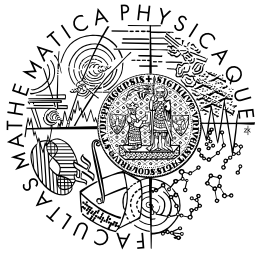


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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 2023

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 with alpha-beta pruning, 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 (**?**), 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 Chess, Go, 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 (**?**) 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 (**?**) and strategy games such as Go (**?**). Recently, DeepMind's Alpha Go for Go, Alpha Zero for Chess (**?**) and AlphaStar for StarCraft (**?**) have widened the gap between the AI and human intelligence even further.

In this thesis, we have chosen Quoridor as the startegy game to analyse and implement an agent of. 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.

1.1 AI algorithms

In this section, we give a brief overview of different AI techniques that have been considered in this thesis.

1.1.1 Minimax algorithm

Minimax algorithm, first proven by John von Neumann in 1928 in his paper *Zur Theorie Der Gesellschaftsspiele* (**?**), 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 (**?**).

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.

1.1.2 Monte-Carlo Tree Search

Monte-Carlo tree search (MCTS) (**?**) 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.

1.1.3 A-star algorithm

A star is a popular algorithm (**?**) 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.

1.2 Related Works

Compared to some other strategy games such as Chess and Go, Quoridor has not been extensively studied in the literature. In (**?**), the author developed an agent for playing Quoridor using genetic algorithm to optimize the weights. The authors in (**?**) study the complexity of the algorithm also develop a Quoridor playing agent based on Minimax algorithm. Likewise, the authors in (**?**) developed an MCTS approach for Quoridor. Recently, in (**?**), the authors performed an analysis of the game for a miniature 5 by 5 board.

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

- **Mastering Quoridor (?)** The author of the thesis implements 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.

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.

1.3 Thesis Outline

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 development of AI algorithms customized to 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 top-performing AI agent among a set of implemented AI agents including the minimax agent, 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, thereby bolstering the game's accessibility and inclusivity.

The structure of this thesis is as follows. In chapter **??**, we will introduce the formal notations of the game and based on it, formally explain the rules of the game. In chapter **??**, we will perform a game analsis from the perspective of game complexity including the state-space complexity and the game tree complexity. In chapter **??**, we will explain the implementation of the game interface and the AI agents. In chapter **??**, we will simulate the results of the game between the implemented AI agents and finally conclude the thesis in chapter **??**.

Chapter 2

Quoridor: Game description

The Quoridor game is 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 can be either played with two players or four players. 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 **??**, 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

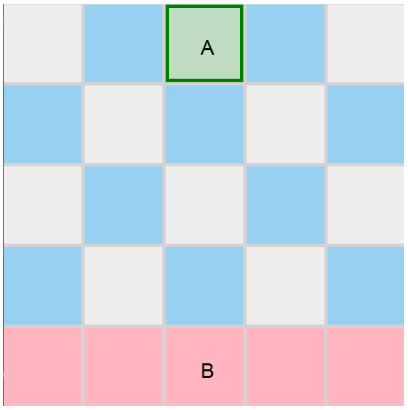


Figure 2.1: A screenshot of a 5x5 game board from our implementation

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** ((**?**)) and the **Quoridor Strat's Notation** ((**?**)). 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 $C_{i,j}$ represnting 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 **??**, 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(**??**), 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 **??** 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 **??** 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.

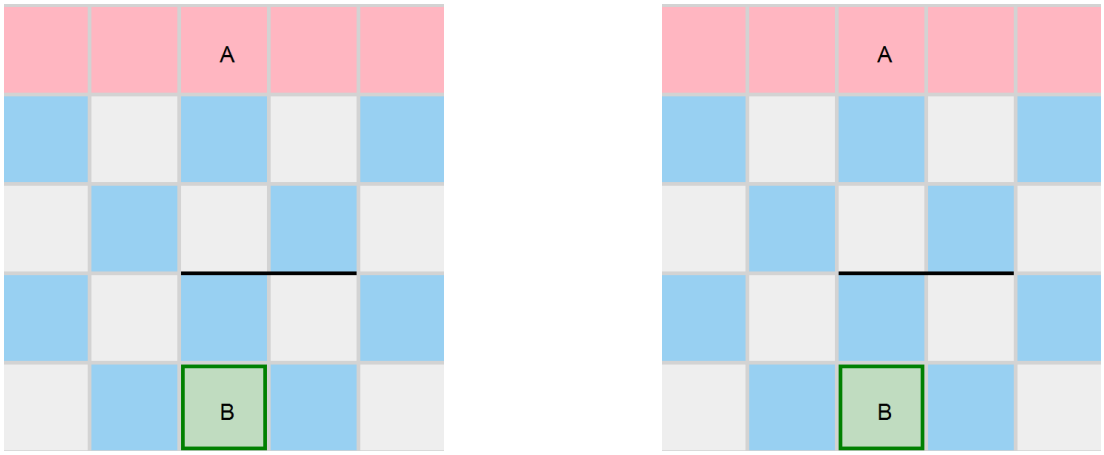
Similarly, after notation of the board and the movement, the notation for wall placement is another important component for Quoridor. The notation for wall placement is defined by the following equations:

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

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, the D in the equations (**??**) 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 represnetation even though the mathematical formulation is the same as presented in equation (**??**).

As seen in Figure **??**, 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 **??**. 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 **??**, the walls can now be represented by **c4N**, i.e. a Northern wall from the cell $C_{4,c}$.

