Using Monte Carlo Tree Search, Neural Networks, and Random Forest Decision Trees in an AI Agent to Play Quoridor

Iram Liu, Jacob Groner, Mason Raffo

May 17, 2025

Abstract

We present a Monte Carlo Tree Search, neural network, and random forest decision tree evaluation method to play Quoridor, a two person deterministic strategy game with a high branching factor and numerous intricacies.

GitHub Repo: `https://github.com/ili23/cs4701_sp25_quoridorai`
Presentation Recording: `https://drive.google.com/file/d/18f8EhxrBQaTMSOwoW8z7Qq6vCy2unBRW/view?usp=sharing`

1. Introduction

Quoridor is a deterministic, two player game with perfect information. Each player has a pawn on a square board and a limited number of fences. Players start on opposing sides of the board, and their goal is simple: to reach their opponent's back row. On each turn, they may either opt to move their pawn a single space or to place a fence. Fences sit between board spaces and block pawns from moving across. Players are required to leave at least one viable path for their opponent to reach the goal, and the first player to reach their corresponding goal area wins the game.

2. Project Description

We sought to make an AI agent that was capable of playing Quoridor. Initially, we intended to focus on a primarily convolutional neural network-based approach and evaluate it against an agent that played using random roll-outs. As we progressed, we found that using random roll-outs presented significant challenges, eventually abandoning their use altogether. We implemented our preliminary attempts in Python, but the language's high overhead forced us to complete our final implementations in C++. This re-implementation proved to be unexpectedly difficult, costing us considerable time debugging the many edge cases surrounding pawn moves.

After our models were complete, we found that the convolutional neural network was under-performing. We investigated deeper into the mechanics of the game, and outsourced some complex pathfinding logic to features before the

models and used a random forest decision tree classifier. Our second model was less complex, but performed better due to the intelligent computations encoded in its input features.

2.1. Rules of Quoridor

As previously discussed, Quoridor is played on a square board and each player controls one pawn (Fig. 1A). Quoridor has several variations, many of them using a 9×9 sized board. We have opted to use a 5×5 board due to limited computing resources. The aim of the game is to move your pawn into the goal regions (Fig. 1C). To slow down their opponent, players have a certain number of fences (3) that they may place on the board to block their opponent's path. These fences each span two spaces, and a player cannot cross a fence (Fig. 1B). A fence may not be placed if doing so would block either player's only remaining path to the goal (Fig. 1D). Pawns may never occupy the same space. To prevent stalemates, pawns may jump over each other if their path is blocke by the other pawn (Fig. 1E). Pawns may jump diagonally only if a fence blocks their primary jump (Fig. 1F). To limit the length of games, we instituted (50). Games that exceed this number of moves are declared to be drawn.

