Monte-Carlo approach with evaluation function bias

COMP 424: Pentago Swap AI Agent Report

McGill University

Author:
Michael Vaquier
michael.vaquier@mail.mcgill.ca

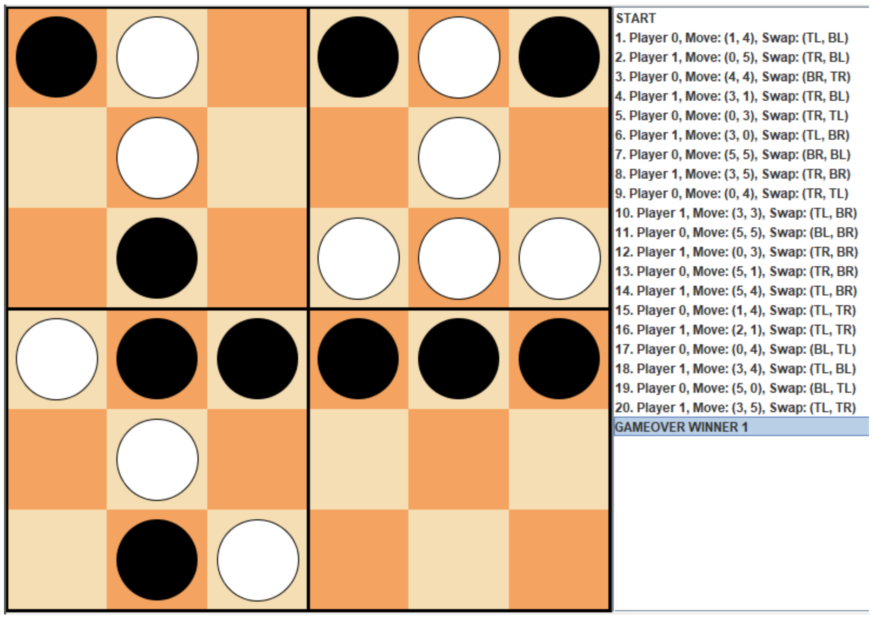
Professor:
Jackie Chi Kit Cheung
jcheung@cs.mcgill.ca

April 11th, 2019

Contents

1 About the Game

Pentago Swap is a variant of the not so popular game Pentago. The game is perfectly deterministic with no hidden information which makes it ideal for developing an AI agent. The game is played on a 6x6 board with two players: the white player, who always plays first, and the black player. Each player plays one after the other by placing a piece, then swapping two quadrants of the board. The goal of the game is to end up with 5 pieces in a row.



```
START
1. Player 0, Move: (1, 4), Swap: (TL, BL)
2. Player 1, Move: (0, 5), Swap: (TR, BL)
3. Player 0, Move: (4, 4), Swap: (BR, TR)
4. Player 1, Move: (3, 1), Swap: (TR, BL)
5. Player 0, Move: (0, 3), Swap: (TR, TL)
6. Player 1, Move: (3, 0), Swap: (TL, BR)
7. Player 0, Move: (5, 5), Swap: (BR, BL)
8. Player 1, Move: (3, 5), Swap: (TR, BR)
9. Player 0, Move: (0, 4), Swap: (TR, TL)
10. Player 1, Move: (3, 3), Swap: (TL, BR)
11. Player 0, Move: (5, 5), Swap: (BL, BR)
12. Player 1, Move: (0, 3), Swap: (TR, BR)
13. Player 0, Move: (5, 1), Swap: (TR, BR)
14. Player 1, Move: (5, 4), Swap: (TL, BR)
15. Player 0, Move: (1, 4), Swap: (TL, TR)
16. Player 1, Move: (2, 1), Swap: (TL, TR)
17. Player 0, Move: (0, 4), Swap: (BL, TL)
18. Player 1, Move: (3, 4), Swap: (TL, BL)
19. Player 0, Move: (5, 0), Swap: (BL, TL)
20. Player 1, Move: (3, 5), Swap: (TL, TR)
GAMEOVER WINNER 1
```

Figure 1: Example of then end state of a game.