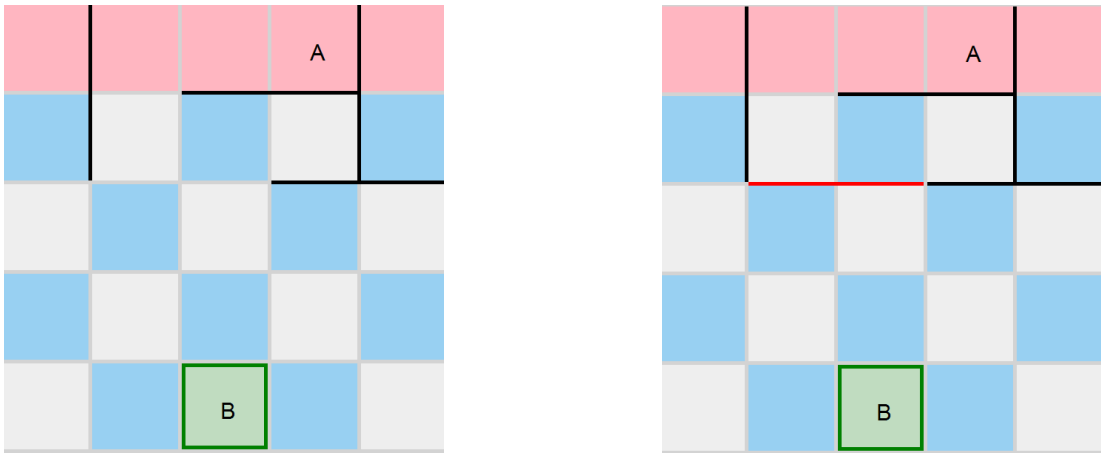
This notation provides an additional flexibility to represent the wall placement in the game and removes the ambiguity that may be present as we saw earlier.

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 **??**, 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 **??**, 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 **??**, 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 **??**, 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 game state

Figure 2.3: Example of an invalid wall placement

The game state represented by Figure **??** 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* **??**), the game state shown in Figure **??** is considered valid.

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

  - Two pawns cannot occupy the same square at a time. As mentioned above, for a player to move to a cell, the cell need to be empty.
  - 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 **??** 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_2, this is only possible if there are no walls in between them.
  - A Player is allowed to jump over multiple opponents (in case of more than 2 players) as long as they adhere to the aforementioned conditions.

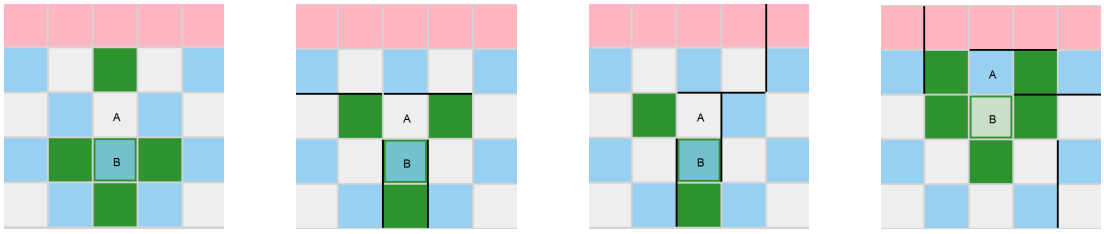


Figure 2.4: Examples of possible moves for player B in the game state

Chapter 3

Game Analysis

In Chapter **??**, 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.

3.1 Classification of Quoridor

Quoridor can be characterized as a discrete, deterministic, zero-sum, sequential, game with perfect information (**?**), and therefore, a combinatorial game (**?**).

3.1.1 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 **??**).

3.1.2 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.

3.1.3 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.

3.1.4 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.

3.2 Game-Tree

A game tree for an **abstract strategy game** 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 action/move.

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 **??** depicts a (partial) game tree for the Quoridor game with a board of size 3x3.

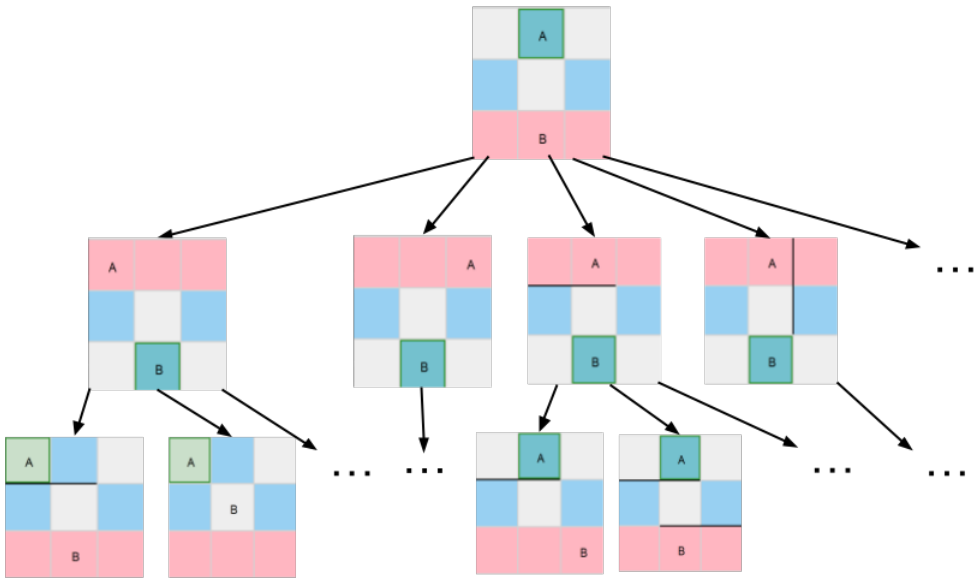


Figure 3.1: A partial game tree for a 3x3 game board.

As depicted in the figure, the first level of the game consists in player A located in the cell $C_{1,b}$ making a choice for the first move which may consist in the player either moving to a square or placing a wall to deter the opponent. The second level of the game then consist in player B

3.2.1 Branching Factor

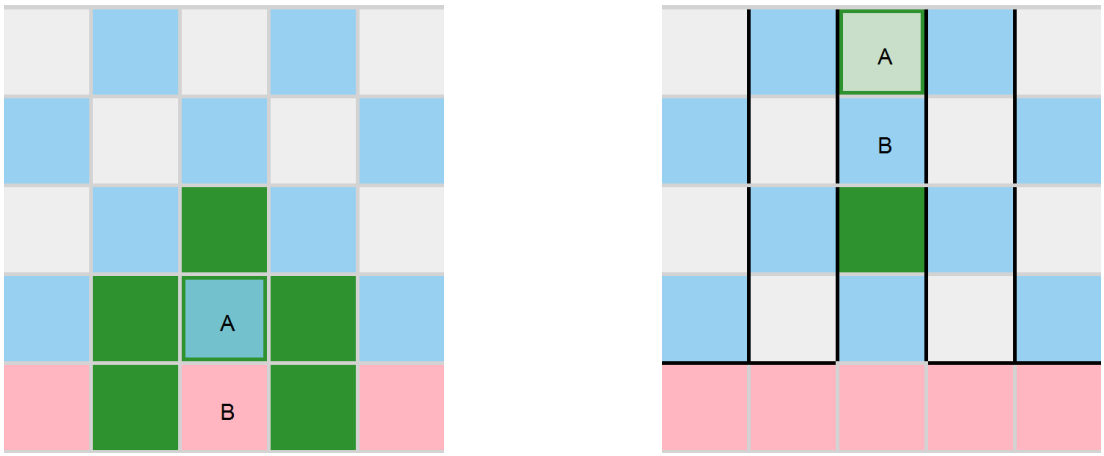
The branching factor of a **Game-tree** is the number of child nodes of each node, or in other words, the number of possible moves a player at their turn can make,

given the game state.

