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 AI algorithms. This thesis explores the implementation and evaluation of three distinct AI algorithms, namely A*, Minimax with Alpha-Beta Pruning, and Monte Carlo Tree Search (MCTS), within the realm of Quoridor gameplay.

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

1. Introduction

Over the past decade, Artificial Intelligence (AI) is becoming an integral part of many elements of modern world. In fact, in the recent years, it has become an the main point of evolution and revolution in many of the technologies and industries. From assisted or autonomous driving [Ma et al., 2020], chat bots Wu et al. [2023] to gaming Skinner and Walmsley [2019], it has been a major revelation in evolving the existing technologies to generating new industry space with the plethora of new use cases.

AI has brought a revolutionary transformation to the gaming world, pushing the boundaries of what's achievable in both single and multiplayer gaming experiences. In recent years, AI has taken center stage with its remarkable accomplishments in mastering age-old games such as Chess, Go, and many others. 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.

Strategy games are a genre of gaming that require planning, often involving various tactics, decision making and execution while under various constraints (e.g., resources). Some of the examples of popular strategy games are Chess, Go, Shogi, Starcraft, etc. Strategy games are unique compared to many other genres of games in the sense that it requires a selection of an optimal move among a subset of sub-optimal moves based on a plan. In many scenarios, the size of the subset of sub-optimal moves depends on exploration of the game space, simulation and prediction of sequence of the player's and the opponent's moves all while managing certain resources efficiently.

AI, due to its suitability towards solving problems involving complex decision making, taking into account sequences of the player's and the opponents possible moves (e.g., game space), outcome prediction or position evaluation all while considering the available resources, have been deemed particularly effective in playing the strategy games. The history of the use of AI in strategy games goes back to the last few decades. One of the initial marked impact of AI in the strategy gaming came in 1997 when IBM's Deep Blue Campbell et al. [2002] defeated World Chess Champion Garry Kasparov. The influence of AI in stratey games was more prominent with the rise and involvement of AI into the real-time strategy (RTS) games such as Warcraft and StarCraft Robertson and Watson [2014] and implementation of Monte-Carlo tree search (MCTS) in the strategy games such as Go Huang et al. [2011]. Recently, DeepMind's Alpha Go for Go, Alpha Zero for Chess Silver et al. [2017] and AlphaStar for StarCraft Team [2019] have widened the gap between the AI and human intelligence even further.

Designed by Mirko Marchesi, Quoridor stands out as an engaging strategic board game that is played between two or four players. The game is playes 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 have 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.

Compared to some other strategy games such as Chess and Go, Quoridor has not been extensively studied in the literature. In Lisa Glendenning [2002], the author developed an agent for playing Quoridor using genetic algorithm to optimize the weights. The authors in Mertens [2006] study the complexity of the algorithm also develop a Quoridor playing agent based on Minimax algorithm. The authors conclude that Quoridor has a similar state-space and game tree complexity as that of the games such as Chess. Likewise, the authors in Brenner [2015] developed an MCTS approach for Quoridor. Recently, in Iwanaga and Sakamoto [2022], the authors performed an analysis of the game for a miniature 5 by 5 board.

The primary objective of this thesis is to construct a well-structured framework and user-friendly interfaces that seamlessly integrate AI algorithms into the Quoridor game. The development of AI algorithms customized to Quoridor's unique 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.

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.

1.1 Acknowledged Works

Quoridor, being a widely popular game, has attracted a fair number of attention from the research community, resulting in successful AI agent developments. Some of the notable works include:

- **Mastering Quoridor [Lisa Glendenning, 2002]** The writer 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 paper, 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.

2. Background

In this chapter, we give a brief overview of different AI techniques that have been considered for implementing the agent for Quoridor including the minimax algorithm.

2.1 Zero-sum game

A zero-sum game is defined as a game where an advantage to one side means an equal loss to the opponent, resulting in the sum of zero. In zero sum games such as chess, tic-tac-toe, an advantage or a win for a team would mean a disadvantage or a loss for another team.

An example of a zero sum game is poker where, for example, 5 people buy in with a total of 50 euros each, making the table total of 250 euros. In the end, despite some winners or losers in the game, the total will be distributed among the players. In this game, there will be a zero net gain resulting in a zero sum game.

Similarly, in chess, the evaluation of a given position on the board may be slightly advantageous to white. This will always imply that the position is slightly disadvantageous for the player playing the black pieces. A loss of a rook for black would mean, in some cases, an equivalent loss worth 5 pawns. This would imply an equivalent 5 pawn gain for the opponent playing with the white pieces.

2.2 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 two-player zero-sum strategy games such as chess and tic-tac-toe

Mathematically, a minimax algorithm can be defined as the following:

$$\bar{v}_i = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}) \tag{2.1}$$

where,

$$i = \text{index of the player of interest} \tag{2.2}$$
$$-i = \text{index of the opponent(s)} \tag{2.3}$$
$$a_i = \text{action of the player of interest)} \tag{2.4}$$
$$a_{-i} = \text{action of the opponents)} \tag{2.5}$$
$$v_i = \text{value function of player i} \tag{2.6}$$
$$\bar{v}_i = \text{minimax value of the player of interest} \tag{2.7}$$

The value represented by $v_i(a_i, a_{-i})$ defines the initial set of values or outcomes of the game depending on various actions of the player i and player $-i$. We

first marginalize away the set of actions of the the player (e.g., a_i) from the set of possible outcomes by maximizing the value v_i over the set of every possible action of the player. The implication here is that we are evaluating all the possible actions of the player and selecting the one that maximizes the value function. We then choose an action a_{-i} from the set of all possible actions of the opponent that minimizes the maximum value function.

In a two-player zero sum game, this concept translates to the following interpretation. Given a two person zero sum game with finite actions of the players, there exists a value V such that given an opponent's strategy, the value to the player of interest is V. This implies that given the player of interest's chooses the strategy, the opponent's value is $-V$ making the sum zero.

Below in an example of a minimax algorithm in zero sum game:

- Consider two players A and B.

- The players simultaneously choose a number (an integer) between 1 and 5.

- The payoff is the difference between the two chosen numbers. For example, if Player A chooses 5 and player B chooses 1, Player B has to give $5 - 1 = 4$ to Player A.