Figure 1 shows the end state of an example game (4th row). The black player won the game after swapping its 5 contiguous pieces into place. It can be seen that the white player also has a few opportunities to get 5 pieces in a row (vertical, and diagonal) making this quite a close game.

2 Strategy

The strategy taken by this AI agent consists of a simple Monte-Carlo method with an Upper Confidence Tree (UCT) as the tree policy. It does however have some filtering capabilities to eliminate certain moves from its set of possible moves because they would be likely to lead to an inevitable defeat. When the move to play is chosen at the end of the turn after running many default policy simulations, it is picked based on maximizing a function of the win-rate and evaluation function of the node.

The strategy was primarily developed by continuously playing the game against the agent to slowly understand what the definition of a good, and bad move is. After playing many times, a more formal and clear definition of a bad move that should not be played, was drafted. The agent can then remove these bad moves from its set of considered moves to highly increase its chances of not losing in the very short run (one or two moves ahead). More details are discussed on this filtering process is discussed in the Pre-Exploration Filtering section.
As for the definition of good moves, patterns in the board that can often leads to the cornering the opponent are identified and encapsulated in an evaluation function. This evaluation function can bias the agent to perform these good moves moves rather than completely relying on the statistics gathered from the Monte-Carlo Tree Search (MCTS).

2.1 Motivation for the Strategy

2.1.1 Choosing a Base Algorithm

The reason why the strategy was built around base of a MCTS algorithm is due to the very high branching factor of the game. The upper bound of the branching factor is 216 when the board is empty. This branching factor then decreases by 6 every time a move is played. Because of limitations of the minimax algorithm (even with alpha-beta pruning) it would be very unlikely to go deep enough into the search tree to be able to play well.

2.1.2 Benchmark Agent

To confirm this, a benchmark agent was developed using the alpha beta pruning approach with a (almost entirely) random evaluation function (Seen in Algorithm 1). It was then confirmed that the minimax algorithm would not be able to go any deeper than two levels (the agent move, and the opponent move) within the time restrictions to play a move.

Algorithm 1 Random Evaluation Function

```
1: if $agentWon$ then
       return MAX_INT
2: if $opponentWon$ then
       return MIN_INT
3: if $draw$ then
       return MAX_INT / 2
4: return (int) ($Math.random() * 1000$)
```

2.1.3 Beating the Benchmark Agent

Once the basic MCTS algorithm was implemented, it was tested by playing against the benchmark agent. Surprisingly, the MCTS algorithm was not able to beat the minimax algorithm that was using a random evaluation function. By watching many games of the two agents playing against each other, it was identified that the key not to lose a game was to carefully swap the quadrants to prevent the opponent to be able to win in one move. Because the minimax algorithm was able to go two levels deep in the game tree, it was able to identify which moves lead to the opponent being one move away from winning and avoided them at all costs. On the other hand, because default MCTS algorithm chooses the best move to play based on win rate statistics of the nodes at the first level, it was not able to capture this notion of *bad move*. This is where the motivation for pre-exploration filtering came from. After filtering all moves that would inevitably lead to a loss, the agent would then rely on the MCTS statistics to chose its move.

2.1.4 Beating the Benchmark Agent... Fast

After being able to win consistently against the benchmark agent, an offensive strategy was needed to get to the win faster. By watching and playing against the agent, some strategies and patterns were identified that it would sometimes make that would lead it to a quick victory. These were cases where no matter what the opponent played next, the agent would win (a critical state). This was the motivation to introduce an evaluation function which would help the agent to pick a *good move* by combining it with the win rate statistics (More details in the Chose move section).

2.2 Evaluation Function

The evaluation function is used to determine how good the current board state is for the Agent. For simple cases, the evaluation of the board follows the logic seen in Algorithm 2.

Algorithm 2 Evaluation Function

```
1: if $agentWon$ then
        return MAX_INT
2: if $opponentWon$ then
        return MIN_INT
3: return $evaluateBoard(agent)$
```

The evaluation function is highly reliant on pattern matching to identify sets of two or three pieces or more complicated patterns called *critical states*. The board score is therefore calculated in two main parts. Identifying critical states, and matching sets. From here forward, assume that the agent is playing with the white pieces.