In figure **??**, 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 3.2: Branching factor differences

As depicted by Figure **??**, 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 place.

For a board of size NxN with no wall(s) 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:

$$2(N-1)^2 \tag{3.1}$$

Coming back to figure **??**, 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 **??** is **37**, which is also the maximum branching factor for board sized 5x5.

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

Average Branching Factor

In sub-section **??**, we saw that the branching factor is not uniform due factors like wall-placements and positioning of the player greatly influencing it.

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

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

To find an estimate of the average branching factor B_{avg}, we proposed the Algorithm **??**

Algorithm 1: Average branching factor

```
Function AvgBranchingFactor(agent1, agent2, simulations):
    input  : Two agents and number of simulations
    output: Average of averages 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 **??**.

From figure **??**, 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 (**?**) which was 60.

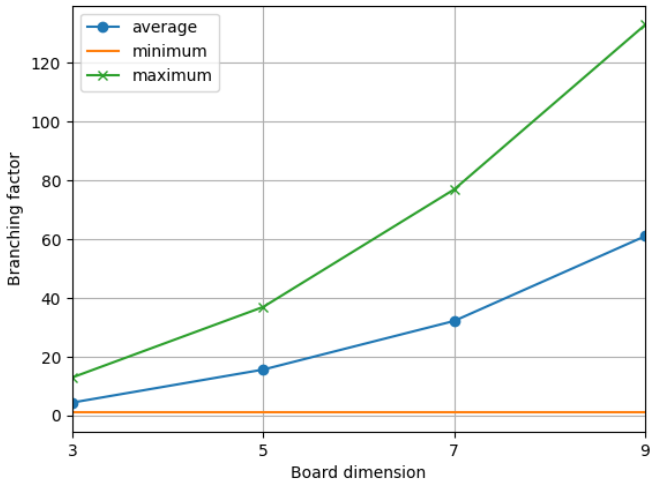


Figure 3.3: Branching factor for boards of different sizes

3.3 State space complexity

The state space complexity is a measure of a game complexity and refers to the total number of possible game states in a game. For example, for illustration with the game of tic tac toe, there are 9 squares and each square has a possibility of either an X or an 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 **??** and pawn movement rules **??**. Because of this, we loosen the regulations to come up with an upper-bound.

In (**?**), 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

$$S_p = N^2(N^2 - 1) \tag{3.2}$$

As for walls, from equation **??**, 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 assume that placing the wall anywhere in the board takes away the possibility of placing walls at 4 different places, even though this is not particularly true at the edges. 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, $N_W = 16$. 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:

$$L_{N_w} = \prod_{j=0}^{N_w} (2(N-1)^2 - 4j), \tag{3.3}$$

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

However, in the game there 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:

$$S_w = \sum_{i=0}^{N_w} \prod_{j=0}^{i} (2(N-1)^2 - 4j). \tag{3.4}$$

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

$$S = N^2(N^2-1) \times \sum_{i=0}^{N_w} \prod_{j=0}^{i} (2(N-1)^2 - 4j). \tag{3.5}$$

In figure **??**, 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.

3.4 Game tree complexity

The game-tree complexity of a game, as defined in (**?**), 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*".

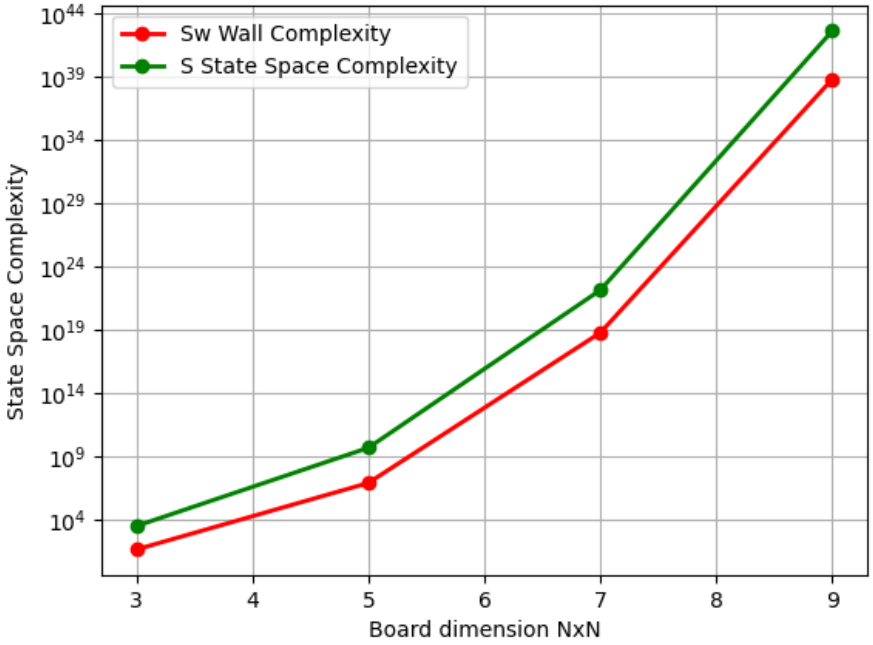


Figure 3.4: Complexity vs Board Dimension

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 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} (**?**), which is given by the following equation:

$$G = B_{avg}^{D_{avg}} \tag{3.6}$$

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:

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

3.5 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 3.1: State space, game tree and branching factor comparison between well-known games

In the Table **??**, 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 (**?**).

As we can see from the above table, the

Chapter 4

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.

4.1 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 **??**

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 **IAIStrategy<TMove, TGame, TPlayer>** interface, 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, **BestMove** (parametrized over generic type **TMove**) being the best move for the current player in the given game state and **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.

