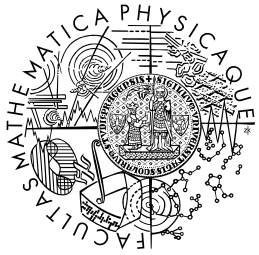


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

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

Introduction

Artificial Intelligence (AI), over the past decade, 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 (**?**), chat bots **?** to gaming **?**, AI 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 **?** 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 **?** and implementation of Monte-Carlo tree search (MCTS) in the 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.

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 **?**, 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. 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 **?** developed an MCTS approach for Quoridor. Recently, in **?**, 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.

0.1 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 (?)** 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.

1. 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.

1.1 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.

1.2 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:

$$\bar{v}_i = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}) \tag{1.1}$$

where,

$$i = \text{index of the player of interest} \tag{1.2}$$
$$-i = \text{index of the opponent(s)} \tag{1.3}$$
$$a_i = \text{action of the player of interest)} \tag{1.4}$$
$$a_{-i} = \text{action of the opponents)} \tag{1.5}$$
$$v_i = \text{value function of player i} \tag{1.6}$$
$$\bar{v}_i = \text{minimax value of the player of interest} \tag{1.7}$$

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 1.1: 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 1.2: 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

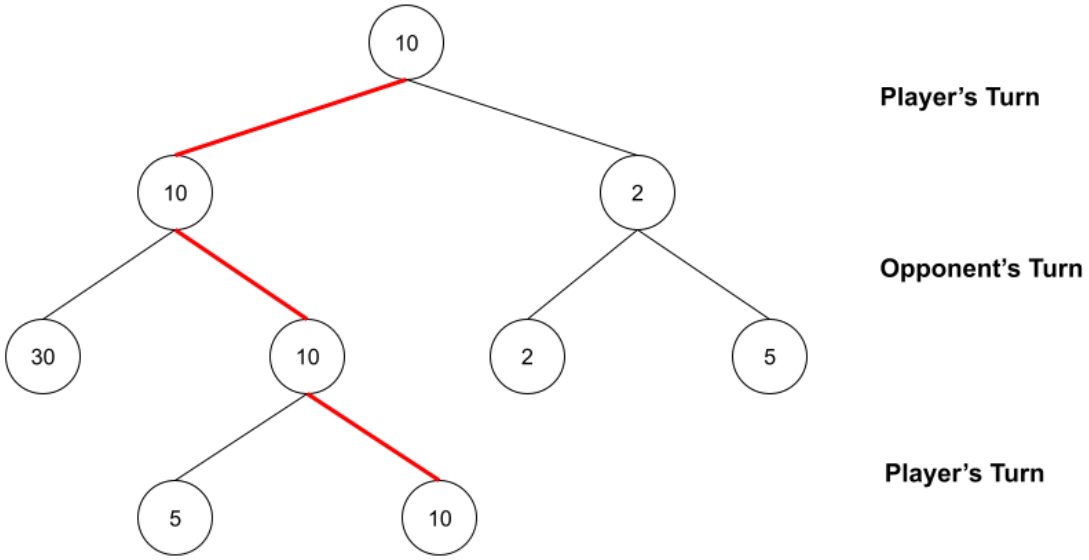


Figure 1.1: Figure illustrating an exemplary game tree and decision based on minimax algorithm