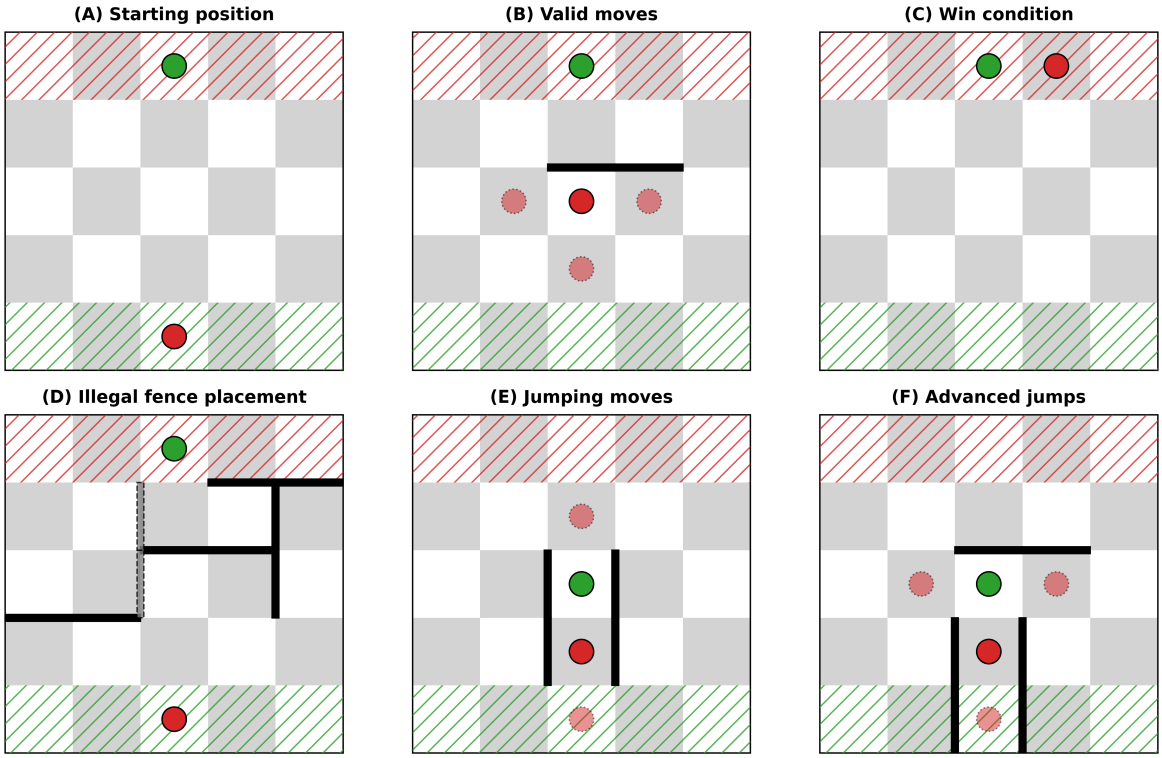

Figure 1. Rules illustrations. **(A)** The starting position. Player 1 is red; Player 2 is green; Fences are black lines. Hashed regions denote the goal area a player must reach to win. **(B)** Valid pawn moves for player 1 are shown. **(C)** Player 1 has won the game. **(D)** Fences cannot be placed if they completely block either player from reaching the goal area. The lightly shaded fence would not be allowed. **(E)** If a player is blocked by the opposing pawn, they may jump over. **(F)** Pawns may move diagonally only if a fence blocks a player's jump.

2.2. Monte Carlo Tree Search

In principle, Quoridor could be a solved game. However, each position has a staggering number of possible moves which makes a brute force tree search infeasible. The multitude of interacting "pawn jump" moves makes a case analysis proof impractically complicated. Instead, we present an AI agent to play the game using a Monte Carlo Tree Search (MCTS) and machine learning based game state evaluation methods.

MCTS is a method of evaluating and iteratively expanding a game tree, the tree of all possible moves from a given position. [3] In an environment where exploring every possibility is impossible, this process aims to balance two competing philosophies of position analysis: exploration and exploitation. Exploration involves devoting time to expanding less-searched areas of the game tree. While the initial analysis indicated that these potential moves had little value, it is possible that there are intricacies that make them extremely valuable moves with the right follow-up strategy. Exploration is about finding these hidden moves.

Exploitation, on the other hand, focuses on more deeply expanding a few promising ideas. The first layers of analysis indicate that these moves are encouraging, and a deeper analysis could determine which is the absolute best. With limited computing time and resources, it is impossible to fully explore both exploration and exploitation. MCTS provides a method to find an equilibrium between devoting all the resources to one or the other.

MCTS is performed iteratively. The primary data structure is a tree, with nodes that represent possible game states and edges that represent a possible move. The children of a node are all the possible moves that a player could make from the current position. The root of the tree represents the current game state. A leaf node either represents a terminal state (the game is over) or an unexplored possibility. An iteration of the MCTS expands one leaf node of the tree to consider one more move into the future along that branch of play. Each iteration has four phases: selection, expansion, evaluation, and backpropagation.

2.2.1 Selection

The selection phase is key in the MCTS's ability to balance exploration with exploitation. During the selection phase, the algorithm decides which leaf node in the game tree to expand. Starting from the root node, children nodes are recursively selected until a leaf node is reached. Children are scored with the following expression, and the child with the highest score is selected. [5]

$$\frac{w}{n} + c\sqrt{\frac{\ln N}{n}} \tag{1}$$

In this expression:

- w is the sum of evaluations for this node and all of its children.
- n is the number of times this node has been selected.

- N is the number of times this node's parent has been selected.
- c is the exploration parameter. Higher values will prioritize exploration while lower values will emphasize exploitation.