```
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.

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

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

Since we are focussed on abstract **Two-player**, Zero-sum 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.

```
TPlayer Simulation(TGame game)
{
    while(!game.HasFinished)
    {
        var move = _moveStrategy.BestMove(game,
            game.CurrentPlayer).BestMove;
        game.Move(move);
    }
    return game.Winner;
}
```

During the Simulation phase, the MCTS algorithm uses e.g the Random agent, which implements the **IAIStrategy** interface, therefore providing access to the aforementioned **BestMove** method, which requires **CurrentPlayer** to be passed to it.

```
void BackPropagation(Node<TMove, TPlayer, TGame> node, TPlayer winner)
{
    var current = node;
    while (current != null)
    {
        current.Count++;
```

```
        if (current.State.Opponent.Equals(winner))
            current.Wins++;

        current = current.Parent;
    }
}
```

Each node is expanded by making a move, and we keep a track of the player making the move in the node. After making the move in the game, we **Switch** the player turn. So the **CurrentPlayer** and **Opponent** gets switched too.

```
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 Parallel Minimax in order to ensure the original game state don't get changed by any means while exploring the game tree by making several moves.

```
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, in the Simulation phase of the MCTS algorithm we described above, we see the usage of the **HasFinished** property. The Simulation phase performs moves until the game reaches a terminal state.

```
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 **Minimax** algorithm to statically evaluate game states instead of exploring them when the specified depth has been reached.

```
MinimaxStep(TGame game, bool maximizingPlayer, int depth)
    if (depth <= 0 || game.HasFinished)
    {
        return game.Evaluate(maximizingPlayer);
    }
```

As we see from the base case of the recursive **MinimaxStep** method, we statically evaluate game states instead of recursing further and later compare them to find the best move.

```
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.

```
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.

```
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.

4.2 Agents

In this section, we discuss the generic AI agent 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.

4.2.1 Random Agent

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 given by:

```
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.

4.2.2 Semi-Random agent

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 firstly 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 **??**, the Semi-Random algorithm in Quoridor would process and return the best move the following way:

```
//IRandomizable<TMove>
//No pawn placement positions here
unplaced_walls = game.GetRandomizableMoves();

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

possible_moves = unplaced_walls + [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.

4.2.3 Minimax Agent

Minimax agent is another agent that we have implemented in the game that provides the move to the agent based on minimax algorithm.

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

$$\bar{v}_i = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}) \tag{4.1}$$

where,

$$i, -i = \text{index of the player of interest, opponent respectively} \tag{4.2a}$$
$$a_i, a_{-i} = \text{action of the player of interest, opponent respectively} \tag{4.2b}$$
$$v_i = \text{value function of player i} \tag{4.2c}$$
$$\bar{v}_i = \text{minimax value of the player of interest} \tag{4.2d}$$

As defined in the equation (**??**), 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 **??** consists of nodes that define either player's or the opponent's actions. For each action, there can be an evaluation where higher evaluation may mean it 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.

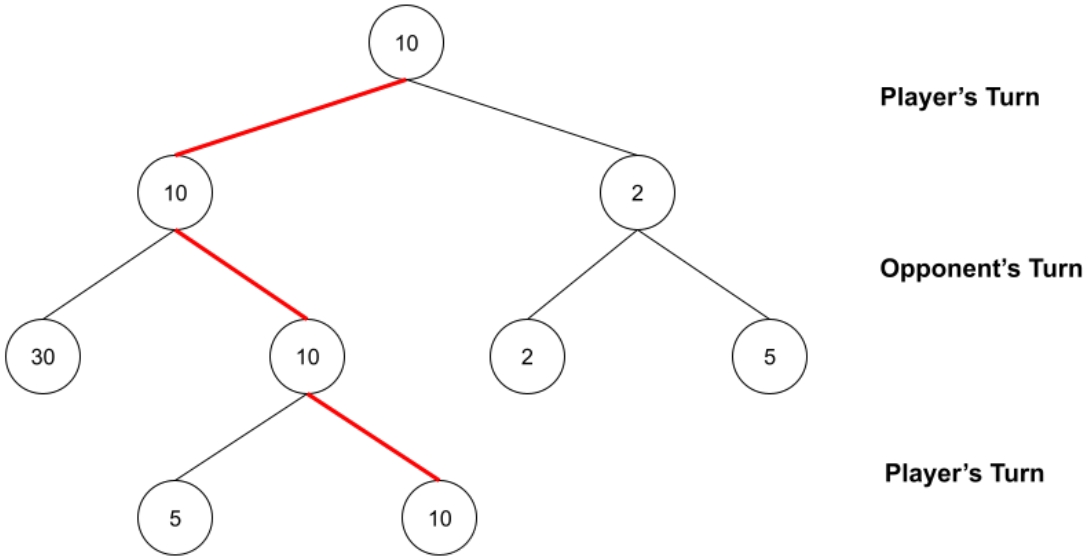


Figure 4.1: Figure illustrating an exemplary game tree and decision based on minimax algorithm

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.

Explain evaluation function for minimax and link it to the graph

As the graph of a game can be large, one way to limit the search space for implementing the minimax algorithm is consider a depth-limited version of the algorithm. In this version, the algorithm only considers a part game tree with limited depth of the graph. Increasing the depth increases the accuracy of the decision at the cost of computational complexity. In terms of game, this depth may be equivalent to "looking ahead only a few moves".

