulations, the additional marginal number of game won does not increase, although the time per move increases. Hence, optimizing the number of simulations can be important to limiting the complexity of the MCTS agent.

Likewise, in Figure 7.5, we use 90 iterations we deduced from Figure 7.4 to find the optimal exploration parameter based on the number of games won varied with different exploration parameters. From Figure 7.5, we see that with an optimal exploration parameter of **1.7**, the win rate for the agent is high i.e 46 wins out of 50 games.

7.4.2 Results

Firstly, we can observe that the random, as expected, does poorly against all the agents. However, in the 3x3 board, the random agent still wins a third of the time again semi-random agent and a few times against the even the A-star and the minimax agents. From the table, it can also be observed that the MCTS agent performs the best in the 3x3 board as it almost does not lose any games against the any other agents.

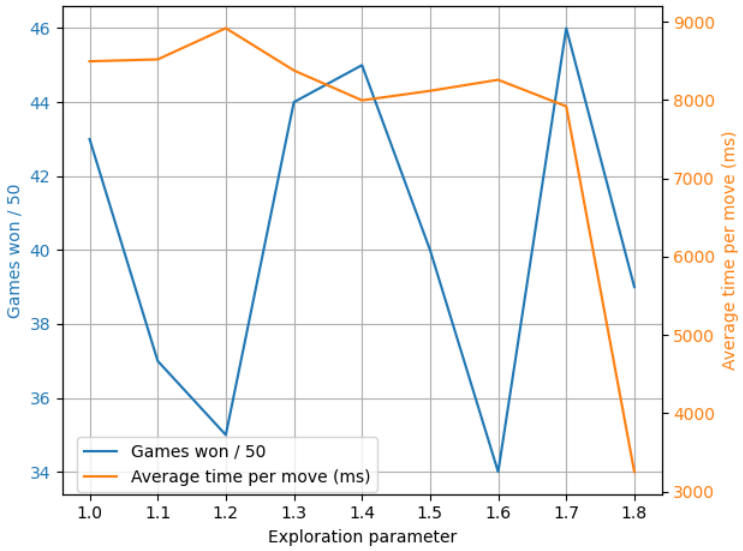


Figure 7.5: Average time per move and win rates of MCTS agent with varying exploration parameter for board size 5x5

p1 \ p2	Random	Semi-Random	A*	Minimax	MCTS
Random		5.8	0	0	0
Semi-Random	93.6		0.3	1.9	3.3
A*	100	99.7		22.7	73.3
Minimax	99.9	99.4	87.3		33.3
MCTS	100	100	56.7	92	

Table 7.3: Agent comparison for dimension 5x5. The values in the table represent the total win percentage of the agent in the row of the table when competing against the agent in the column of the table

In Table 7.3, we present the result of the tournament when considering a Quoridor board of dimension 5. For this board dimension, we again consider the total number of simulation runs as 1000. For the minimax agent, we considered the depth as 2. For the MCTS simulation, we considered the determined optimal exploration parameter of 1.7 and ran 90 simulations per move.

From the table, we can observe that considering a board of larger dimension compared to before, we can see that both random and the semi-random agents performs poorly in a larger board. Since the dimension of the board is larger, the random and semi-random agents require a large number of good moves in a sequence to win and this is less likely when the required number of moves is high. From the table, we can also see that the MCTS algorithm performs best against all the algorithms winning more than 90% of its games except for the A-star agent where it wins more than half of its games. Another surprising factor is that the A-star algorithm which only considers the distance from the start position and end position as the main cost functions performs much better than the minimax algorithm.

7.5 Experiment 3: 7x7 board

For the MCTS agent, since the average move time was 34.4 seconds with the available simulation hardware, we could only perform 100 simulations. Additionally, the simulation run time also prevented us from using an optimized parameter. Hence, we used the theoretical optimal parameter of $\sqrt{2}$ [Kocsis and Szepesvári, 2006] for this dimension. For the minimax agent, we considered the implementation with depth 2.

p1 \ p2	Random	Semi-Random	A*	Minimax	MCTS
Random		0.2	0	0	0
Semi-Random	99.6		0.2	0.7	0.2
A*	100	99.7		18.2	2
Minimax	100	99.3	79.8		58
MCTS	100	100	3	44	

Table 7.4: Agent comparison for dimension 7x7. The values in the table represent the total win percentage of the agent in the row of the table when competing against the agent in the column of the table

In Table 7.4, we illustrate the result of the tournament when considering a board of dimension 7x7. For this dimension we also ran 1000 simulations for all the agents except for the MCTS agent.