- This can be exemplified with the payoff matrix below:

	B chooses 1	B chooses 2	B chooses 3	B chooses 4	B chooses 5
A chooses 1	0	-1	-2	-3	-4
A chooses 2	1	0	-1	-2	-3
A chooses 3	2	1	0	-1	-2
A chooses 4	3	2	1	0	-1
A chooses 5	4	3	2	1	0

Table 2.1: Payoff matrix for the described game from the perspective of player A

For the $\max_{a_{-i}} v_i(a_i, a_{-i})$ part, the player A first marginalizes away the actions of the opponent a_{-i}. This results in the following table:

	B chooses 1	B chooses 2	B chooses 3	B chooses 4	B chooses 5
$\max_{a_i} v_i(a_i, a_{-i})$	4	3	2	1	0

Table 2.2: Marginalied value of the payoff matrix after maximizing over the player's actions

We first marginalize away the set of actions of the player (e.g., a_i) from the set of possible outcomes by maximizing the value v_i over the set of every possible action of the player.

We then determine the $\min_{a_{-i}}$ over the set of chosen values and determine that 0 is the minimum when B chooses 5.

We determine that for player A, no matter what B chooses, the optimal strategy to maximize the value to the player i is choosing 5. this is the dominant strategy. This is also known as the dominant strategy when the value of the
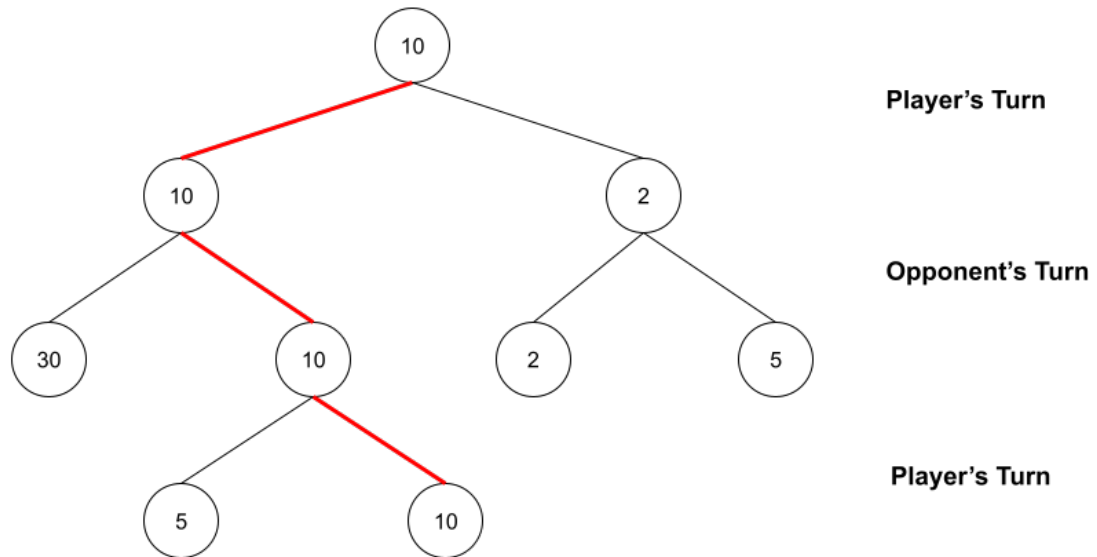
Figure 2.1: Figure illustrating an exemplary game tree and decision based on minimax algorithm

minimax can be determined only based on player A's actions without depending on the actions of player B.

In the above example, the search size is limited. However, in many games such as chess, the size of the search space is too big. In such cases, it would be impossible to perform an exhaustive search on the game space due to limited computational resources. For example, in chess, for every possible action of the player, there are many responses of the opponent. This search space increases even more if we consider the further moves and hence the possibilities of the player and the opponent. In terms of graph, every action of the player can be considered as a node and the action of the opponent can be considered as a child of the node. The further moves of the player and the opponent can be captured by representing the moves as further child nodes.

In Figure 2.1, we illustrate an example minimax evaluation and decision selection procedure. The tree in 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.

The minimax algorithm first creates a game tree based on all the possible moves of the player and opponent. Then for the leaf node positions, the game position is evaluated and assigned a value. For example, the leaf nodes associated with the branch labelled in red in this graph is assigned values of 5 and 10. In this graph, we do not show further child nodes due for better illustrations, but it can be assumed that the tree has further children nodes. In the game, the player wants to maximize the evaluation on their turn whereas the opponent wants to minimize the evaluation on their turn. The player chooses a move with an evaluation of 10 (maximizing its evaluation) and labels the action as 10. The player then determines that the opponent chooses an action that minimizes the evaluation, choosing an action with evaluation of 10 rather than 30. The player

hence labels the parent node as 10 indicating that with best play, the player can achieve an evaluation of 10 from that node. The player then chooses the node with evaluation of 10 when choosing between 10 and 2, maximizing its evaluation and subsequently labels the parent node as 10. The red line shows the path the player determines as optimal leading to a decision choosing the the action labelled with evaluation of 10.

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 Figure 2.1, the full game tree may consist of further children nodes and branches. However, due to limited computational resources avaliable, the player only considers a depth of 3 to determine the game evaluation. With further depth, the player may be able to make better decisions at the cost of the computational complexity required to evaluate all the different subsequent possibilities.