2.2.1 Critical State

If the opponent is in a critical state, the evaluation of the board will return MAX_INT. A critical state is a state of the board when the opponent is cornered and will lead to the agent being able to win in one move no matter what the opponent plays. With experience, some of these critical states configurations were identified.

The first critical state is the *four in a row*. When the agent has four pieces in a row with **both** ends **not** occupied by the opponent. This critical state can be seen in Figure 2, where no matter where the black player plays, the white player will be able to win.

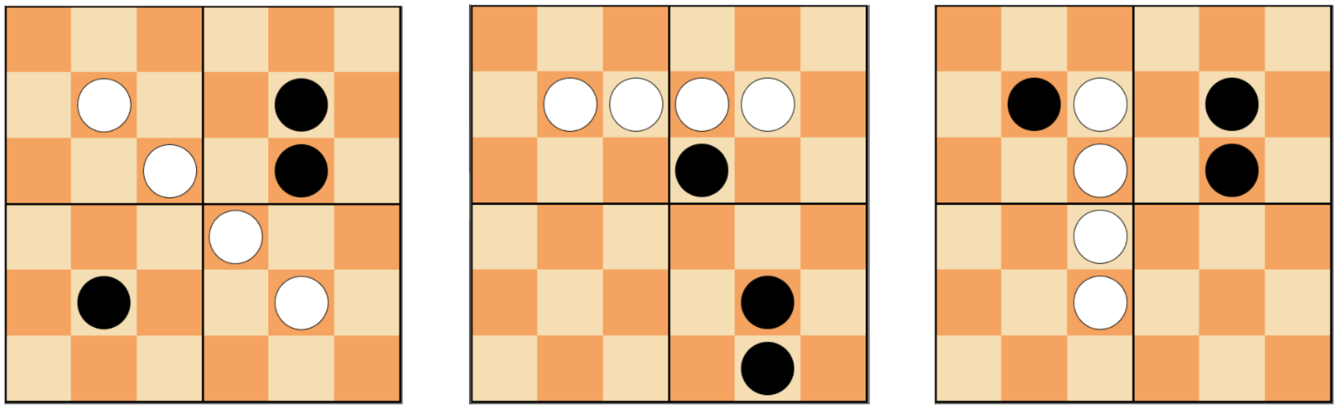


Figure 2: Critical state for the black opponent for the *four in a row*.

The four in a row can be located anywhere in the board in any direction. It is very important that both ends are not occupied by the opponent, otherwise it could be blocked. This is a very quick way to win a game if the opponent is not very defensive in the early game.

The other critical state that has been identified is the *triplet corner*. The *triplet corner* happens when the agent has a set of three pieces in a quadrant as well as the matching middle piece in two other quadrants. This time it is important that one of the non triplet middle pieces is completely unblocked from the opponent. The other is allowed to have a block on one of the sides.