Here, the performance of the random agent is even more significantly worse compared to the previous boards as it fails to win any games against A-star, minimax and MCTS agents. Unlike the previous smaller boards, in this instance, the minimax algorithm performs much better against A-star algorithm as it wins almost all of its games. From this, we can conclude that the strategical play with the minimax algorithm for larger board is much more significant than the simple cost function oriented traversal strategy of the A-star algorithm. Likewise, the MCTS algorithm wins almost 3 out of every 4 games both when starting first and starting second.

From the above tournament simulation presented in Table 7.4, we can conclude that the board dimension does have an effect on the strategy that might be used. For board of smaller dimensions such as 3 and 5, simpler strategies such as random and semi-random in some cases can be enough to get some wins against more strategical agents such as the minimax agent. However, as the board dimension gets higher such as 7 and 9, the required strategy is very prominently visible as semi-random and random agents win almost no games against the other agents.

Chapter 8

Conclusion

In conclusion, in this thesis, we considered a zero-sum strategy game of Quoridor and implemented various AI agents for the game. The main focus of our work was to create a generic interface based implementation for the game such that the agents could be seamlessly used for other games as well, and perform experiments on the game of Quoridor to show how to find the ideal parameters for the Minimax and the Monte Carlo Tree Search algorithm.

In Chapter 2, we formalized the notations for the game including the cells of the game board, movements and wall placements and using them, we formally described the rules of the game. Additionally, we also proposed an improvement upon the notation for wall placement to remove the ambiguity on the notations used in the current literature.

In Chapter 4, we described the game tree of the game Quoridor. We provided metrics to quantify the complexity of the game in terms of state space and game tree complexity. During our analysis, we determined that the state space complexity increases exponentially as a function of the board dimension and is dominated by the complexity of wall placement. We also approximated the game tree complexity in terms of average branching factor and the average depth. Finally, we compared the complexities against other popular games such as Tic-tac-toe, Chess, Go and Connect-Four and determined Quoridor of dimension 3 has a complexity comparable to that of Tic-tac-toe, Quoridor of dimension 9 has complexity comparable to Chess.

In Chapter 5, we explained the background and the algorithm behind the AI agents implemented in this thesis including the minimax algorithm with alpha-beta pruning and parallel implementation, MCTS algorithm and the A-star algorithm.

In Chapter 6, we explained our implementation of the interfaces which are the fundamental part of our work. Further, we explained the various agents that we have implemented including the minimax agent, Monte-Carlo agent and A-star agent for Quoridor using the interfaces. We further illustrated with an example of another strategy game Tic-tac-toe how the interfaces could be seamlessly integrated as agents for other games as well.

Finally in Chapter 7, we presented the numerical simulations of our game considering various aforementioned agents. We simulated for the time complexity per move as a metric of quantifying the complexity of the game for different agents and the results of the agents playing against each other with varying

board dimensions. From the tournament, we showed that as we increase the board dimension, the more strategical agents such as the minimax and the MCTS win almost all of their games against more simpler agents such as random, semi-random and A-star.

Bibliography

Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

Matyáš Brenner. Artificial intelligence for quoridor board game. 2015.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.

Shih-Chieh Huang, Rémi Coulom, and Shun-Shii Lin. Monte-carlo simulation balancing in practice. In *Computers and Games: 7th International Conference, CG 2010, Kanazawa, Japan, September 24-26, 2010, Revised Selected Papers 7*, pages 81–92. Springer, 2011.

Takuro Iwanaga and Makoto Sakamoto. Analysis of 5x5 board quoridor. 2022.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

Lisa Glendenning. Mastering Quoridor. 2002. URL `https://api.semanticscholar.org/CorpusID:18564930`.

Peter JC Mertens. A quoridor-playing agent. *Bachelor Thesis, Department of Knowledge Engineering, Maastricht University*, 2006.

Quoridor Strats. The Official Quoridor Notation, Sep 2014. URL `https://quoridorstrats.wordpress.com/notation/`.

Glen Robertson and Ian Watson. A review of real-time strategy game ai. *Ai Magazine*, 35(4):75–104, 2014.

Rafael Moreira Savelli and Roberto de Beauclair Seixas. Tic-tac-toe and the minimax decision algorithm. *Luiz H. de F., Waldemar, C., ve Roberto L.(Eds.), Lua programming gems*, pages 239–245, 2008.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Geoff Skinner and Toby Walmsley. Artificial intelligence and deep learning in video games a brief review. In *2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS)*, pages 404–408, 2019. doi: 10.1109/CCOMS.2019.8821783.

AlphaStar Team. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind blog*, 24, 2019.

Thomas S Ferguson. *A Course In Game Theory*. World Scientific, 2020.

J v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100 (1):295–320, 1928.

Acronyms

AI Artificial Intelligence.

C# C Sharp.

MCTS Monte-Carlo tree search.

RTS real-time strategy.

Appendix A

Attachments

A.1 First Attachment