The theoretical ideal value for c is $\sqrt{2}$, and we found this value to work well. If there are any child nodes that are yet to be selected ($n = 0$), one of them will always be selected to avoid divide by zero errors.

2.2.2 Expansion

If the selected node is a terminal state (the game is won, lost, or drawn), this step is skipped. Otherwise, the selected leaf node is expanded. Every possible move from the game state represented in the selected node is added as a child to the leaf node.

2.2.3 Evaluation

Perform an evaluation for the selected node. This step will vary based on the method in use. Evaluation functions f are a scalar valued function that map a game state to an evaluation value e such that $-1 \leq e \leq 1$. Terminal states are always mapped to one of $\{1, -1, 0\}$ depending on the outcome. A good evaluation function maps game states that are more favorable to the current player as higher values and less favorable game states to lower values.

2.2.4 Backpropagation

After the evaluation is determined, it is recursively backpropagated to every parent node. Concretely, the value of w is updated for every node such that $w \leftarrow w + e$. The evaluation is then flipped with the operation $e \leftarrow -e$ and passed to the current node's parent. The evaluation must be flipped because the evaluation is always from the perspective of the player to move, so a strong position for a child is a weak position for its parent. This process is repeated until the root node is reached.

2.3. Heuristics

If a game tree can be fully expanded, one can use trivial algorithms to determine which player can force a win. (The Minimax decision rule is one example. [3]) When the game tree cannot be fully expanded, it is necessary to create a heuristic that can determine the value of a position. The accuracy of the heuristic function is directly correlated with the playing strength of the AI agent. Previous research into AI agents for Quoridor gave us insight into designing useful heuristics. [1] In this project, we compare five distinct heuristic methods:

1. **Random roll-outs.** Moves are played randomly from the position in the leaf node until one player wins. The evaluation corresponds to the winning player.
2. **Naive.** Has no knowledge of the game at all. Always gives a neutral evaluation ($e = 0$) unless a player has successfully won the game. Serves as a baseline for comparison.

3. **Basic.** Gives scaled difference between the player's Manhattan distances to the goal. Does not take any pathfinding or advanced logic into account.

4. **Convolutional Neural Network.** Uses a convolutional neural network to analyze the board position.

5. **Path Resiliency Random Forest.** Uses random forest decision tree classifier to categorize the position as won, lost, or drawn based on path resiliency features. A scalar is calculated based on the model's confidence in its classification.

2.4. Convolutional Neural Network Design

One key limitation we encountered throughout the project stemmed from the rule that requires both players to be able to reach their goal row after each fence placement. This necessitated running a breadth-first search (BFS) for each player for every possible fence placement. Since random rollout takes a lot of time to actually win, and there are many possible fence placements that the player can do, this greatly bottlenecked the number of MCTS iterations which could be performed.

To address this inefficiency, we aimed to replace the rollout function with a trained model capable of serving as an accurate evaluation function with less execution time. We chose a convolutional neural network (CNN) for this task due to its strength in capturing spatial patterns, such as pawn positions and wall configurations, independent of their locations on the board. CNNs have been successfully used to design AI agents for other board games, most notably Go. [4] This characteristic makes CNNs particularly well-suited for structured board games. The architecture of our CNN is shown in Fig. 2. The input consists of a 5×5 board represented across four channels: the positions of both pawns, and the locations of horizontal and vertical walls. The network includes three convolutional layers with ReLU activations, followed by fully connected layers. These layers map the extracted features to a single scalar output between -1 and 1, which estimates how favorable the current position is for player 1.

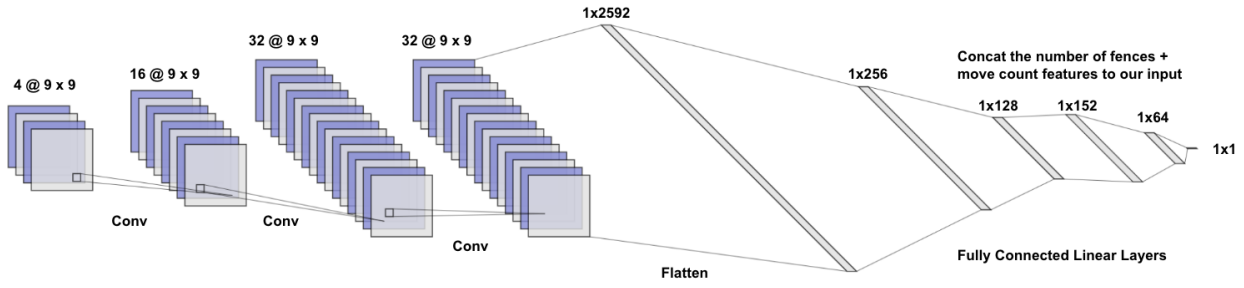

Figure 2. CNN architecture. Layer depth is not drawn to scale. Visualization created with open source tools. [2]