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

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 and beta values. The alpha value stores the minimum score the player is assured

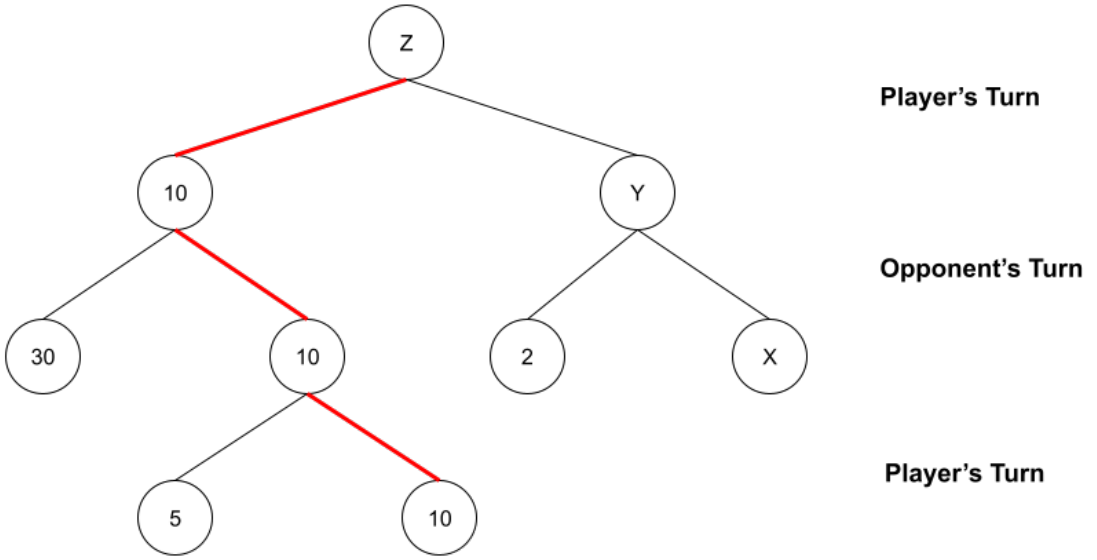


Figure 4.2: Exemplary figure illustrating the alpha beta pruing for minimax algorithm

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 **??**, if the player determines that

$$Z = max(10, Y) = max(10, min(2, X)), \tag{4.3}$$

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.

The minimax algorithm (and the alpha-beta pruning optimization of it) 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 would then be the following:

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

Below, we list the implementation of the minimax algorithm with alpha beta pruning implementation:

```
public TMove MinimaxStep(
    TGame game, int depth, bool maximizingPlayer, int alpha, int beta) {
```

```
    // return static evaluation if depth reached or game over
    // IStaticEvaluation

    bestMove = maximizingPlayer ? MinValue : MaxValue;

    // IValidMoves<TMove>
    foreach(var move in game.GetValidMoves())
    {
        // IMove<TMove>
        game.Move(move);
        result = MinimaxStep(game, depth - 1, !maximizingPlayer);
        // IMove<TMove>
        game.UndoMove(move);

        if (maximizingPlayer)
        {
            if (result.Value > bestMove.Value)
            {
                bestMove.BestMove = move;
                bestMove.Value = result.Value;
            }
            if (bestMove.Value > beta)
                break;

            alpha = Math.Max(alpha, bestMove.Value);
        }
        else
        {
            if (result.Value < bestMove.Value)
            {
                bestMove.BestMove = move;
                bestMove.Value = result.Value;
            }
            if (bestMove.Value < alpha)
                break;

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

We have implemented a recursive minimax agent with alpha-beta pruning that is depth limited. The algorithm alternates with the variable *maximizing-Player* 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

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.

The above defined methods to reduce the computational complexity of the minimax algorithm can be combined together. For example, the minimax algorithm can be depth limited and/or run with alpha-beta pruning and/or run in parallel as shown in the following pseudocode:

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

    var result = ParallelMinimaxStep(clonedGame, alpha, beta, depth -
        1, !maximizingPlayer);

    lock (_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.

4.2.4 Monte-Carlo tree search agent

MCTS is a heuristic algorithm for searching the game tree based on Monte-Carlo simulations. As shown in Figure **??**, 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. For example, in figure **??**, in the first player's turn, an action gives 3/6 chances of winning based on previous iterations whereas another gives

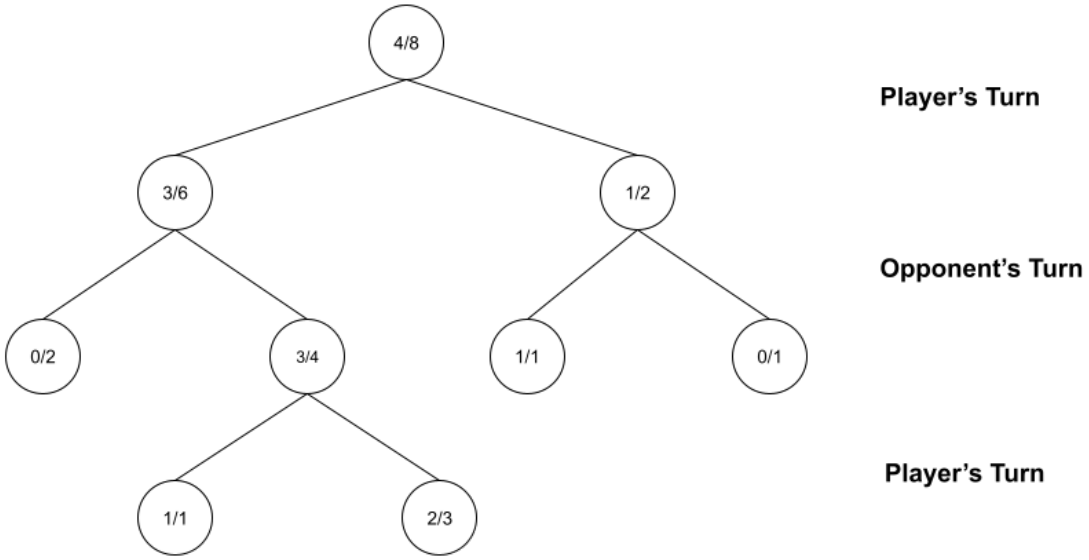


Figure 4.3: Figure illustrating an iteration of the MCTS algorithm

1/2 winning chances. The player selects a node that has highest possibility of winning, which in this case is equal. The player can also choose a new move that may be already explored to avoid not exploring further options.

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 **??**, an unevaluated 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 **??**, 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.

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.