Other ways to limit the complexity of the minimax algorithm are alpha-beta pruning and parallel minimax algorithm.
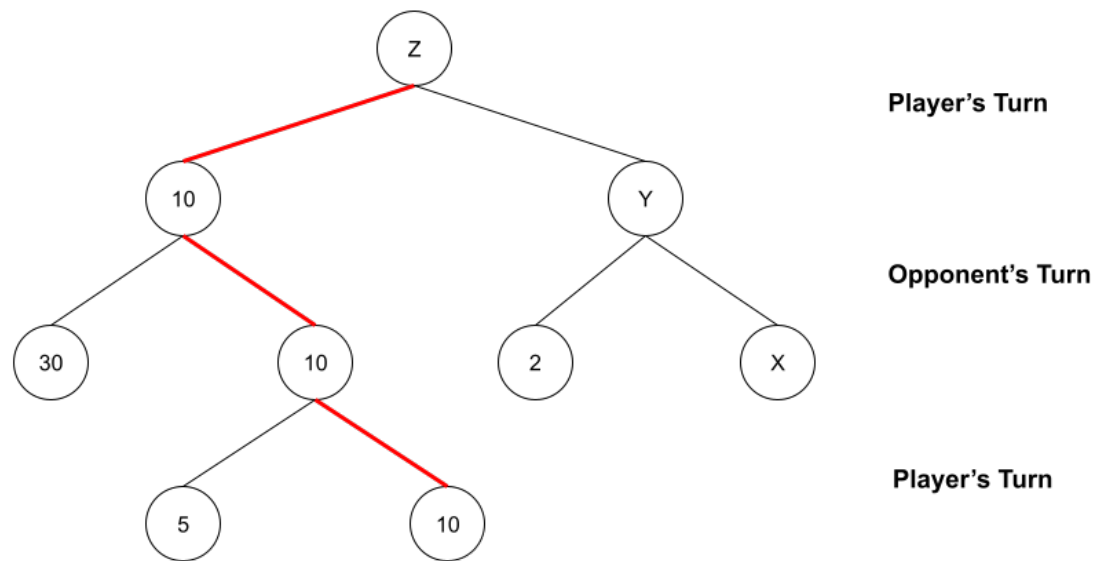
2.2.1   Alpha-beta pruning

Figure 2.2: Exemplary figure illustrating the alpha beta pruing for minimax algorithm

Alpha-beta pruning is a another way to reduce the number of nodes that are evaluated by the minimax algorithm. 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 moves on to another action instead of further evaluating the worse action.

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 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 2.2, if the player determines that

$$Z = max(10, Y) = max(10, min(2, X)) \tag{2.8}$$

, 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.

2.2.2   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.

For example, in Figure 2.1, the player can evaluate the branches with evaluation 10 and 2 in parallel reducing the time complexity of the algorithm.

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.

2.3   Monte-Carlo Tree Search

MCTS Coulom [2006] is a heuristic tree search algorithm popular in decision-making processes, mostly popular in strategic games such as chess and Go to solve the game tree. This algorithm is mostly popular in scenarios 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. For example, for a given position in tic-tac-toe, the MCTS algorithm can be executed randomly a large number of times. Based on the statistics of the game wins and losses based on selection of certain moves, the weights are allocated to the children nodes in the graph.
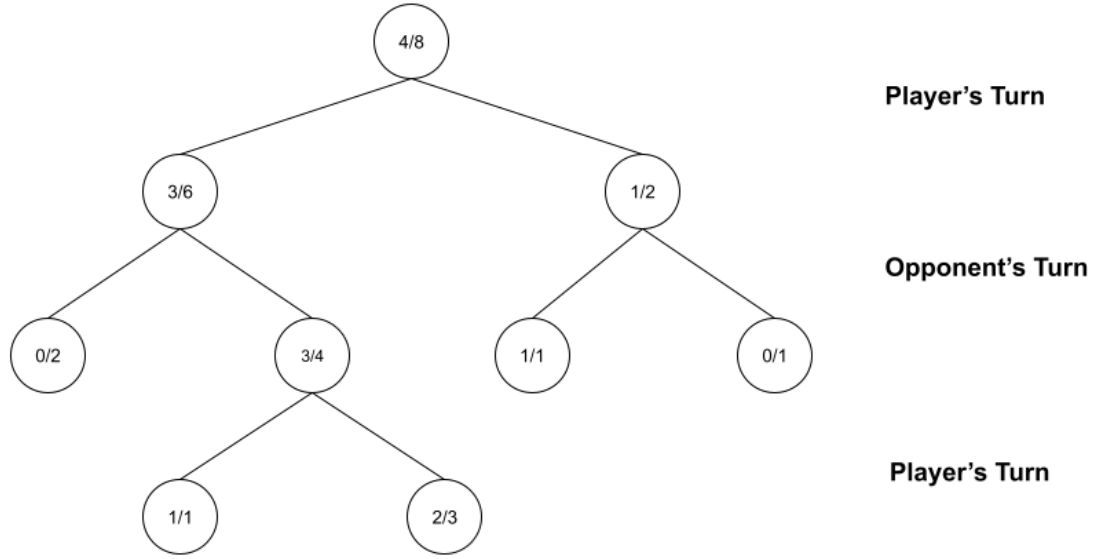
Figure 2.3: Figure illustrating an iteration of the MCTS algorithm

An example of the MCTS algorithm is presented in Figure 2.3. The game is evaluated from the given position (e.g., the root node in the graph) for players and opponenet turns. The policy for the game can be random (e.g., determined by a random variable with certain distribution such as the uniform random distribution). 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 based on the game run through the node. 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. In figure 2.3, in the first player's turn, an action gives 3/6 chances of winning based on previous iterations whereas another gives 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 2.4, an unevaluated option (e.g., node inside the dotted box) is explored and simulated.