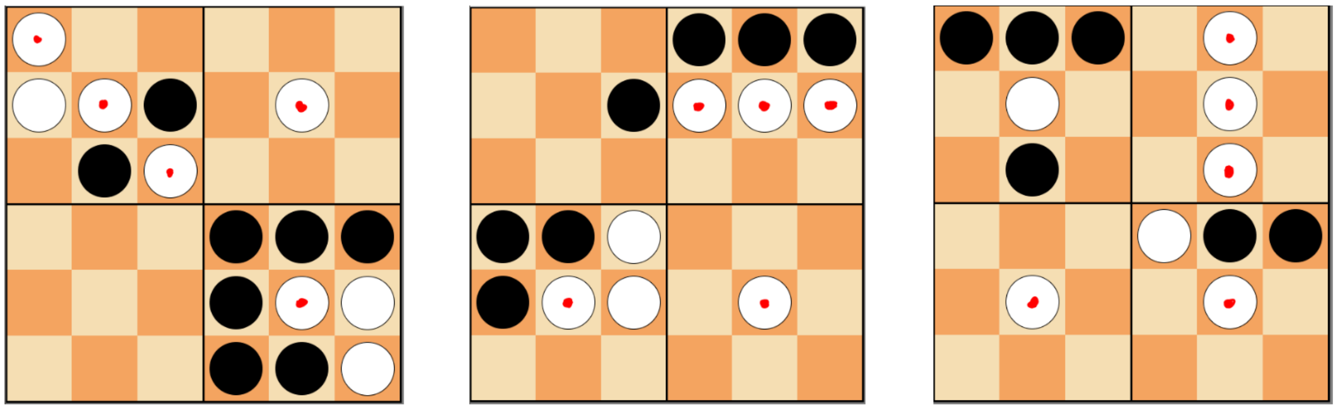


Figure 3: Critical state for the black opponent for the *triplet croner*.
Note: The pieces important to the critical state are marked with a red dot.

Figure 3 shows some example of the *triplet corner* critical state. This works in all directions, However it is important to note that for the diagonal cases, it it possible for the black player to block this case, but this move would still be a lot of pressure, and would likely lead to a win for the agent.

2.2.2 Finding Valuable Patterns

The second component of the evaluation of the board will only happen if the board is not in a critical state for the opponent. This part consists of matching patterns of the board by giving a reward for **making pairs of two**, having the **same pair in different quadrants**, or **blocking an opponent pair**. This same score will also be calculated for the opponent and subtracted from the agent score to get the final number for the evaluation of the board.

$$evaluateBoard(agent) = setBonusScore(agent) - setBonusScore(opponent)$$

All pairs calculated in this evaluation function have to be in the same quadrant. This is because of the nature of the game, having consecutive pieces in different quadrant is not very valuable as the pieces can be separated by a simple swap. Figure 4 shows an example of a pair in the same quadrant and an example of a pair with pieces in different quadrants.

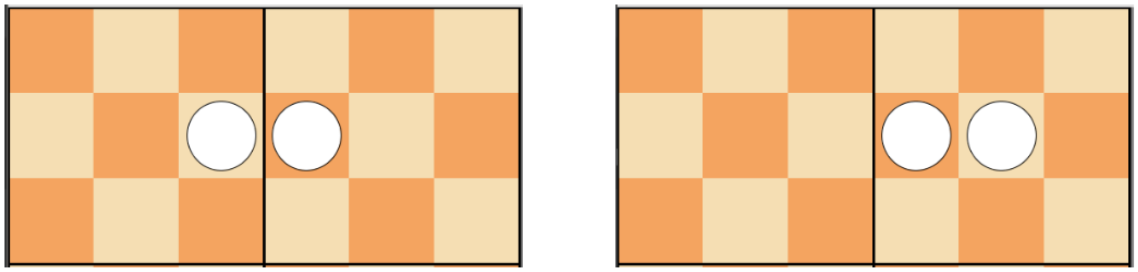


Figure 4: Pairs that are in the same quadrant are much more valuable than a pair where pieces are from different quadrants.

1. Making Pairs of Two

Making pairs of two pieces in the same quadrant is very simple and is valuable in order to put some pressure on the opponent as this gets the agent closer to getting five pieces in a row. The agent is therefore rewarded for making any permutation of a pair of two within a quadrant. It is important to note that three pieces in a row counts for the bonus for two sets of two with only three pieces. As the agent tries to increase its score fast, this inherently encourages it to make sets of three. Figure 5 shows a few examples of pairs. Pairs can be in any direction anywhere in the quadrant.