```
public class MonteCarloTreeSearch<TMove, TGame, TPlayer> :
    IAIStrategy<TMove, TGame, TPlayer>
    where TGame : IMCTSGame<TGame, TMove, TPlayer>
    where TPlayer : IEquatable<TPlayer>
{
  ⋮

    public AIStrategyResult<TMove> BestMove(TGame game, TPlayer
        player)
```

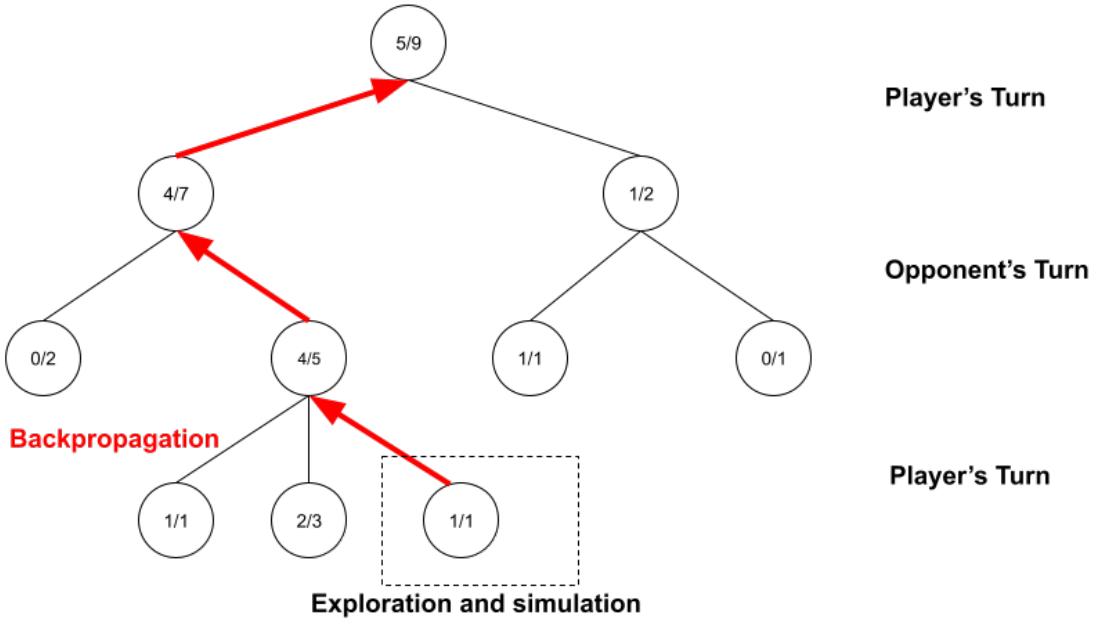


Figure 4.4: Figure illustrating steps 2, 3, and 4 of the MCTS algorithm

```
{
        ⋮
    for (int i = 0; i < _simulations; i++)
    {
        var selectedNode = Selection_Expansion(root);
        var winner = Simulation(selectedNode.State.DeepCopy());
        BackPropagation(selectedNode, winner);
    }
        ⋮
}

private Node<TMove, TPlayer, TGame>
    Selection_Expansion(Node<TMove, TPlayer, TGame> node)
{
        ⋮
}

private TPlayer Simulation(TGame game)
{
        ⋮
}

private void BackPropagation(Node<TMove, TPlayer, TGame> node,
    TPlayer winner)
{
        ⋮
}
```

As described in the above code snippet, for MCTS algorithm, the simulation is run *_simulations* times and in each step, the game runs the four described steps. The best strategy is defined based on the Monte-Carlo runs and the determined final weights in the graph.

4.2.5 A-Star agent

A-Star is a search algorithm for traversing from the source node to the destination node while minimizing the cost function. 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,

$$f = g + h \tag{4.4}$$

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 parametrized 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.
The **CalculateHeuristic(TMove move)** is the heuristic function - the cost of the cheapest path from the current position to the goal. In Quoridor, for example, we use **Manhattan Distance** as the heuristic.
Suppose player P is at at cell C_{xy} and player Q starts at cell C_{ab}, 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

$$H_P = |n - x| \tag{4.5}$$

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

$$H_Q = a \tag{4.6}$$

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>
```

```
public AIStrategyResult<TMove> BestMove(TMaze maze, TPlayer player)
{
    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.

4.3 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>** iterface. 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

$$Move(C_{ij}) = i + 3 * j \tag{4.7}$$

As an example, consider the cell $C_{1,2}$. From equation **??**, 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 repesented by equation **??**, 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 **??** to i and j values. To do so from given move $Move(C_{ij})$, we can use the following equations:

$$i = Move(C_{ij}) \mod 3 \tag{4.8}$$

$$j = \frac{Move(C_{ij})}{3} \tag{4.9}$$

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.

4.4 Quoridor Game Implementation

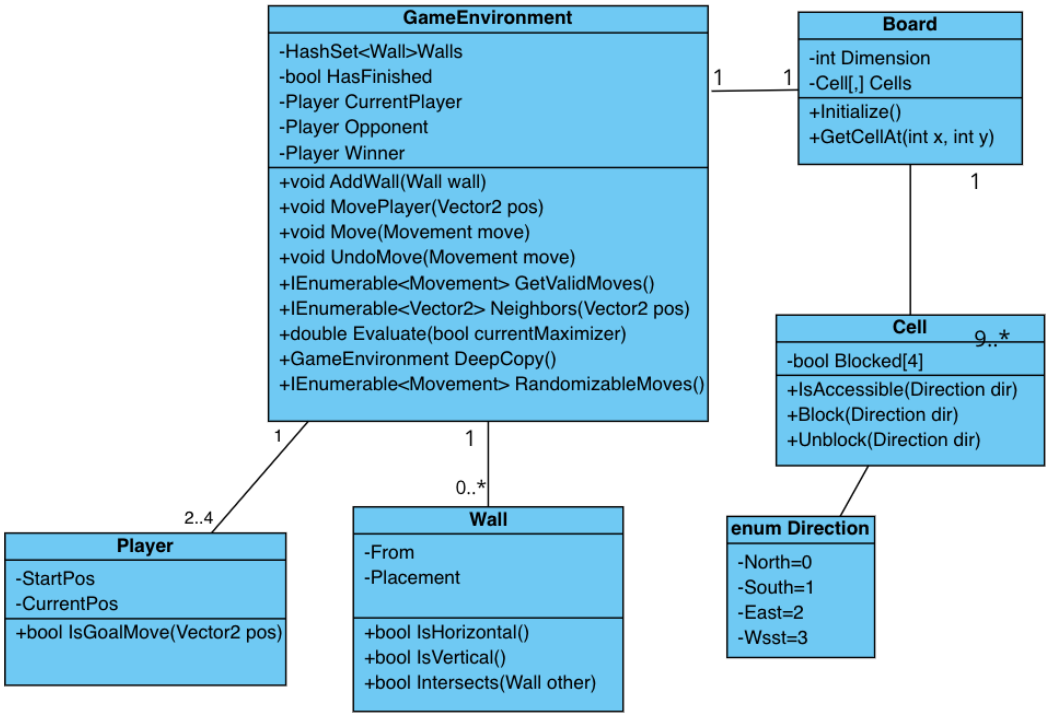


Figure 4.5: A UML diagram depicting relationship in the core library

As seen in figure **??**, the `GameEnvironment` class implements all interfaces necessary for AI integration.

In contrast to the **Tic-Tac-Toe** implementation we saw earlier in section **??** 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 **??**. 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
}
```