3. Simulation: This step involves in 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 2.4, the red arrows show

the back propagation step as a result of exploration and simulation step where the probabilities or the weights of the nodes are updated as a result of exploration and simulation steps.
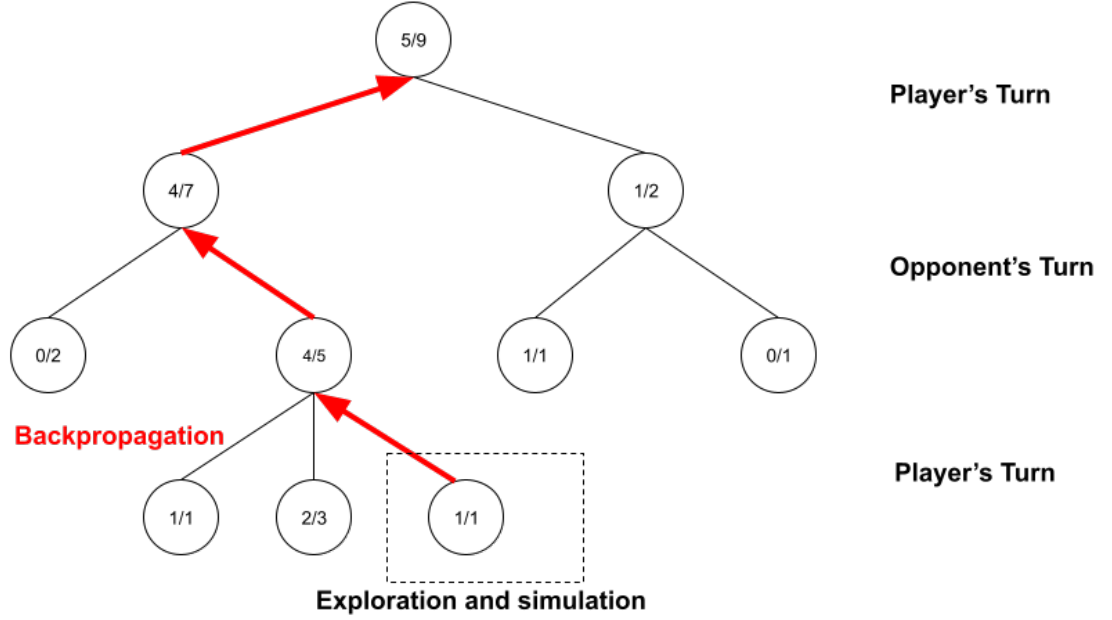

Figure 2.4: Figure illustrating steps 2, 3, and 4 of the MCTS algorithm

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

2.4 A-star algorithm

A star is a popular algorithm [Hart et al., 1968] used mainly for graph search and traversal problems. The main aim of the A-star algorithm is to find a path between a starting node and an destination node with the least cost.

3. 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.
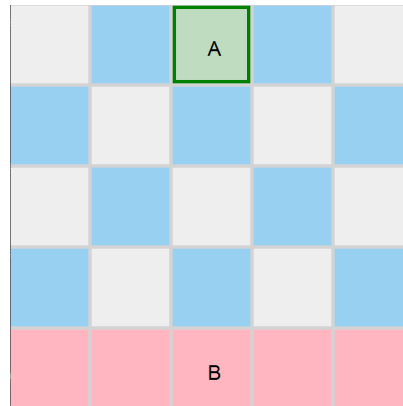

Figure 3.1: A 5x5 game board

As seen in figure 3.1, in this thesis, we consider a two-player Quoridor game with player A and player B. The pawn of player A is represented by the letter 'A' and that of player B is represented by the letter 'B'. In the figure, we display a 5x5

board with the squares on the edge of the board highlighted in pink belonging to player B whereas the squares on the opposite edge of the board belonging to player A. The board in the figure also shows the starting position of the two 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 movennt and wall placement rules.

3.1 Notation

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

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

First, defining the notation of the squares of the board, consider a Quoridor board of dimension 9x9 as an example. Let $\mathbb{K} = \{a, ..., i\}$ and $\mathbb{R} = \{1, ..., 9\}$ and C denote the 2D Quoridor board with $C_{i,j}$ 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 3.1, with these notations, the pawn A is in the position $C_{1,c}$ and the pawn B is in the position $C_{5,c}$.

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

$$M(C_{ij}) = ji, \text{ where,} j \in \mathbb{K} \text{ and, } i \in \mathbb{R} \tag{3.1}$$

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(3.1), the movement of the pawn to the cell $C_{j,i}$ can simply be defined by the notation ji. The movement of the pawn A, in figure 3.1 from its position from cell $C_{1,c}$ to the position of B in $C_{5,c}$ through the cells $C_{2,c}$, $C_{3,c}$ and $C_{4,c}$ can be defined with moves $c2$, $c3$, $c4$ and $c4$ requiring a total of 4 moves.

Additionally, as described earlier, the Quoridor game is played between multiple players where each player in a turn can make a move. Hence, the turns are represented numerically, for e.g., 1., 2., 3. represents the first, second and third move respectively. Hence considering the scenario where player A moves its pawn in figure 3.1 from $C_{1,c}$ to $C_{2,c}$ and then to $C_{3,c}$ and player B moves its pawn from $C_{5,c}$ to $C_{4,c}$ to $C_{4,b}$, the notations can be represented as: $1.C2C4$, $2.C3B4$. Here the numbers 1. and 2. represents the two moves of each players and $1.C2c4$ represents the movement of the players A's pawn to C2 and then player B's pawn to C4. The order of the player's movement can be dependent on the player that moves first. For example, in the example, player A moves first to $C4$, hence it is notated before player B's move. This order has to remain constant throughout the game.

Similarly, after notation of the board and the movement, the notation for wall placement is another important component for Quoridor. The notation for wall placement is defined by the following equations:
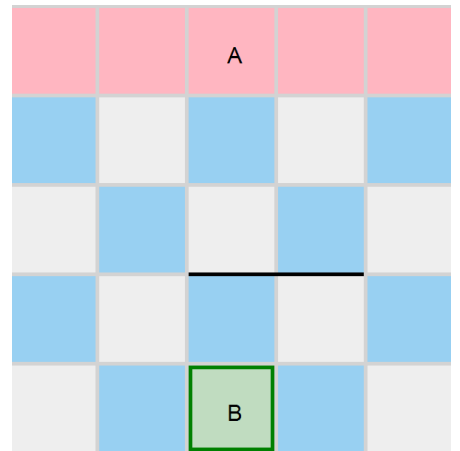
$$W(C_{i,j}) = jiD \text{ , where, } j \in \mathbb{K}, i \in \mathbb{R} \text{ , and } D \in \{h, v\} \qquad (3.2)$$

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 (3.2) defines the arrangement of the wall. The wall from a reference point can be placed vertically or horizontally. This is represented in the equation by the characters 'h' and 'v' respectively.