2.5. Training Data Generation

To use machine learning methods to evaluate board states, we needed to generate training examples. These training examples would include the path resiliency features, the board state, and a "ground truth" value of the position. To generate training examples, we used a simple agent to play hundreds of games against itself. The agent selected the first four moves randomly in order to create a diversity of board positions. After the fourth move the agent performed 1000 MCTS iterations using the basic heuristic for move selection (Manhattan distance). This was a good generation technique since we found a sufficient number of iterations allows any MCTS model to perform well (Section 3.2). After the game was complete, each board state played during the game was labeled with the player that won, or zero if the game ended in a tie. This set of labeled board states was aggregated and used to train both the decision tree and convolutional net.

We trained the CNN using 20,000 samples of only the board state. We used a standard training-validation split and trained until the loss stopped decreasing. We trained the random forest with 3,000 samples of only the resiliency features. With a uniquely generated test set, it achieved a accuracy of 0.81 when classifying each position as won, lost, or drawn.

2.6. Shortest Path Resiliency

The player with the shortest path to the goal will win without intervention from their opponent. The primary intervention is fences, which can be placed to block the previously existing shortest path between a player and the goal region. The essence of a good Quoridor strategy is determining which fence placements best block the opponent. Some placements will force a longer detour than others, but some will impede the player placing them more than their opponent. With this in mind, identifying which player has good fence placements available is an efficient way to evaluate a position.

A good fence placement will drastically increase the shortest path of the opponent. In general fence placements will only change the shortest path by the one or two moves as the player steps around the fence. In some cases, a fence placement will completely cut off a whole portion of the board from a player and force them to take a completely different path to the end. Shortest path resiliency is a technique which aims to find bottlenecks: fence placements that will optimally disrupt the opponent while providing minimal disruption to the current player's path. In identifying the bottlenecks, we are able to generate resiliency scores which provide insight into how fragile a player's shortest path. These scores are used as features for a random forest decision tree which evaluates board states.

The first step is to create a resiliency graph (R_1, R_2) for each player. This algorithm iteratively searches for shortest paths and then increases weights of edges along the shortest path by a constant value δ. After several iterations, edges that are difficult to avoid will accumulate large weight values, and edges that have equivalent alternate routes will have the additional weight distributed amongst them. This means that an edge with a high value represents a place were a wall could have a huge impact on the player's path. For all experiments, δ was set to 2. Due to the discrete and uniform

nature of these edge weight modifications, the number of iterations is not relevant beyond a sufficient threshold ($= 25$, in our testing).

Algorithm 1 Construct Resiliency Graphs

for each player $p \in \{1,2\}$ **do**
    Construct an undirected graph $R_p = (V,E)$:
        Create a node $v \in V$ for each board square
        Add edge $(u,v) \in E$ if u and v are adjacent and there is not a fence between u and v
        Set initial weights $w(e) \leftarrow 1$ for all $e \in E$
        Add a terminal node t to V
        For each goal square g of player p, add edge (g,t) with weight 0
    for $i = 1$ to 25 **do**
        Compute the shortest path π from player's current position s_p to t using Dijkstra's algorithm
        for each edge $e \in \pi$ **do**
            $w(e) \leftarrow w(e) + \delta$ if there exists a valid fence placement to block this move
        end for
    end for
end for
return R_1, R_2

The resiliency graph is used to generate a liability graphs (L_1, L_2) for each player. The liability graph is similar to the resiliency graph, but it recognizes that an edge crucial to blocking the opponents path may also block the player's own path. This is handled by computing weight differences.