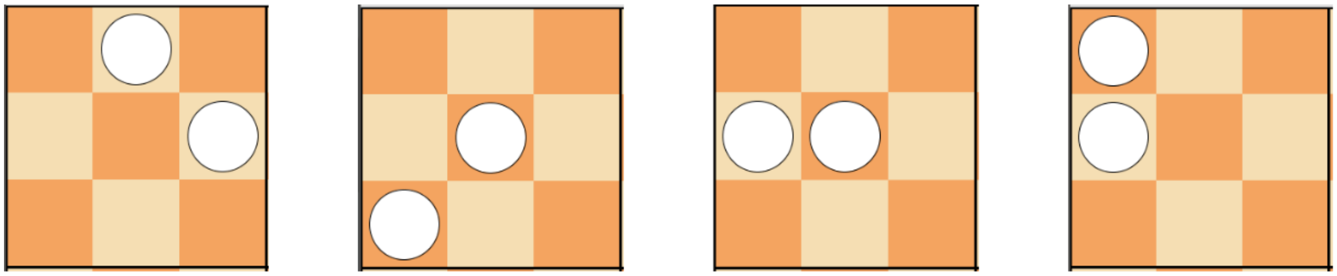


Figure 5: Pairs in the same quadrant rewards the agent.
Any permutation of two consecutive pieces is considered a pair.

2. Same Pair in Different Quadrants

Having the same pair in different quadrants further rewards the agent. This is because this would mean that the agent is one piece away from getting five in a row if swapped correctly. This reward can lead to the agent to inherently create matching, more complicated patterns in different quadrants. The pattern seen in Figure 6 is very powerful as the agent could win with the diagonal, the vertical, and the horizontal.

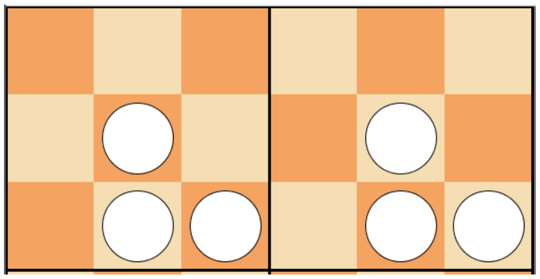


Figure 6: Example of a complex pattern that the agent is encouraged to make.

3. Blocking the Opponent

Lastly, the agent is rewarded by blocking the third spot that would lead to the opponent pair becoming a triplet. This is important to keep a good balance of defensive and offensive moves.

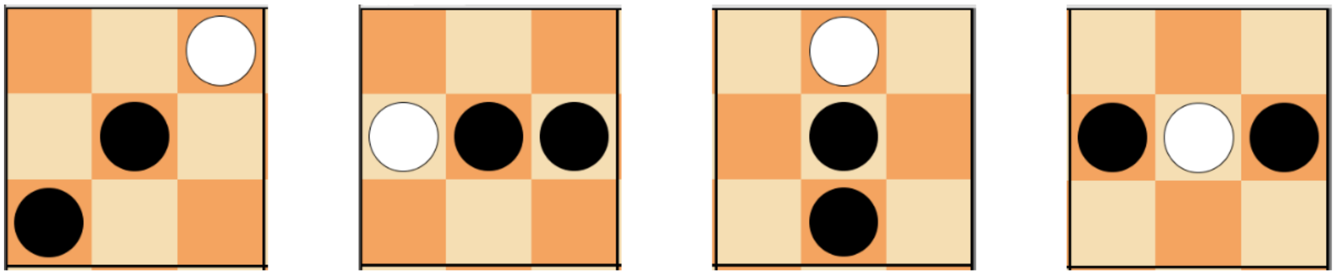


Figure 7: Preventing the opponent from getting three in a row is valuable to the agent.

Figure 7 Shows some example of the scenarios the agent receives a reward from blocking the last slot of an opponent pair. This feature combined with the previous can nudge the agent to play extremely valuable moves. Figure 8 shows an example where these concepts are combined. In one placement in the quadrant, the agent prevented the opponent of getting three in a row, and

created two new pairs for itself (vertical, and diagonal). This move is much more preferred than to complete a three in a row.