Assume a Quoridor game instance of 2 players P and Q, with P starting at cell C_{1c} and Q starting at cell C_{Nc}, where N is the dimension of the game board. Suppose P is at cell C_{xy} and Q is at cell C_{uv} at an arbitrary game state G_s, and let S_P be the shortest path from C_{xy} to C_{N*} and let S_Q be the shortest path from C_{uv} to 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, the static evaluation function for player for player P, F_P in game state G_s is given by:

$$F_P = S_P - S_Q + W_P - W_Q \tag{4.10}$$

So, for the static evaluation function, 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

4.4.1 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 **??**, **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 **??**.

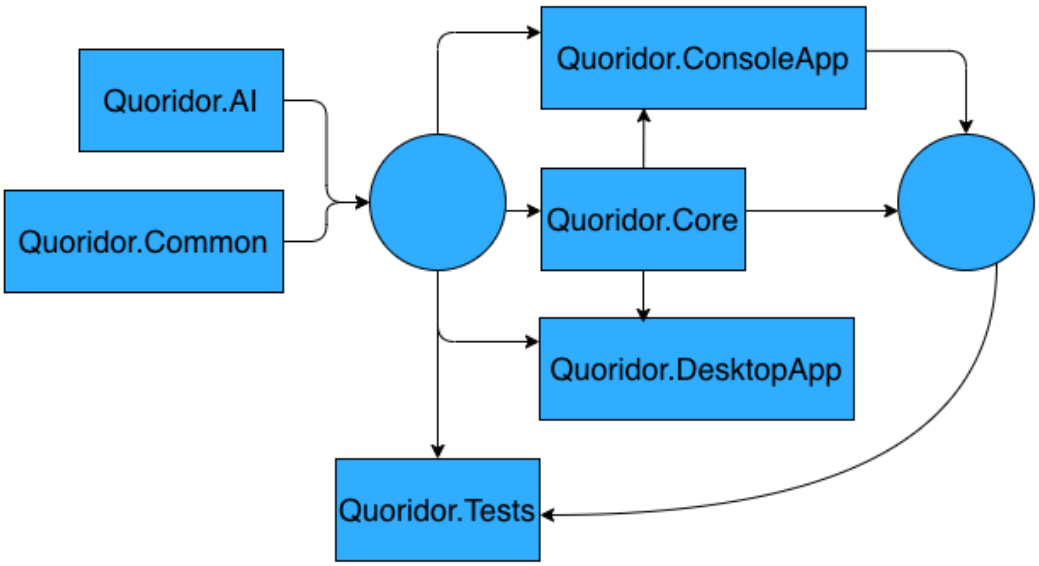


Figure 4.6: Quoridor project dependency diagram

Chapter 5

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 move time complexity with different implementations and the result of tournament considered between the different agents.

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 move time complexity.
To run the experiment on all 3 implemented variants of the Minimax algorithm, 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).

For example, we use the following command line arguments to 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 of **3.04 ms** per move.

In the figure **??**, we can see the average move time complexity of the implementation with different board dimension. Considering the board of large dimension (e.g., 7x7 and 9x9), 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.

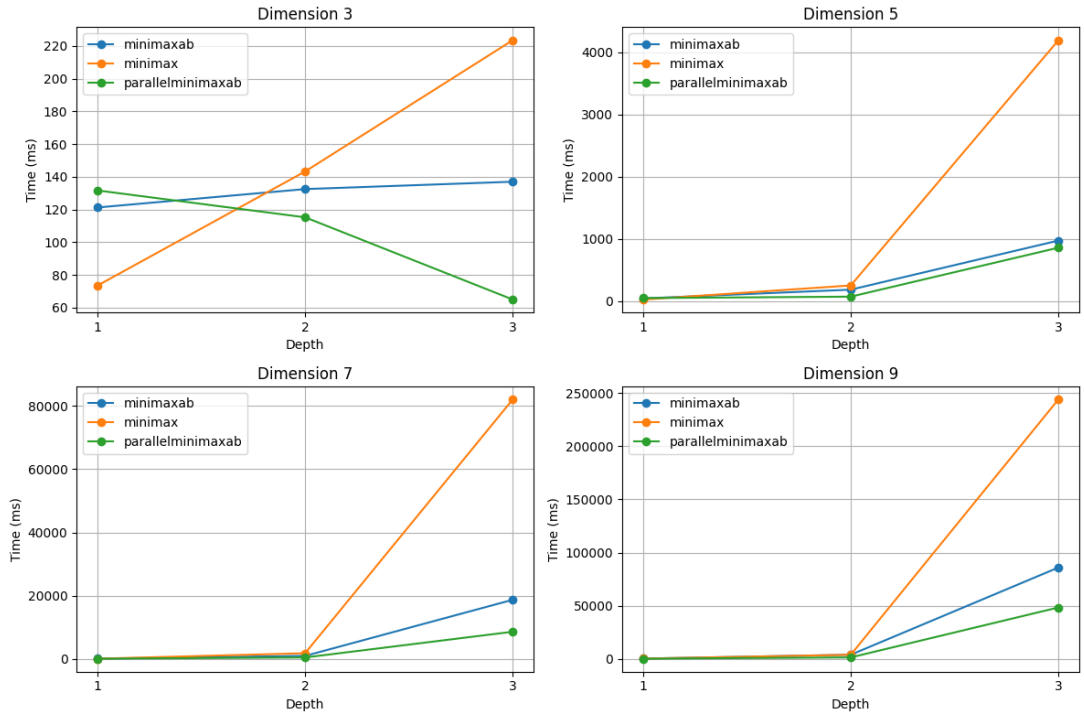


Figure 5.1: Average move time comparison between all 3 Minimax variants for different board sizes

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 best. 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. Similarly, the complexity of implementing the parallel algorithm outweighs the simple sequential evaluation.