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

1.2.1 Alpha-beta pruning

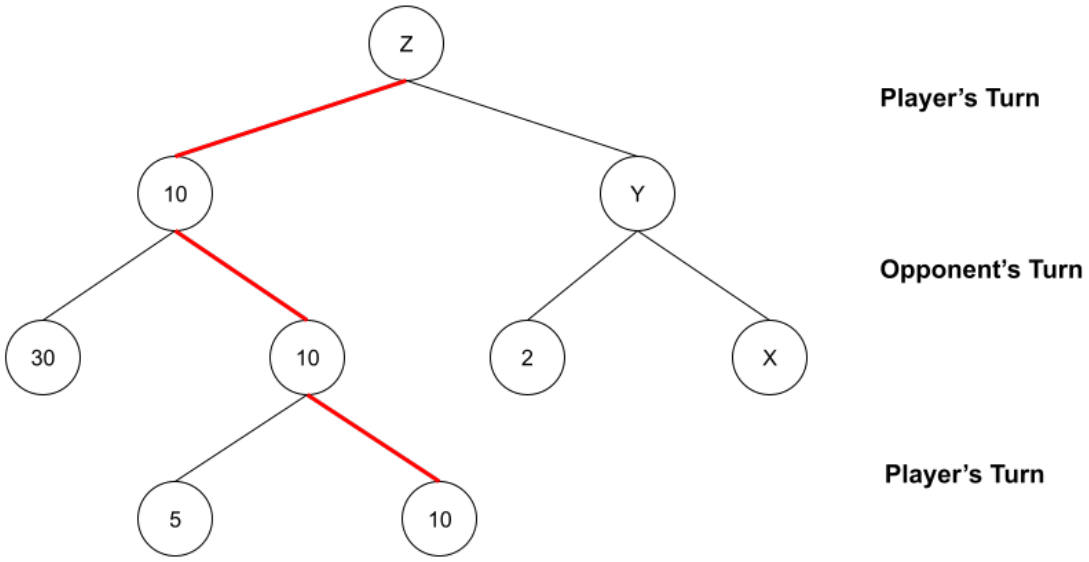


Figure 1.2: Exemplary figure illustrating the alpha beta pruing for minimax algorithm

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

$$Z = max(10, Y) = max(10, min(2, X)) \tag{1.8}$$

, 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.

1.2.2 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 **??**, 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.

1.3 MCTS

MCTS **?** 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.

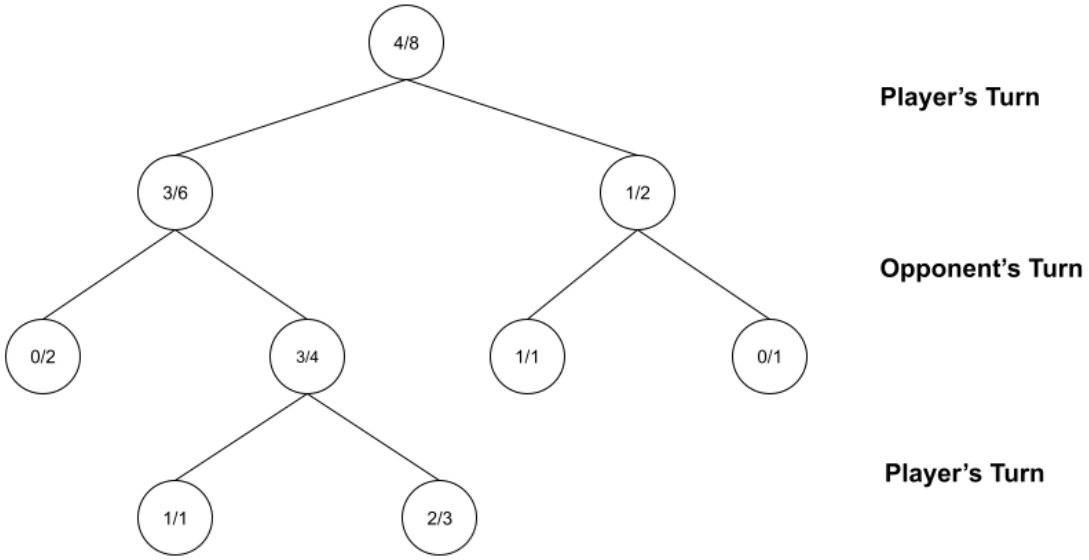


Figure 1.3: Figure illustrating an iteration of the MCTS algorithm

An example of the MCTS algorithm is presented in Figure **??**. The game is evaluated from the given position (e.g., the root node in the graph) for players and opponenent 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 **??**, 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 **??**, 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 **??**, 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.