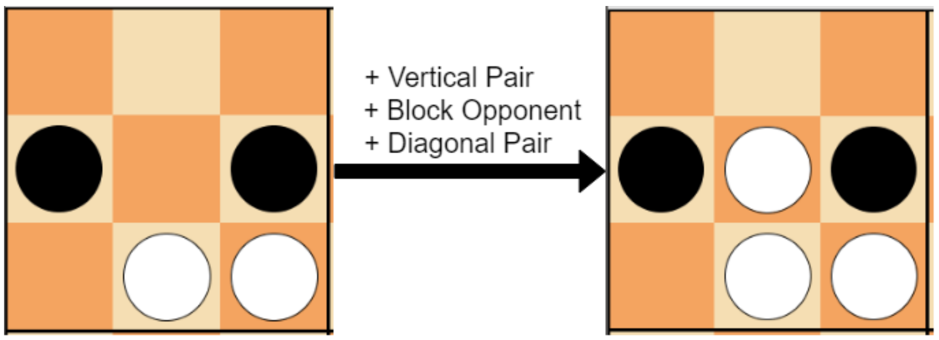


Figure 8: The agent blocks, an opponent pair, and creates two new pairs for itself

2.3 Steps to Play a Move

2.3.1 Pre-Exploration Filtering

The pre-exploration filtering is used to eliminate some obvious *bad moves* which could lead to an inevitable defeat from the set of considered moves. A *bad move* has one of the following characteristics:

1. Never play a move that leads to the opponent winning.

This seems very intuitive but it is something that needs to be explicitly told to the agent. As seen in Figure 9, the node directly under the current state (CS) of the board where the agent loses is not considered. The node is denoted in red.

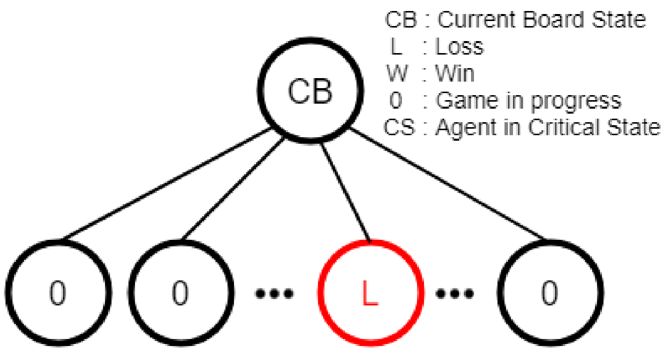


Figure 9: The agent filters out moves that leads to the opponent winning.

2. Never play a move that can let the opponent win in one move.
It is assumed that anyone or playing against the agent will be able to identify that they can win in one move. The agent should never play a move that puts the opponent in this position where it can win in one move. In Figure 10, the move in red is removed because the opponent can win on one move from it.

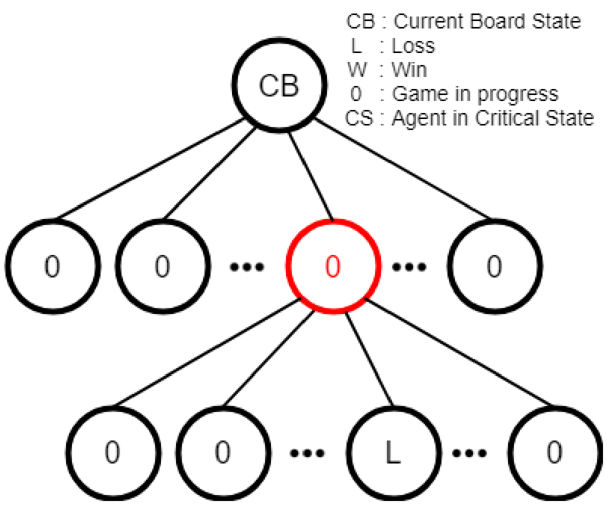


Figure 10: The agent filters out moves where the opponent can win in one move after it.

3. Never play a move that gives the opponent an opportunity to put the agent in a critical state.

The last *bad move* that the agent filters out is a bit more complicated. It is made to defend against the same kind of offensive strategy that it uses. The goal is to filter out all moves that can lead to the opponent playing a move that puts the agent in a critical state. This move is however still considered is the agent can win in one move even if it is in a critical state. Figure 11 shows how the agent does not remove the node depicted in green because even if the agent can enter a critical state, it will be able to immediately win from it. On the other hand, the node depicted in red is not considered because the agent could potentially enter a critical state where it will likely lost two moves later. This is extremely powerful as it can selectively search deeper in the tree when necessary to ensure that the agent does not perform any obvious *bad moves*.