In the notations defined as per Glendenning's notations and Quoridor Strat's notations, there is a difference when it comes to the wall represnetation even though the mathematical formulation is the same as presented in equation (3.2).
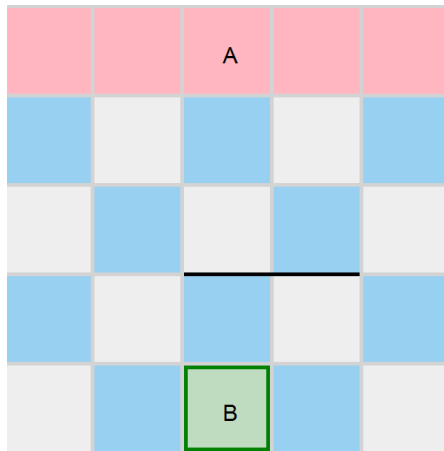
(a) Glendenning's Notation: **c3h**   (b) Quoridor Strats Notation: **c4h**

Figure 3.2: Notation differences

As seen in Figure 3.2, the difference lies in the exact position of the reference point of the walls of the wall. In the reference square, there are 4 corners. The reference point for horizontal wall placement is always the left corners of the cell. However, the difference in the notations lie in fact that whether the reference left corner is the left upper corner or the left lower corner. In the Quoridor Strats 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 3.2. In Glendenning's Notation, the wall placement with reference to the square $C_{3,c}$ i.e., c3h is defined with respect to the lower left corer of the square $C_{3.c}$. In contrary, in the Quoridor Strats Notation, the wall placement with reference to the $C_{4,c}$ i.e., c4h, is defined with respect to the upper left corner of the cell $C_{4,c}$. Since the lower left corner of the cell $C_{3,c}$ and the upper left corner of the cell $C_{4,c}$ are the same, the wall placement due to the notation difference were the same as seen in the figure.

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

$$W(C_{ij}) = jiD \text{ , where } i \in R, j \in K \text{ ,and, } D \in \{N, S, E, W\} \qquad (3.3)$$

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 3.2a, the walls can now be represented by **c4N**, i.e. a Northern wall from the cell $C_{4,c}$.

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

3.2 Rules

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

3.2.1 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 3.3a, the walls are placed in such a way that player A has a valid path towards the edge of the player B, in particular, to the cells $C_{5,a}$, $C_{5,b}$, $C_{5,c}$, $C_{5,d}$ and $C_{5,e}$. However, the wall placement in figure 3.3, due to the placement of the wall **c3N**, the player A does not have a valid path towards the edge cells of player B. Hence, the move **c3N** would be classified as an invalid move.

- A placed wall cannot intersect any of the previously placed walls. For example, in figure 3.3a, a walls has been placed with the moves **a1W**. A subsequent move **a2N** would be an invalid move in the game due as it requires the walls to intersect.

- Walls cannot be placed along the edges of the board. Walls must be placed to create a barrier for exactly 4 cells. For example, in figure 3.3a, a wall cannot be placed with the move **a1E** as it would only touch 2 cells $C_{1,a}$ and $C_{2,a}$ as it is on the edge of the board.

- Every player possesses a limited supply of walls, and once they exhaust these walls, they are unable to place any additional ones. Consequently, in such situation the player is only allowed to maneuver their pawn on the board.

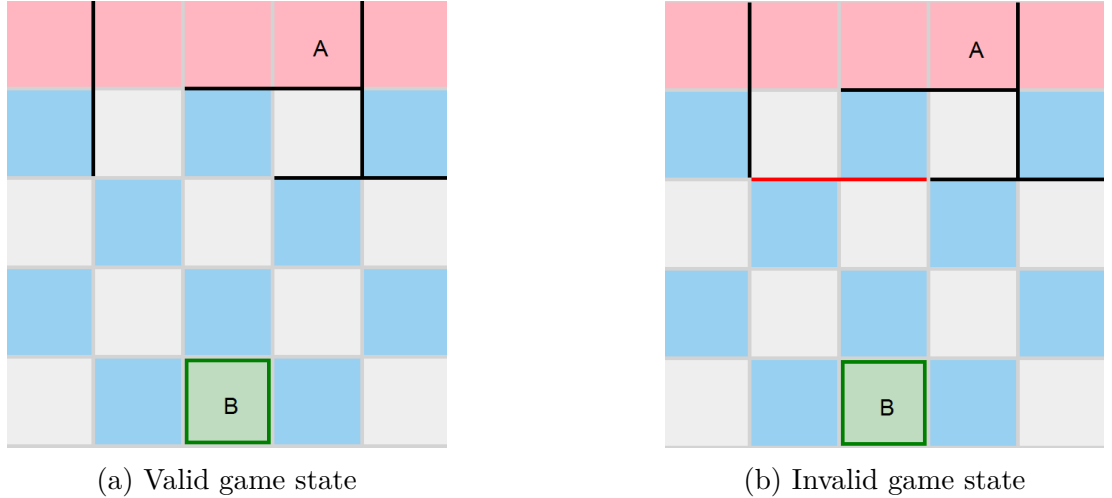(a) Valid game state   (b) Invalid game state

Figure 3.3: Example of an invalid wall placement

The game state represented by Figure 3.3a shows the situation after 5 turns, with it currently being player B's turn to move. Since both **A** and **B** have viable paths to their respective goal rows and all walls have been placed according to the rules (see *Section 3.2.1*), the game state shown in Figure 3.3a is considered valid.

However, player B disrupts the rules by placing the red wall, violating the specified wall-placement rules (see *Section 3.2.1*), consequently rendering the game state represented by Figure 3.3b invalid.

3.2.2 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 3.4 the player B can move to either of the squared highlighted in green. The player can move to east, west or south direction or can jump in the north direction over the player A from square $C_{4,c}$ to the square $C_{2,c}$ as there are no walls between the squares $C_{4,c}$, $C_{3,c}$ or $C_{2,c}$.

  - If there is a wall between the opponent and the jumping square, the player can jump to a cell on either side of the opponent's cell, given the cell is accessible from the opponent's cell (i.e., there are no walls). In the section picture from left in the figure, for example, the player B cannot jump over player A to the cell $C_{2,c}$ due to the wall placement **c3N**. The player in this case can jump over to the cells on either the east side (i.e., cell $C_{3,b}$) or the west side (i.e., cell $C_{3,d}$) of the player A given there are no walls in between $C_{3,c}$ and $C_{3,b}$, and $C_{3,c}$ and $C_{3,d}$ respectively.