Algorithm 2 Construct Liability Graphs

for each player $p \in \{1,2\}$ **do**
    $L_p \leftarrow R_p$
    Let R_o be the resiliency graph for the other player
    for each edge (u,v) in L_p **do**
        Let w_p be the weight of (u,v) in R_p
        Let w_o be the weight of (u,v) in R_o
        Set the weight of (u,v) in L_p to $\max(0, w_p - w_o)$
    end for
end for
return L_1, L_2

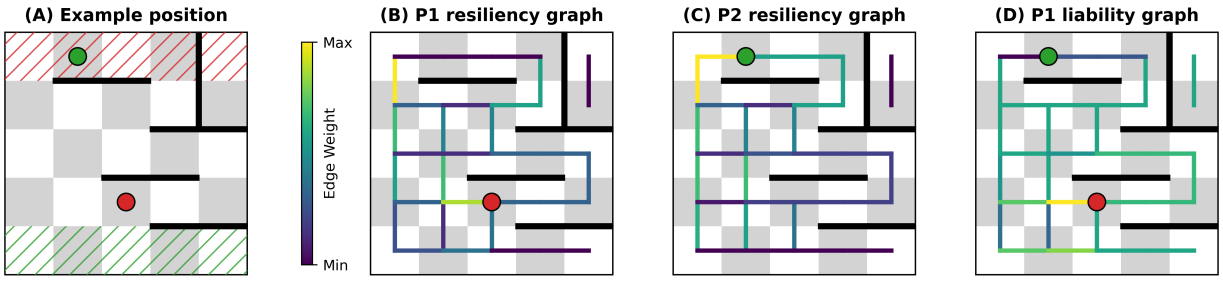

Figure 3. Path resiliency. **(A)** An example position of the board. Player 1 is red; Player 2 is green; Fences are black lines. Hashed regions denote the goal area a player must reach to win. **(B, C)** Resiliency graphs of players 1 and 2, respectively. Lines between board squares represent edges. The color of the edge represents the weight after the resiliency algorithm. Higher values indicate that the edge is harder to avoid when finding alternative short paths to the goal area. **(D)** Liability graph for player 1, as calculated by algorithm 2. In this example, a fence placed directly to the left of player 1 would require major detours for player 1 without interrupting many of player 2's paths to the goal. Thus, it has a high weight.

From these graphs, we compute several scalar-valued features for the decision tree. Each feature is calculated twice: once from each player's perspective. Thus, the input to the decision tree is a 10-dimensional feature vector.

1. **Path resiliency.** The distance of the shortest path in the resiliency graph from the player's current location to the terminal node. When this value is higher, the path is less resilient to wall placements (Fig. 3B,C).

2. **Path liability.** The distance of the shortest path in the liability graph from the player's current location to the goal. The higher the path liability, the less resilient player's shortest path is to wall placements in comparison to the opponent (Fig. 3D).

3. **Maximum liability.** The maximum weight value in the player's liability graph. This identifies if there exists one major bottleneck.

4. **Shortest path distance.** The distance of the player's shortest path to the goal region, not considering future wall placements. This does not rely on resiliency or liability graphs.

5. **Fence count.** The number of remaining fences the player has available to place.

The random forest classifies positions as either won, lost, or drawn, with certainties p_w, p_l, and p_d, respectively. The numerical evaluation used for backpropagation in the MCTS is

$$1 \cdot p_w + 0 \cdot p_d + (-1) \cdot p_l = p_w - p_l. \tag{2}$$

3. Evaluation

3.1. Principles of Evaluation

As mentioned earlier, our original goal was to have our bot compete against other people's algorithms as well as existing online Quoridor bots. Unfortunately, since we reduced the board size to 5×5 to address the computational limitations, we couldn't find any available Quoridor agents that were compatible with this configuration. As a result, we had to explore alternative evaluation strategies.

The strength of an MCTS system is entirely dependent on the strength of its evaluation function, the number of search iterations it performs, and the proper balance between exploration and exploitation. To confirm that our MCTS system was working as expected, we conducted several tests. These tests aim to validate the performance of our system by checking for two principles:

1. As the number of iterations is increased for a given system, the playing strength should increase.
2. As the accuracy of the evaluation function improves, the number of iterations that are required to achieve the same level of performance should decrease.

The first principle verifies that an MCTS system is working as expected, while the second principle gives us a method of comparison between different systems.