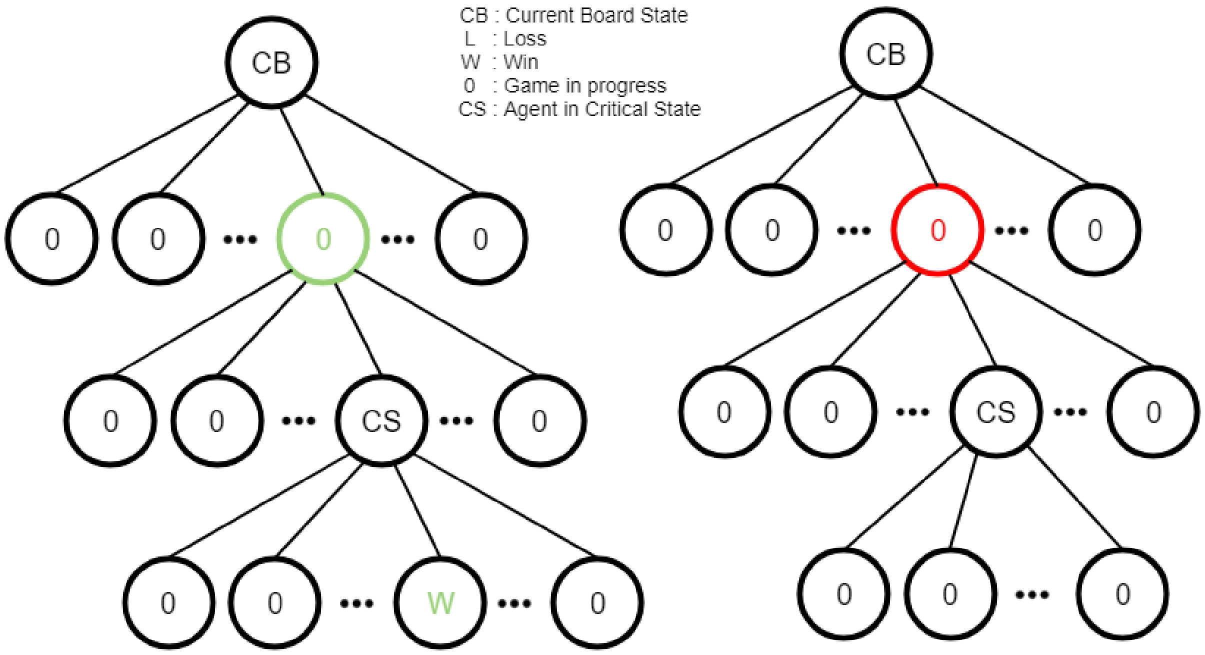


Figure 11: The agent filters out moves where the opponent
can put the agent in a critical state, unless the agent can win from it.

2.3.2 Monte-Carlo Method with UCT

After all *bad moves* are filtered out, the simulations stage starts. Nothing fancy is done here, basic Monte-Carlo simulations are performed with an upper confidence tree (UCT). The node selection process uses the formula:

$$nodeUCTValue() = winRate + \sqrt{2 * \frac{parentVisitCount}{vistCount}}$$

This balances the exploration of slightly unpromising nodes and the exportation of promising nodes. When a node is selected, the default policy is used to rollout until the game is over. The value that is back propagated is 1 for a win, -1 for a loss and 0 for a draw.

2.3.3 Choosing the Move

Choosing the move is a very important state to efficiently corner the opponent. At this point there is less worries about potentially playing a move that would put the agent in a dangerous position because most *bad moves* were filtered out in the filtering step. What is now important is to make use of the evaluation function to nudge the agent towards a state where the opponent is in a critical state which will likely lead to a victory. The agent therefore combines the evaluation function with the win rate of the node as the metric to chose the move to play.

$$moveToPlay = \max_{m \in moves} \{m.getWinRate() + 0.05 * m.getBoardHeuristic()\}$$

3 Analysis

3.1 Advantages

The main advantage of this method is that it is able to encapsulate the advantages of both the MCTS and minimax algorithm. *Bad moves* can be filtered out in the pre-exploration filtering step by traversing the first two levels of the game tree the same way as minimax does. However, the dependence on the assumption that the opponent is playing optimally with respect to the same evaluation function is removed. Instead, the evaluation function is only used to nudge the agent to play what is considered *good moves*, but will still rely on the win rate statistics from the Monte-Carlo method. Furthermore, the nudges from the evaluation function encourages the agent to play moves that create winning opportunities while blocking the opponent from doing the same.