  - Players cannot jump over walls. If a player is jumping from a cell C_1 to cell C_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.
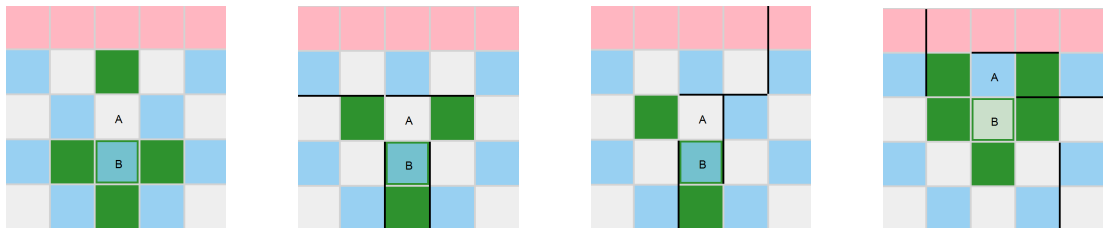

Figure 3.4: Examples of possible moves for player B in the game state

4. Game Analysis

In Chapter 3, we briefly 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 will classify Quoridor within the realm of strategic games, examine its game tree, state-space and tree complexity, and explore the implications of these factors on gameplay and artificial intelligence application.

4.1 Classification of Quoridor

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

4.1.1 Discrete

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

4.1.2 Deterministic

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

4.1.3 Zero-sum

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

4.1.4 Perfect Information

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

4.2 Game-Tree

A game tree for an **abstract strategy game** 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.

The following depicts a (partial) game tree for the Quoridor game with a board of size 3x3:
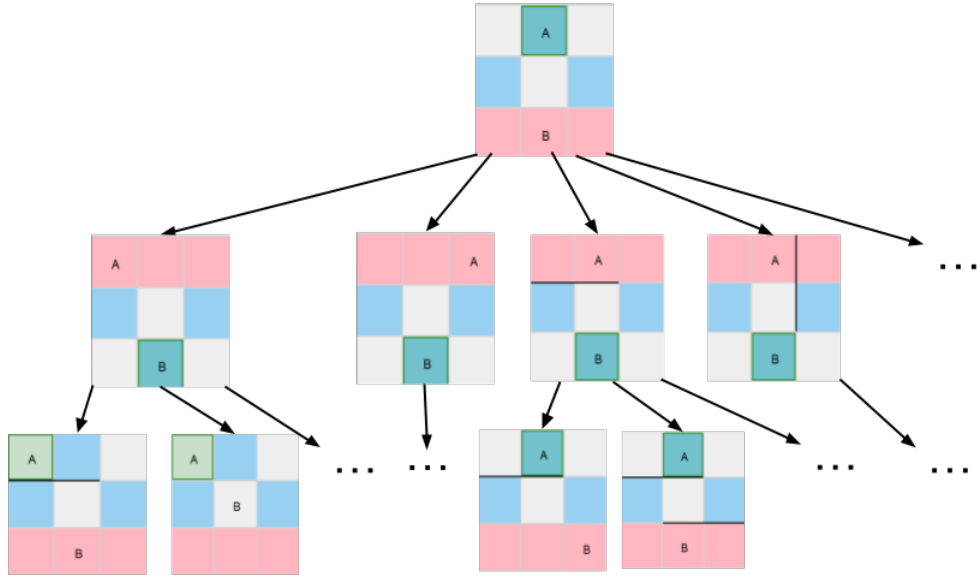

Figure 4.1: A partial game tree for a 3x3 game board.

As depicted in the figure 4.1, 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

4.2.1 Branching Factor

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

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

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

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

<table>
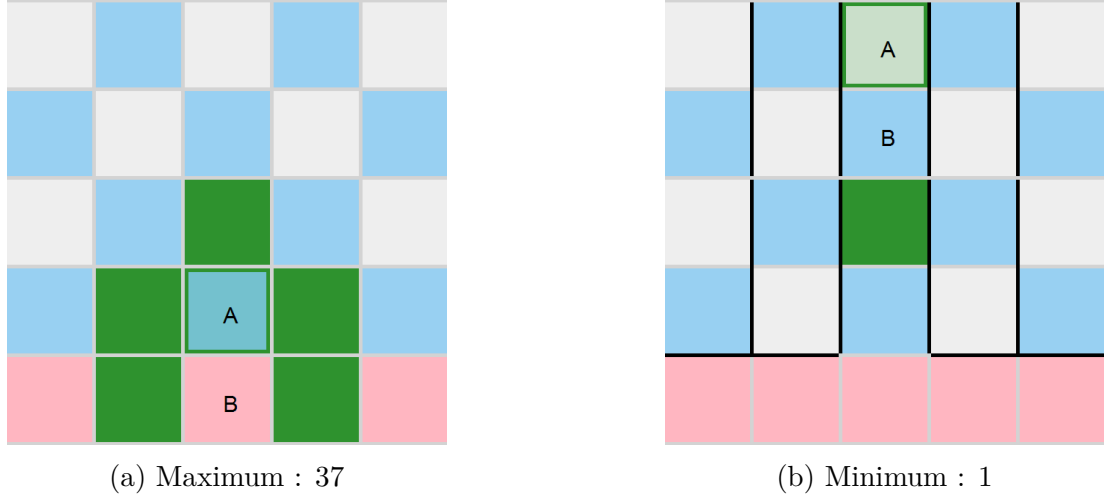<tr><td>(a) Maximum : 37</td><td>(b) Minimum : 1</td></tr>
</table>

Figure 4.2: Branching factor differences

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

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

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

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

Average Branching Factor

In sub-section 4.2.1, we saw that the branching factor is not uniform due 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 affect in the average branching factor.