3.2. Principle 1: Increasing Iterations Improves Performance

To verify that increasing the number of iterations increased the performance of our models, we created a methodology to determine the relative playing strength between two models. A first thought would be to play the model against itself; however such a method would only test the model against positions that it itself selects rather than the full breadth of positions that it could encounter. Instead, we conduct many test games and begin each test game with a sequence of randomly selected moves. While this process does ensure the model is forced to play from a variety of unique positions, it presents a certain unfairness. Some positions are clearly lost, and are easily converted to wins by the opponent. We avoid this issue by looking at the ratio of wins to losses and draws. Stronger models will be able to find more subtle paths to victory, and will thus achieve a higher win to loss ratio against weaker models when starting from random positions.

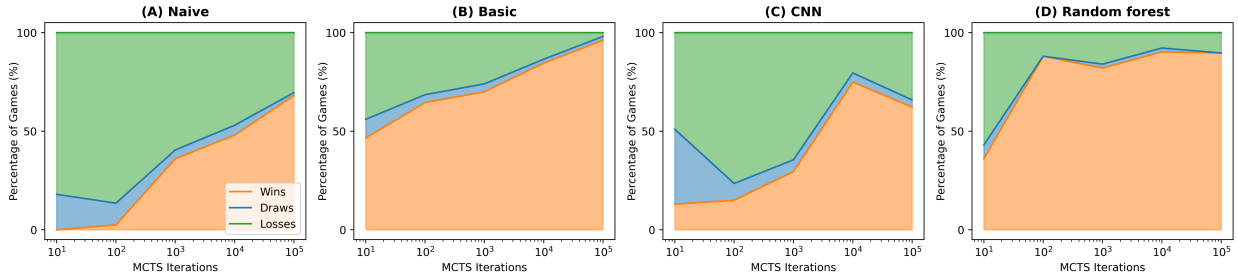

Figure 4. Win ratios. All models play as player 1 against the naive MCTS model as player 2. The number of MCTS iterations for the model in question (player 1) varies. The number of iterations for the naive opponent (player 2) is constant (= 1000). Every game is started with 8 random moves (4 for each side). Random moves are drawn uniformly from the set of all legal moves.

Playing various MCTS models against the naive model verifies that the MCTS models works as expected. The proportion of wins increases as the number of iterations increases, even when playing the naive model against itself (Fig. 5A). The differences in performance highlight the difference in accuracy between the models' evaluation functions. We were initially surprised by the high performance of the basic model (Fig. 5B). After more time, it seems that the basic model's laser focus on reaching the end goal without metaphorical distractions is not a big hindrance when it has a huge number of MCTS iterations to investigate hidden pitfalls. The CNN performed extremely similarly to the Naive model (Fig. 5C). We will examine this curious result later. The random forest decision tree performs very strongly, achieving an high win ratio with a minimal number of iterations (Fig. 5D).

3.3. Principle 2: Better Evaluation Functions Require Fewer Iterations

MCTS systems compensate for imperfect evaluation functions by iteratively exploring and exploiting the most promising potential moves. As the number of iterations increases, the model's understanding of the position becomes more complete, and its analysis becomes more accurate. We compare several different evaluation functions to understand how quickly our models are able to understand more complex positions.

3.3.1 Simple Tactics

We ask the models to evaluate a prearranged position (Fig. 5A). This position is designed like a tactics puzzle in chess: there is a clear winner with a bit of subtlety in the execution. First, we verify that our MCTS system is able to solve the puzzle. This puzzle is a forced win for player 1, but there is only one right move. Every other move allows player 2 to force a win. The MCTS system selects the best move by finding the child of the root node with the highest w/n ratio. We examine these ratios after 200 and 5000 iterations. We also examine the value of w/n for the root node after every MCTS iteration. This value is the strength of the position. A positive strength value means that the model has predicted the player to move will win the game. A negative strength value means the opponent is predicted win the game. It is clear that the naive model has found the best move after 200 iterations, and it is even more certain after 5000 iterations (Fig. 5B).

We use the naive MCTS system as a baseline to compare other systems against. Its evaluation function provides no information about the board, always returning zero in non-terminal positions. When MCTS systems evaluate with the naive evaluation function, there is an immediate and rapid drop in the calculated strength of the position (Fig. 5D). This drop is an expected result when analyzing a position with a hidden tactic. With only a shallow search, it seems that the position is lost. It is only after the model has found the winning move and sufficiently explored to verify it the position strength evaluation the increases. As expected, the calculated strength of the position continues to increase until the end of the experiment. (The true strength of the position is 1.)