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 implementation as baseline.

In the table **??**, 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 significant especially 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.

Following, we also performed the simulations with the MCTS agent. In the simulation, we consider a Quoridor board of dimension 5x5 and play against a Minimax agent.. As described in earlier sections, the MCTS algorithm requires multiple iterations through the game tree. More iterations provide a more reliable strategy at the cost of time complexity. This is even more prominent as we increase the dimension of the board due to the exponential increase in the state

	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 5.1: Table with the values of relative move-time complexity compared to the baseline minimax algorithm

space and game tree complexity as described in section **??**.

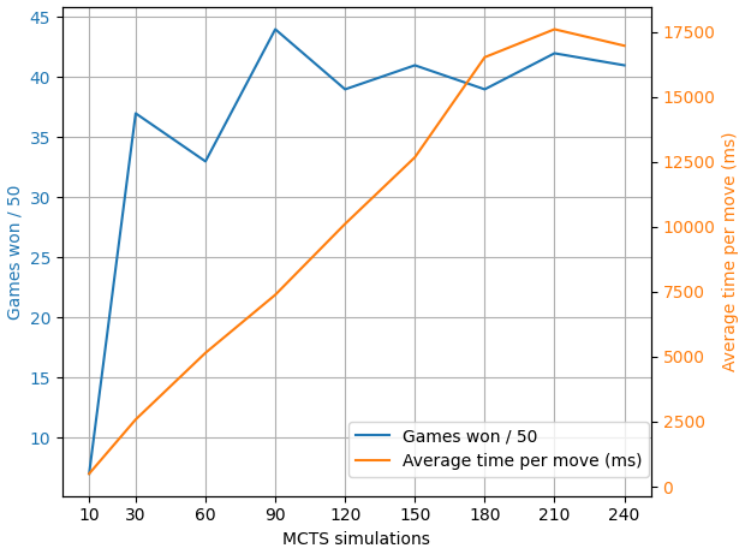


Figure 5.2: Average time per move and win rates of MCTS agent with varying simulations for board size 5x5

In figure **??**, we show variation of the number of games won with the MCTS agent againt the minimax agent varied with the number of game iterations. In the same figure, we also show the average time per move required when varying game simulations with the number of MCTS simulations. For each number of varied MCTS simulation, we performed 50 game simulations. We can see that after 90 simulations, the additional marginal number of game won does not in-

crease, although the time per move increases. Hence, optimizing the number of simulations can be important to limiting the complexity of the MCTS agent.

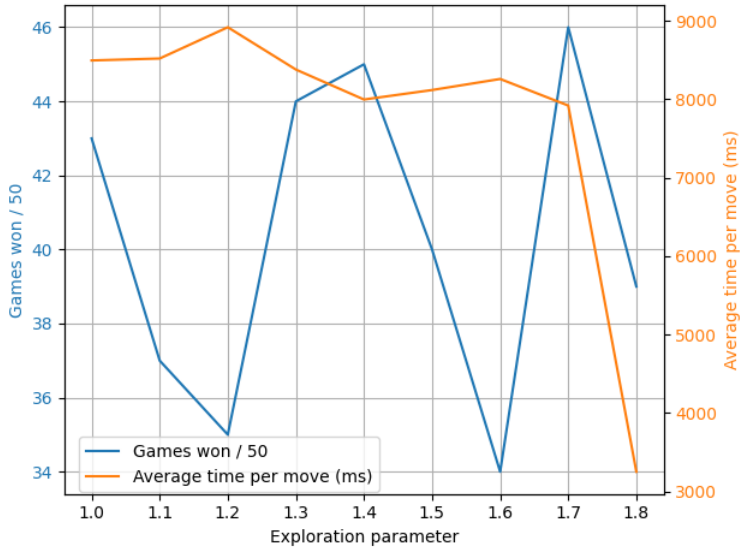


Figure 5.3: Average time per move and win rates of MCTS agent with varying exploration parameter for board size 5x5

Likewise, in figure **??**, we consider find the optimal exploration parameter based on number of games won varied with different exploration parameters, In MCTS, the exploration parameter is used in the exploration step to explore the nodes not visited yet. In addition, in the figure, we also show the average move time for different exploration parameters. In the figure, 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.

	Random	Semi-Random	A*	Minimax	MCTS
Random		31.3	6.6	5.3	0
Semi-Random	73.5		11.5	10.2	0.1
A*	92.2	90.2		0	0
Minimax	92.4	88.9	1.5		0
MCTS	98	89.3	16	3	

Table 5.2: Agent comparison for dimension 3x3

	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		1	73.3
Minimax	92.4	88.9	1.5		33.3
MCTS	100	100	56.7	92	

Table 5.3: Agent comparison for dimension 5x5

	Random	Semi-Random	A*	Minimax	MCTS
Random		2	0	0	
Semi-Random	99.6		0.2	0.7	
A*	100	99.7		0	
Minimax	100	99.3	100		
MCTS					

Table 5.4: Agent comparison for dimension 7x7

	Random	Semi-Random	A*	Minimax	MCTS
Random		0	0	0	
Semi-Random	100		0	0.1	
A*	100	100		2.8	
Minimax	100	100	100		
MCTS					

Table 5.5: Agent comparison for dimension 9x9

Chapter 6

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.

In chapter **??**, we formalize the notations for the game including the cells of the game board, movements and wall placements and using them, we formally describe the rules of the game. Additionally, in the chapter, we also propose an improvement upon the notation for wall placement to remove the ambiguity on the notations used in the current literature.

In chapter **??**, we describe the game tree of the game Quoridor. We provide 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 that.

In chapter **??**, we explained our implementation of the interfaces which are the fundamental part of our work. Further, we explain 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 **??**, 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. We show that

List of Figures

List of Tables

Appendix A

Attachments

A.1 First Attachment