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

To find an estimate of the average branching factor B_avg, we use the Algorithm 1

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 $i \leftarrow 1$ **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 AB** and **Semi-random** agents, each for boards of dimensions 3, 5, 7 and 9, and the results of the average branching factor can be seen in figure 4.3.
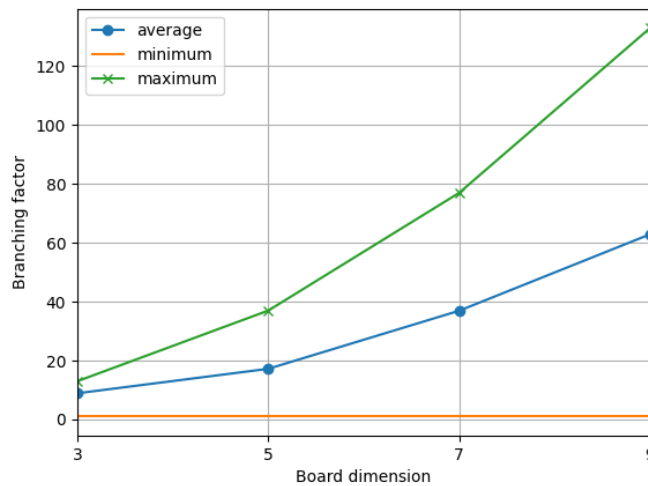

Figure 4.3: Branching factor for boards of different sizes

From figure 4.3, like with the maximum branching factor, we can see a similar exponential trend of the average branching factors in order of N. Furthermore, based on only the four dimensions and their averages and maximum branch-

ing 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 **62**, which is very close to the value proposed by [Lisa Glendenning, 2002]

4.3 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 3.2.1 and pawn movement rules 3.2.2. Because of this, we loosen the regulations to come up with an upper-bound.

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

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

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

As for walls, from equation 4.1, we know that there are $2(N - 1)^2$ ways of placing walls on the board, both horizontally and vertically. Since we know that each wall occupies 2 cells, we will 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, $N_W = 20$. We can extrapolate the number of walls that may be available for boards of other dimensions too based on the number of squares. For example, for 3x3 board, $N_W = 2$, for 5x5 board $N_w = 5$, for 7x7 board $N_W = 11$.

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:

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

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

However, in the game 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:

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

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 $S = S_w * S_p$.

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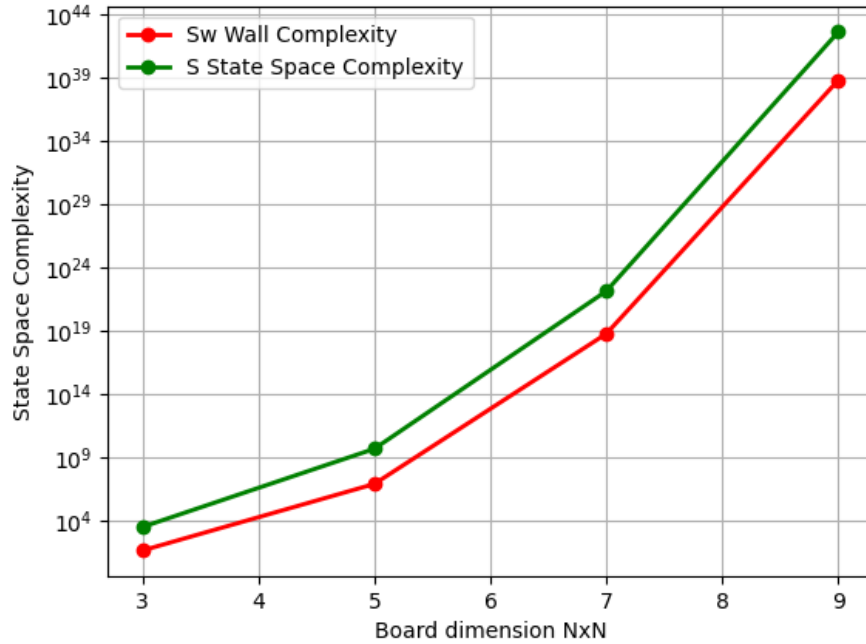

Figure 4.4: Complexity vs Board Dimension

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

4.4 Game tree complexity

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

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

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

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

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

4.5 Comparison with other games

	$\log(S)$	$\log(G)$	B_{avg}
Tic-tac-toe	3	5	4
Connect-four	13	21	4
Chess	44	123	35
Quoridor 3x3	3		8
Quoridor 5x5	10		18
Quoridor 7x7	22		38
Quoridor 9x9	42	162	62
Go	170	360	250

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

In the above 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 [Mertens, 2006].

As we can see from the above table, the

5. 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, game-agnostic systems 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.

5.1 Interfaces

The architecture of our implementation leverages interfaces, fundamental constructs in object-oriented design that define contracts for implementing classes. These interfaces specify a set of methods related to game mechanics, which are vital for the operation of AI agents. 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.

The fundamental interfaces and their contents include the following:

```csharp
interface IPlayer<TPlayer>
{
    /*
     *Retrieves the active player in the game.
     *Parametrized over TPlayer instance
     */
    TPlayer CurrentPlayer { get; }
}


interface IDeepCopy<T>
{
    /*
     *Create a deep copy of an instance, (e.g. of a game
     *state, allowing for safe simulations and backtracking)
     *without altering the actual object.
     */
    T DeepCopy();
}


interface IStaticEvaluation
{
    /*
     *Computes a 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
     */
    public double Evaluate(bool currentMaximizer);
}
```

```csharp
interface ITerminal
{
    /*
    * A property that checks whether the game has
    * reached a terminal state.
    */
    bool HasFinished { get; }
}

interface IValidMoves<TMove>
{
    /*
    *Returns all the valid moves that can
    *be made from the current state.
    */
    IEnumerable<TMove> GetValidMoves();
}

interface IOpponent<TPlayer>
{
    /*
    * Retrieves the active opponent in the game state,
    * parametrized over TPlayer
    */
    TPlayer Opponent { get; }
}

interface INeighbors<TMove>
{
    /*Yields the neighboring positions or states from a given
    *position pos, crucial for deter-mining potential player actions
    */
    IEnumerable<TMove> Neighbors(TMove pos);
}

interface IMove<TMove>
{
    // Applies a move to the game state
    void Move(TMove move);

    /*
    *Reverts a move, restoring the
    *game state to its previous condition
    */
    void UndoMove(TMove move);
}

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

They form the backbone of the implemented AI system, ensuring that the

agents are versatile and can be adapted to new games with minimal changes to the underlying codebase.

5.2 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.

5.2.1 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 pseudocode for the random agent along with the interface is given below:

```
public class RandomStrategy<TMove, TGame, TPlayer>(int seed)
where TGame : IValidMoves<TMove>
{
    public TMove BestMove(TGame game, TPlayer player)
    {
        // IValidMoves<TMove>
        var validMoves = game.GetValidMoves();

        //pick a random number corresponsing to the index of the move
        var randIndex = _random.Next(0, validMoves.Count());

        //return the random move
        return validMoves.ElementAt(randIndex)
    }
}
```

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

5.2.2 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. 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.

For this purpose, we define the Semi-Random agent defined by the following code.