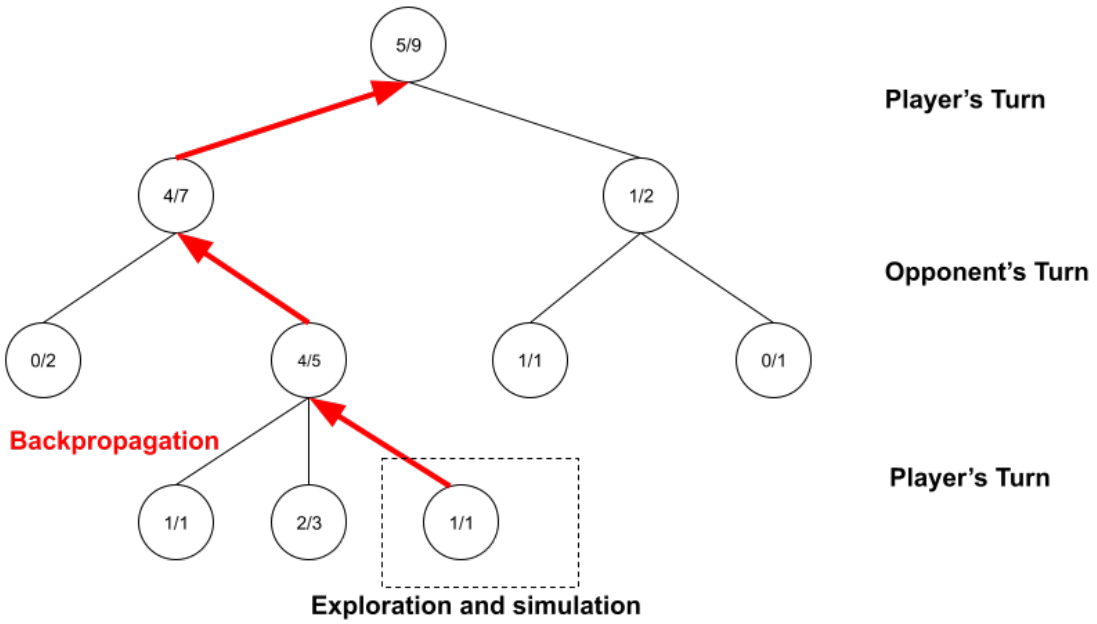


Figure 1.4: Figure illustrating steps 2, 3, and 4 of the MCTS algorithm

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. Description

The game begins with an NxN chess-like board where each square represents a potential position for the game pieces. It contains grooves that runs between the squares where players can place their walls. Each player is represented by a pawn represented by a character label that start on opposite sides, with their pawns located at the center of their respective edge. The primary objective is to be the first player to move their pawn to the row of squares on the opposite side 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.

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 cells and occupies exactly four cells 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.

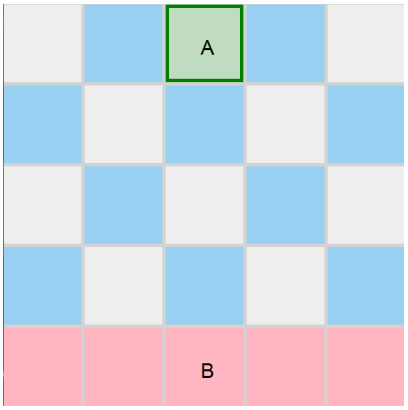


Figure 2.1: A 5x5 game board

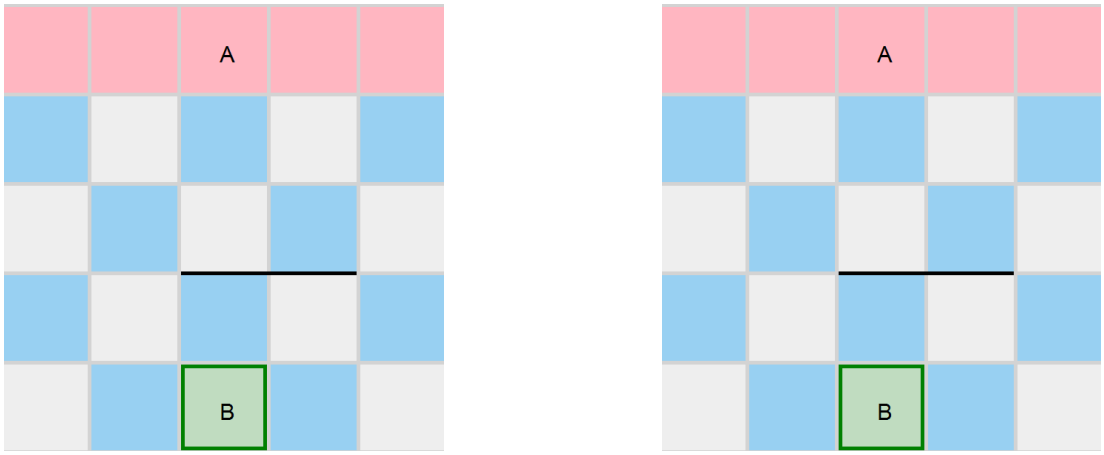
As seen in figure **??**, In the beginning, player A starts at cell (2, 0) and player B starts at cell (2, 4). A's goal is to reach the cells highlighted in pink. The gray areas between the cells are grooves for wall placement.