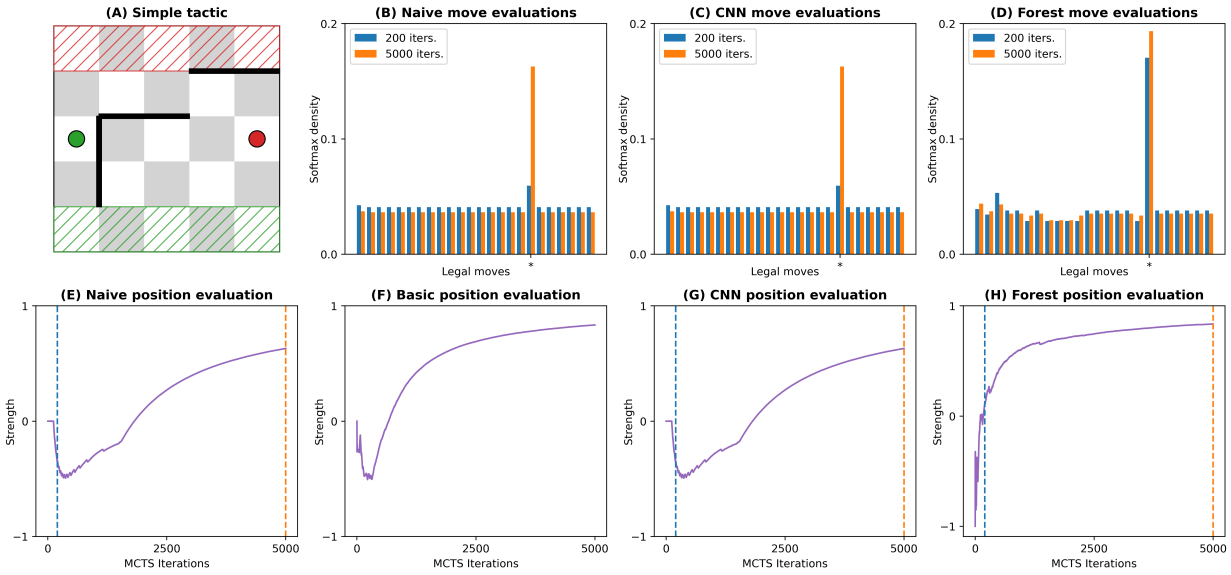

Figure 5. Solving a Simple Tactic. **(A)** A sample benchmark position that we selected to examine the model's strength as the iteration count increases. Player 1 (red) is to move. If they move directly toward the goal, they will lose and player 2 (green) will reach their goal first. However, if they block player 2 using a wall, they will win. For simplicity, each player has one wall remaining to place. This simple strategy puzzle should be solvable by all MCTS systems with enough iterations. **(B-D)** There are 24 valid moves from this position. Exactly one of them prevents player 2 from winning the game. This winning move is marked with *. The height of the bars is the softmax of w/n for each possible move after 200 and 5000 MCTS iterations of the basic MCTS model. Higher bars indicate moves that the model prefers. **(E-H)** Various models' evaluations of the position as they perform more MCTS iterations. The correct evaluation is 1, as the position is a forced win. Dashed lines indicate when the histograms were sampled.

The basic evaluation function provides an iterative improvement over the naive function. Comparing the strength calculated by the basic evaluation function to the strength calculated by the naive evaluation function reveals a similar yet improved pattern (Fig. 5E). There is a rapid drop followed by a more gradual improvement as the model gains an understanding of the position. However, this model approaches the correct evaluation slightly more quickly, confirming the improved performance of the basic evaluation function.

The evaluation of the CNN was strikingly similar to the evaluation of the naive model (Fig. 5C,G). Upon further investigation, we found that the trained CNN was always yielding quite small evaluations. This observation completely explains the observed behaviors, the evaluation will be very similar to the naive model. These results indicate that our

CNN was unable to learn to understand these positions.

The random forest evaluation function demonstrates a marked improvement over both the naive and basic functions. The model quickly understands the position and returns a high strength evaluation after only a few iterations (Fig. 5H).