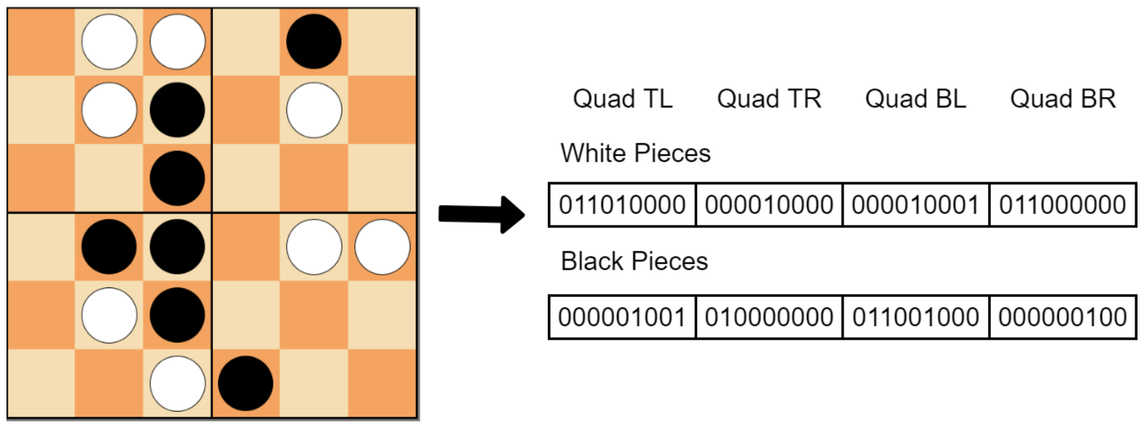
3.2 Disadvantages

The agent's lack of ability to plan forward is one of its major weaknesses. The agent is able to prevent a loss on the opponent's next move, and sometimes two moves ahead, when checking for critical states. However, this is completely dependent on the defined critical states. There are only two types of critical states defined to the agent because they are computationally easy to identify. The agent ignores many more critical states because it is not aware of them. It would be computationally impractical to define them all in order to truly check if a move is safe to play two moves ahead for the opponent.

Another potential weakness of the agent is that, in order to improve its offensive and defensive weaknesses, it mostly played against itself or humans. Its offensive and defensive strategies are optimized to play against itself. Therefore, it is hard to tell how well it will perform against agents developed by others because they might use different strategies.

4 Further Improvements

An improvement that can be be made to the agent is to make further optimizations to the *PentagoBoardState* class. A custom class was made to be used instead, adding useful methods such as reverting a move and evaluating the board. Bit-wise operations were taken advantage of in order to make the evaluation function to run vary fast especially for pattern matching. The board state was first converted to an integer format where the individual bits would represent whether or not a piece is present at that location. Figure 12 shows an example of this new representation. This way a bit-mask can be made for all patterns to check. To see if the pattern is present in the quadrant the simple operation $quadInt \ \& \ bitMask == bitMask$ can be used.



	Quad TL	Quad TR	Quad BL	Quad BR
White Pieces	011010000	000010000	000010001	011000000
Black Pieces	000001001	010000000	011001000	000000100

Figure 12: The board is represented as two arrays of integers of size 4.

Because the integer representation of the board has to be calculated every time the evaluation of the board state is calculated, it would make sense to always store the information in this format. All methods of the default *PentagoBoardState* class could be re-written to support this data format. This would make the overall performance of the class much better by taking advantage of bit-wise operations optimizing the runtime.