2.1 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 Strats Notation** ((**?**)).

Let $R = \{a, ..., i\}$ and $C = \{1, ..., 9\}$
Both notations follow the same principle of labelling each cell by C_{ij} where $i \in R$ and $j \in C$.
Move M and Wall W are denoted algebraically, where
$M(C_{ij}) = ij$ where $i \in R$ and $j \in C$,
$W(C_{ij}) = ijD$ where $i \in R$, $j \in C$ and $D \in \{h, v\}$

The difference between the two notations is the way the walls are represented.



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

Figure 2.2: Notation differences

As seen in Figure **??**, the difference lies in the point of reference of the wall. In **Quoridor Strats Notation**, each wall coordinate is represented by the lower-left cell the wall follows along, whereas in **Glendenning's Notation**, each wall is represented by the upper-left cell the wall follows along.

Even though they 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 is why I have decided to use a different representation for walls. Instead of vertical and horizontal wall with respect to a cell, we define a direction explicitly.

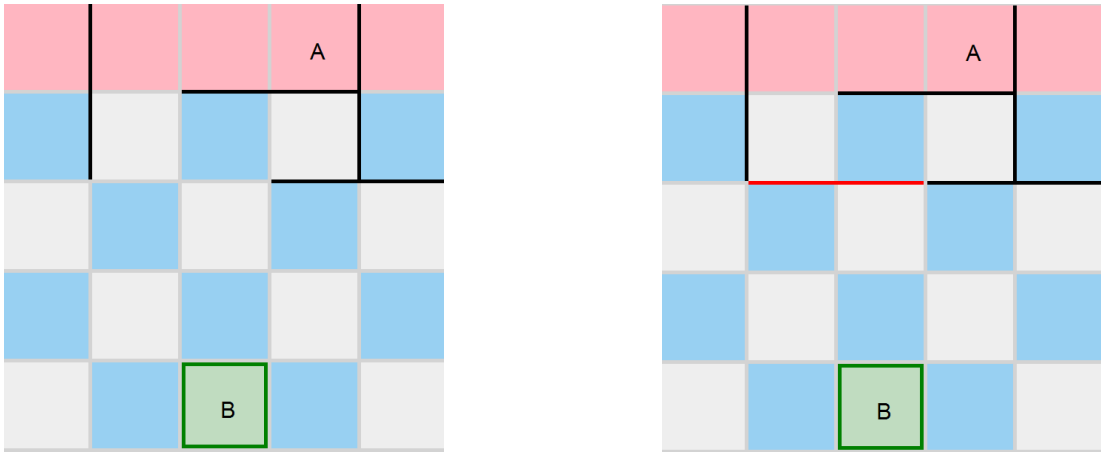
$W(C_{ij}) = ijD$ where $i \in R$, $j \in C$ and $D \in \{N, S, E, W\}$

Looking back at Figure **??**, the walls can now be represented by **c4N**, i.e. a Northern wall from the cell C_{c4}.

2.2 Rules

2.2.1 Wall placement rules

- Walls 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 *(See figure* **??***)*
- A placed wall cannot intersect any of the previously placed walls.
- Walls cannot be placed along the edges of the board. Walls must be placed to create a barrier for exactly 4 cells
- Every player possesses a limited supply of walls, and once they exhaust these walls, they are unable to place any additional ones. Consequently, 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 unit at a time in the North, South, East, or West directions during their turn. Diagonal movements are not allowed.
- **Jump**
  - If an opponent is at to the cell a player intends to move to, the player can jump over the opponent provided there is no wall between them or behind the opponent they intend to jump over. In the latter case, 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.
  - Players cannot jump over walls
  - A Player is allowed to jump over multiple opponents as long as they adhere to the aforementioned conditions.