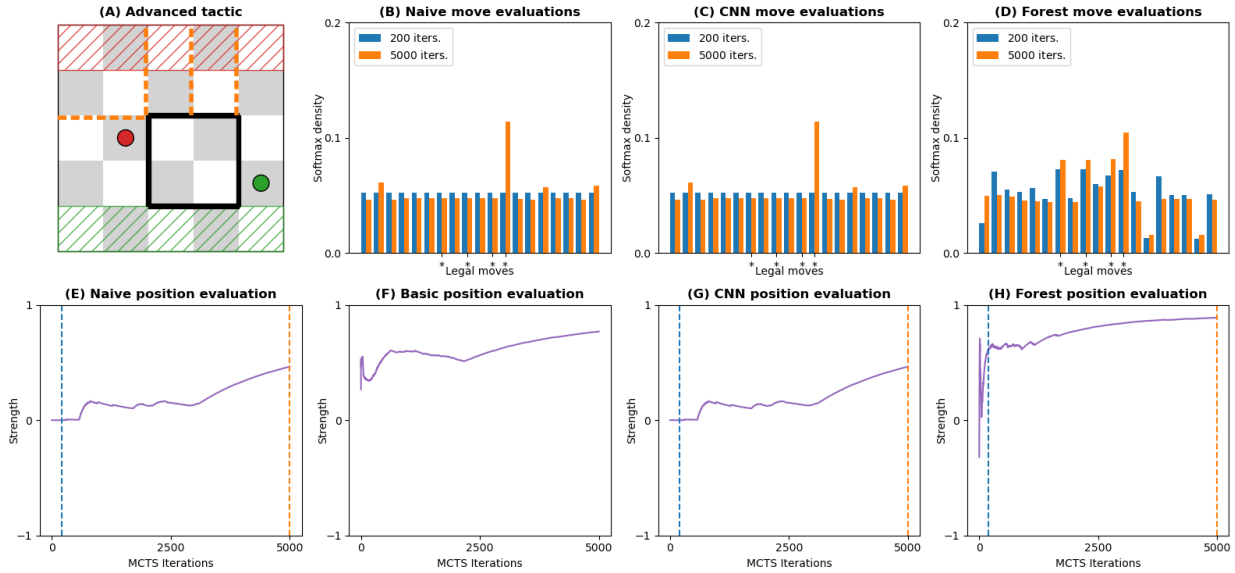

Figure 6. Advanced Strategy. (A) This puzzle is player 1 (red) to move. Each player has one fence remaining to place. Player 1 is closer to their goal and moves first, but they are susceptible to a fence blocking their path. If player 2 (green) is able to block player 1 with a fence, player 2 can force a win. Player 1 must employ an unintinitive move to maintain the lead: placing a fence to block their own inopportune paths to the goal. As the rules require that every player have a path to the goal, if a player deliberately blocks their own alternative paths, their opponent will not be allowed to block them with fences. There are four legal fence placements that player 1 could employ to accomplish this goal. These placements are denoted as dashed orange lines. These are the only winning moves for player 1. **(B-H)** See Figure 5.

3.4. Advanced Tactics

We repeated the position evaluation test with a more complicated position. This puzzle is carefully designed to require an unintuitive self-block as the only winning strategy (Fig. 6A). Models must have a concrete grasp of their susceptibility to their opponents fences to solve this position quickly. The naive, basic, and CNN models all solve the position with sufficient iterations (Fig. 6E-G). The random forest decision tree solves the position much more quickly (Fig. 6H), supporting our hypothesis that it has a stronger evaluation function. Furthermore, the random forest is the only model to find all four winning moves (Fig. 6D). The other models all stumble upon one of the winning moves and focus on it entirely, suggesting that they do not understand the position as deeply (Fig. 6B,C).

References

[1] Joseph Brown and Hamna Aslam. Monte carlo tree search for quoridor, 2018. Unpublished manuscript.

[2] A. LeNail. NN-SVG: Publication-Ready Neural Network Architecture Schematics. *Journal of Open Source Software*, 4(33):747, 2019.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4 edition, 2020.

[4] David Silver, Aja Huang, Chris Maddison, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 2016.

[5] Wikipedia contributors. Monte carlo tree search. `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`. Accessed: 2025-02-01.