```
public class SemiRandomStrategy<TGame, TMove, TPlayer>
    where TPlayer : IAStarPlayer<TMove>
    where TGame : INeighbors<TMove>, IRandomizableMoves<TMove>
{
    public TMove BestMove(TGame game, TPlayer player)
    {
```

```
        // IRandomizableMoves <TMove >
        //these moves , when used 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 a random move
        var randIndex = _random.Next(0, possibleMoves.Count);
        return possibleMoves[randIndex];
    }
|
```

Elaborate a bit more on the semirandom strategy, what is strategy.bestmove, what is randomizable move, what is non randomizable move in the context of the game

5.2.3   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.

Define the evaluation function

The minimax class then has the following signature:

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

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

    // ITerminal
    if (depth <= 0 || game.HasFinished) {
        // IStaticEvaluation
        return game.Evaluate(maximizingPlayer);
    }
                    ⋮

    // IValidMoves <TMove >
    foreach(var move in game.GetValidMoves())
    {
        // IMove <TMove >
        game.Move(move);

        //recursively call the MinimaxStep functino and update the
        //best score and move
```

```
        result = MinimaxStep(game, depth - 1, !maximizingPlayer);

        //update best score and move
        bestMove, bestScore = Update(result);

        // IMove<TMove>
        game.UndoMove(move);
    }
    return bestMove;
}
```

In the above implementation, the minimax algorithm is performed for a defined depth indicating how many moves for the player and the opponent is to be evaluated. The evaluation is performed either each time the indicated depth is evaluated or if the game has finished. Define the evaluation function here.

The evaluation with the minimax alogithm is performed for all the valid moves that indicates all the children node of from the current node in the game tree. For each valid mode, the minimax algorithm (MinimaxStep) method is recursively called to evaluate the branches of the children node. The minimax method is defined below:

```
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);

        //recursively call the MinimaxStep functino and update the
        //best score and 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)
```

29

```
        {
            bestMove.BestMove = move;
            bestMove.Value = result.Value;
        }
        if (bestMove.Value < alpha)
            break;

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

In the above code, the minimax algorithm with alpha-beta pruning method is defined.

5.2.4   A-Star Agent

```
public class AStar<TMove, TMaze, TPlayer>
    where TPlayer : IAStarPlayer<TMove>
    where TMaze : INeighbors<TMove>
{
    public TMove BestMove(TMaze maze, TPlayer player)
    {
        var start = new Node<TMove> { CurrMove =
            player.GetCurrentMove() };
        var openSet = new HashSet<Node<TMove>>() { start   };
        var closedSet = new HashSet<Node<TMove>>();

        //set the initial node as the current node
        var currNode = start;

        while (openSet.Count() > 0)
        {
            //get the node from the opoen set with the lowest f-score
                value
            var nodeWithLowestFscore = openSet.MinBy(s => s.FValue);
            currNode = nodeWithLowestFscore;

            //put this node in closed set and remove it from open set
            closedSet.Add(nodeWithLowestFscore);
            openSet.Remove(nodeWithLowestFscore);

            //if the closed set contains a goal node, we're done
            if (closedSet.Any(node => player.IsGoal(node.CurrMove)))
                return currNode;

            //INeighbors<TMove>
            foreach (var neighbor in maze.Neighbors(currNode.CurrMove))
            {
                var neighborNode = new Node<TMove> { CurrMove =
                    neighbor };
                //if it's in the closed list, skip it
                if (closedSet.Contains(neighborNode))
                    continue;
```

```
                var gScore = currNode.GValue + 1;

                //if the neighbor is not in the open set or if the
                    neighbor's G score is
                //lower than the calculated g-score, update the g score
                    and set the neighbor
                //as current node's parent.
                if (!openSet.Contains(neighborNode) ||
                    neighborNode.GValue < gScore)
                {
                    neighborNode.GValue = gScore;
                    neighborNode.HValue =
                        player.CalculateHeuristic(neighbor);
                    neighborNode.FValue = gScore + neighborNode.HValue;
                    neighborNode.Parent = currNode;
                    //if neighbor was already present, it won't re-add.
                    openSet.Add(neighborNode);
                }
            }
        }
        return null;
    }
```

5.3   Example: Tic-Tac-Toe

In this section, we demonstrate how seamless it is to integrate the AI agents to
games. We will use the tic-tac-toe game for this purpose.

The interfaces above 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, IValidMoves<int>, IMove<int>, IPlayer<int>,
        IOpponent<int>, IDeepCopy<TicTacToe>, IWinner<int>,
        IStaticEvaluation

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

    public bool HasFinished => CheckWin();

    public int Winner => GetWinner();

    public int CurrentPlayer => turn;

    public int Opponent => turn % 2 + 1;

    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;
    }

    public double Evaluate(bool currentMaximizer) {
```

```
        var winner = GetWinner();
        if (winner == CurrentPlayer) return 1.0;
        if (winner == Opponent) return -1.0;
        //other complex unfinished game states
        return 0.0;
    }

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

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

    public TicTacToe DeepCopy() {
        var t = new TicTacToe();
        t.Cells = Cells; //copy by value
        t.turn = turn;
        return t;
    }

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

    int GetWinner() {
        //check if someone won the game, and if so, return
    }

    bool CheckWin() {
        //check all 3 consecutive adjacent squares (including
        //diagonals), and return true if they're filled by the
        //same player.
    }
}
```

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

```
var tt = new TicTacToe();

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

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

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

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

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

6. Experiments

Conclusion

Bibliography

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

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

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

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

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

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

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

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

Yifang Ma, Zhenyu Wang, Hong Yang, and Lin Yang. Artificial intelligence applications in the development of autonomous vehicles: a survey. *IEEE/CAA Journal of Automatica Sinica*, 7(2):315–329, 2020. doi: 10.1109/JAS.2020.1003021.

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

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

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

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

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

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

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

Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. A brief overview of chatgpt: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10(5):1122–1136, 2023. doi: 10.1109/JAS.2023.123618.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1   First Attachment