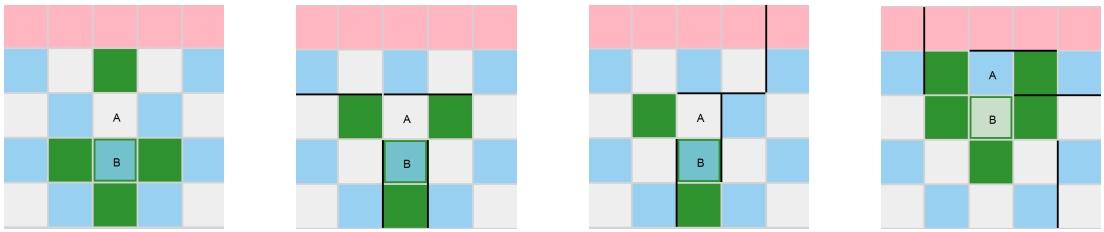


Figure 2.4: Examples of possible moves for player B in the game state

3. Game Analysis

In Chapter **??**, 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.

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, 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 **??**)

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.

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.

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.

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

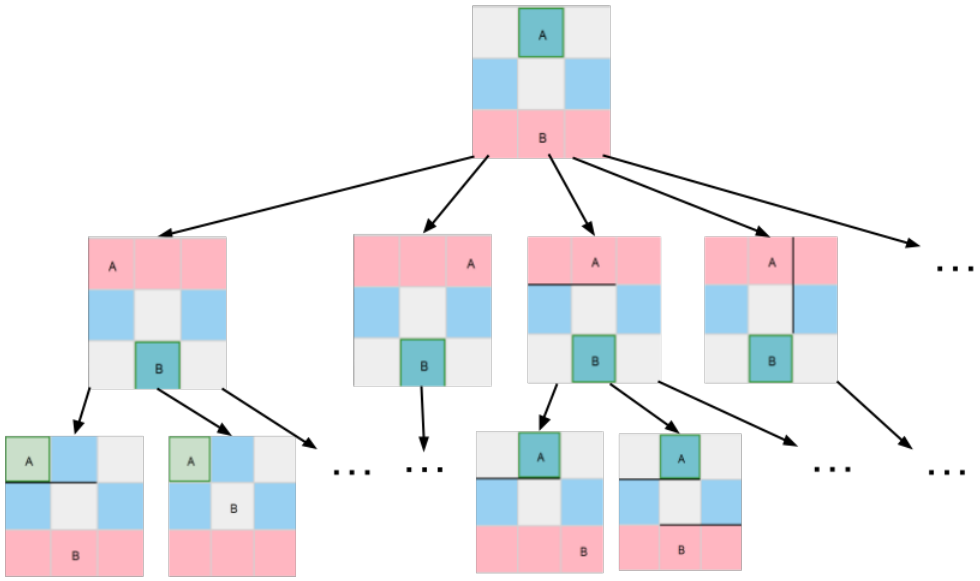


Figure 3.1: A partial game tree for a 3x3 game board.

3.2.1 Branching Factor

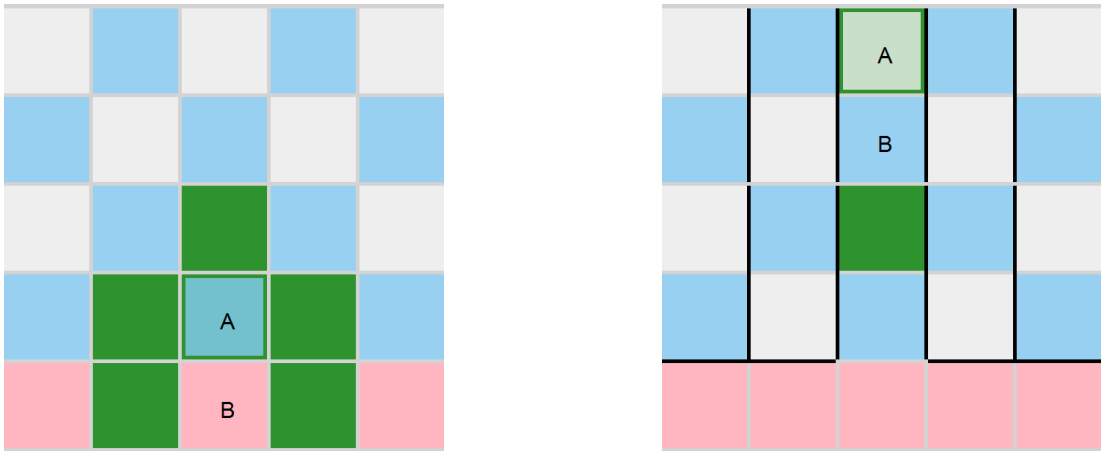
The branching factor of a **Game-tree** is the number of child nodes of each node, or in other words, the number of possible moves a player at their turn can make, given the game state.

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 has a **branching factor** of **11**.

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

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

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.



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

Figure 3.2: Branching factor differences

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 **??**, A has no available slot for wall-placement, and the already present walls block A from moving anywhere except for cell **c3**. 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 affect 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, we use 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 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 **??**

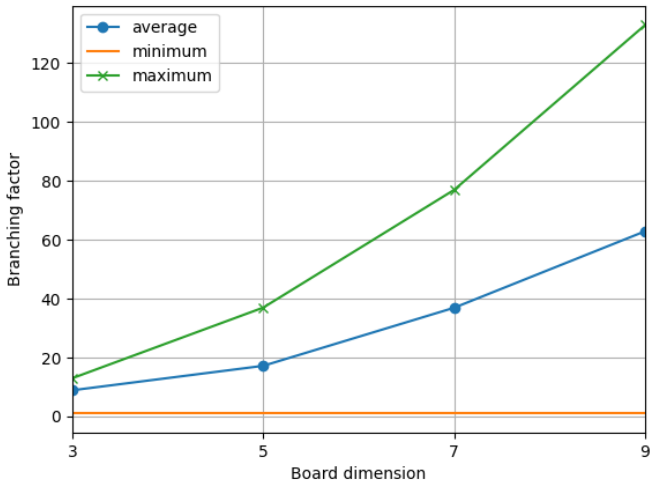


Figure 3.3: Branching factor for boards of different sizes

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 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 **?**

3.3 State space complexity

The state space complexity in Quoridor refers to the total number of possible game states that can be reached from the initial game state. 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.

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. So, the total number of wall placement, S_w is given by: **?**

$$S_w = \sum_{i=0}^{20} \prod_{j=0}^{i} (N^2 - 4i) \tag{3.3}$$

And thus, the total state space complexity is given by $S = S_w * S_p$, which, for the standard quoridor board size 9x9 is $3.9905 * 10^{42}$ **?**

3.4 Game tree complexity

The game tree complexity refers to the number of possible games that can be played, i.e the total number of states in the game tree. For example, the root node for game tree for the tic-tac-toe game has a branching factor of 9, or, in other words, the first player has 0 possible places to put the 'X' into. The branching factor decreases by 1 as the game progresses, and hence, the game complexity of tic-tac-toe is $9 * 8 * ... * 1 = 9!$.

In complex games like Quoridor, the game-tree complexity can be estimated by raising the branching factor to a power of the total number of moves by the players, which is given by **?**

$$G = 1.7884 * 10^{162} \tag{3.4}$$

3.5 Comparison with other games

	log(state space complexity)	log(game tree complexity)	Branching factor
Tic-tac-toe	3	5	4
Connect-four	13	21	4
Chess	44	123	35
Quoridor	42	162	60
Go	170	360	250

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

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, 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.

4.0.1 Interfaces

The architecture of our AI algorithms 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:

- **ICurrentPlayer<TPlayer>**
  - **TPlayer CurrentPlayer { get; }**: Retrieves the active player.
- **IDeepCopy<T>**
  - **T DeepCopy()**: A method that creates a deep copy (eg. of a game state), allowing for safe simulations and backtracking without altering the actual object.
- **IMove<TMove>**: Encapsulates the operations of making and undoing moves.
  - **void Move(TMove move)**: Applies a move to the game state.
  - **void UndoMove(TMove move)**: Reverts a move, restoring the game state to its previous condition.
- **INeighbors<TMove>**: Defines adjacency relations on the game board.
  - **IEnumerable<TMove> Neighbors(TMove pos)**: Yields the neighboring positions or states from a given position `pos`, crucial for determining potential player actions.
- **IOpponent<TPlayer>**:
  - **TPlayer Opponent { get; }**: Provides access to the opposing player.
- **IStaticEvaluation**: Evaluates the static value of a game state.

  - **double Evaluate(bool currentMaximizer)**: 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.

- **ITerminal**:
  - **bool HasFinished { get; }**: A property that checks whether the game has reached a terminal state.

- **IValidMoves<TMove>**: Provides a set of legal moves available from the current game state.
  - **IEnumerable<TMove> GetValidMoves()**: Returns all valid moves that can be made from the current state.

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.

Conclusion

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment