COMP 424 Final Project (Pentago-Swap)

Zhao, Zhuoran
260716696
zhuoran.zhao@mail.mcgill.ca

***Abstract*—Pentago-Swap is a variation on popular game named Pentago. This game falls into the Moku family of games, which also include popular games such as Tic-Tac-Toe and Connect-4. The biggest change in Pentago-Swap is that the board is divided into 4 3-by-3 quadrants, which for each move the player could not only place a move but also swap any two of four quadrants on the board. The goal of this project is to construct an AI agent to play this game and get a good performance on it.**

I. Introduction

In this project we are aiming at implementing an AI agent play on the board game named Pentago-Swap. Pentago-Swap is a two-player game played on a 6x6 board, which consists of four 3x3 quadrants. To begin, the board is empty, and the first player plays as white and second player plays as black. In order to win the game, each player tries to achieve 5 pieces of their colour in a row before their opponent does. (which can be achieved horizontally, vertically, or diagonally) Draw will be declared if either both players have 5 in a row or the board is occupied without a winner. The additional rules for this project is that it constraints the player for 30 seconds to choose first move and no more than 2 seconds to choose for subsequent moves. Also the memory should not exceed 520 mbs. In this report, I will introduce the technical approach and motivation for my AI agent design and comparison about advantages and disadvantages associated with it also with other approaches then discuss the improvement and further work.

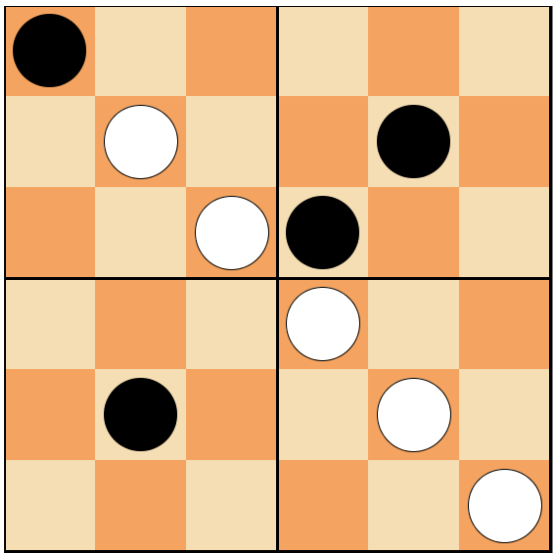

Fig. 1. Win State for Player-0

II. Technical Approaches and Motivation

After several trials and experiments, our AI agent is implemented using a recursive Minimax Algorithm with alpha-beta pruning, searching by iterative deepening with minimum setup depth 2. For each move in the search tree, a linear evaluation function will evaluate the value of current board and thus link the result to the minimax algorithm. For our AI agent, there are two separate evaluation functions, one is used when our player is white(in this case, we will move first in the game), the other is used when our player is black(we move second in the game). The algorithm will either be stopped by reaching the end depth or be interrupted by the 2 second time slot then return the value of move. Inside our chooseMove function we have a list that store the best moves using our evaluation function, we will choose the best move depending on the value of this move.

A. Minimax Algorithm

1) Motivation: As a wide used strategy in 2 players zero-sum game, minimax algorithm is a decision rule that maximizes the minimum gain.(1) This strategy consists constructing a search tree that captures all possible moves from a given state, and each move can be represented as gain/loss for one player. Consider Pentago-Swap is for each player to maximum its gain, Minimax algorithm is a good choice for our game. At the mean time, consider for Pentago-Swap, there are 36 grids for placing a move and 6 ways of swapping, in total the search tree could be very big due to the number of legal moves. In this case, the searching time could be very long. In order to solve this problem, alpha-beta pruning is used to prune the tree. Since Alphabeta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree.(2)

2) Description: The recursive minimax associated with the alpha beta pruning strategy. For each iteration of the minimax search we start from depth =0 and max depth=2, then add depth by one for the next search. At mean time we use a list to keep track of the best few options. Then using pruning to stop the function if it have one improvement. Compared to the normal minimax tree, using pruning may prunes the branches that could have high value in further step. Since our game involve swapping in each move, I change the weight of evaluation function to avoid pruning the good moves. One pseudo-code can be find below that describe how this minimax function worked.

Algorithm 1 Minimax

```
if depthismaxDepth then return (evaluation(board), null)
bestMove ← null
if isOurPlayer then
forall legal movesprocessMove(move)
if(isWinner)return(max, move)
else MiniMax(depth+1)
   if alpha < resultValue then
    alpha = value
    bestMove = resultMove
   if alpha >= beta then
      bestMove = (beta, null)
       return bestMove

If (isOpponent) then
forall legal moves processMove(move)
      if (Winner) then return (min, move)
      else: MiniMax(depth+1)
      if beta > resultValue then
       alpha = value
       bestMove = resultMove
      if alpha >= beta then
          bestMove = (beta, null) bestMove
```

B. Iterative Deepening Search

1) Motivation: Since the time constraint for the AI agent to choose a subsequent move is only 2 seconds, it is not possible to search for entire tree. And if we choose at fixed depth, it will either be not deep enough to get the best result or could not finish searching before time slot ends. So iterative deepening search will be the best choice. Since using iterative deepening, the depth is not fixed. After several trials, it can be found out that it is possible for us to perform at least 2 depth search, and thus we set up the minimum search depth for iterative deepening as 2.

2) Description: IDDFS calls DFS for different depths starting from an initial value. In every call, DFS is restricted from going beyond the given depth. So basically we do DFS in a BFS fashion.(3) In order to implement the iterative deepening search, we use the java.util.concurrent.ExecutorService and Futuremethods to control the time as 1.95s. However, there is one certain issue when we connect with other people's server: timeout problem could happen. After analyzing the functions, we find out that computation time varies between different CPUs thus 1.95s on our CPU may be longer than others'. Since timeout can cause a big loss, to be safe, we changed this value to only 1.70s. Then we use a new class that store both the move and the value, and a list storing the best two options we could give. When this function is called, the student player will continue searching for deeper depth until it is caught by the interrupt exception. If so, then the program will return the best move it have searched so far. We record how does the depth change with the increase of iteration. Since the depth we could search improved a lot, this method could significantly improve our performance.

Iteration	Average Depth
1-6	4.5
6-12	5.5
12-18	7
18+	very large

C. Evaluation Function

1) Motivation: When placing different trials on the practice games, we found out that the strategy should be different considering whether our AI is the first player or not, since the player who play first always has one more piece before the second player plays. Also considering the maximum steps could only be 36, every step could have large effect on result. Furthermore, considering the situation that the first player already have 3 or 4 in a row, the best strategy for the second player should block the first player instead of constructing its own win state. On the other hand, for the first player(white piece), as long as the second player's heuristic is lower than ours, it should focus on generating a win state that maximize the possible situations to win instead wasting the piece to block the opponent.

2) Description: There could be several ways to evaluate the board, after analysis for time saving and efficiency we simply choose to traverse the board once by go through all possible rows or columns or diagonals that could form a win state. By evaluating the board, we first go through the board and for each row, column or diagonal that could construct a win state, we calculate the number of white pieces and black pieces separately. Since there are 6 rows and 6 columns and only 6 diagonals that could form a win states, the algorithm will just go through these 18 possible lines then store the result of evaluation in the list for both white and black state. The figures that describe how we traverse the board can be found below:

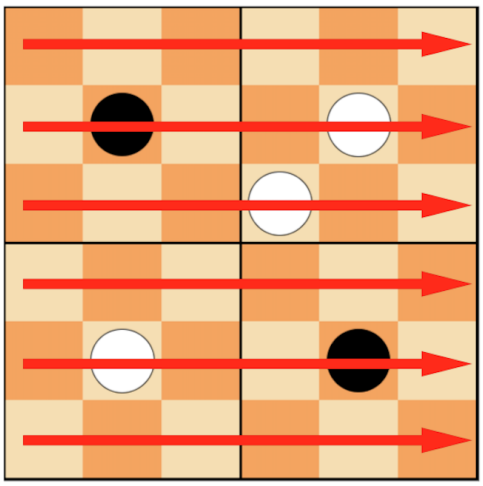

Fig. 2. Traverse by row

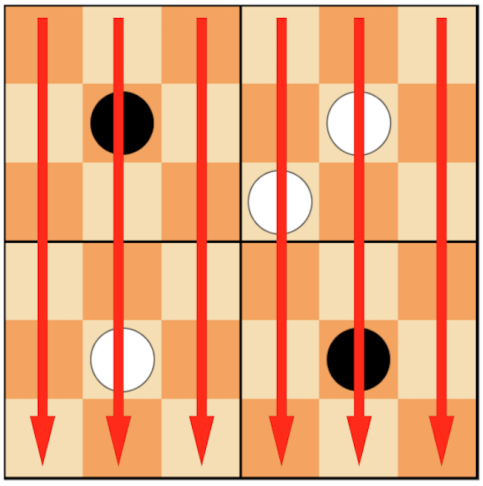

Fig. 3. Traverse by column

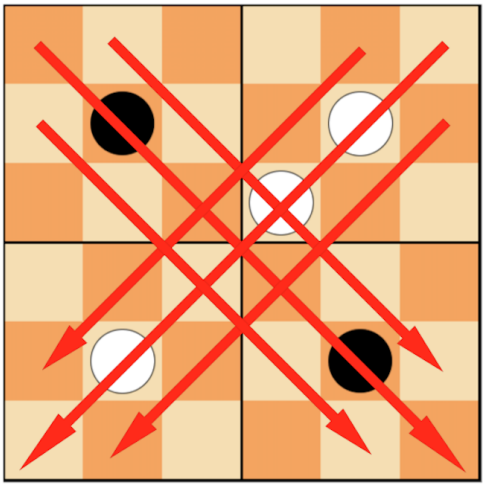

Fig. 4. Traverse by diagonal

Then we use two lists that contain the result of traversing the board as whiteScore and blackScore. For the score we keep tracking the follow cases:

1) 2 pieces in a row plus 3 empty pieces.
   In this case we could win the game by either fill the other three pieces or swap the board. However, since we only have 2 pieces in a row, this means the opponent have one piece in this row, even though we fill the empties, the probability for win is still relatively low, thus the weight is relatively small compared to other cases.
2) 2 pieces in a row plus 4 empty pieces
   In this case, 4 empty pieces means that the opponent did not put their piece in this row. Thus it can be easily form a 3+3 cases. Conversely, if this is achieved by our opponent, we need to block this row in order to defend the further win state for opponent. Thus with this case, the weight is a little bit higher compared to 2+3.
3) 3 pieces in a row plus 2 empty pieces.
   In this case we already have 3 pieces in a row, but only with 2 spaces left, putting one at one space, there is very large probability that he opponent will block the final result. Thus I give the weight to this case slightly larger than above cases.
4) 3 pieces in a row plus 3 empty pieces.
   In this case we will have a very high weight. Since 4+2 state will occurs if we just put one more piece in this row. Even though our opponent could block our win state and change this to a 4+1, we still have big chance to win. Thus with a very high probability to win, we will give this state a high weight.
5) 4 pieces in a row plus 1 empty pieces
   In this case, although it is very likely be blocked by the opponent, it is still a good state. Since blocking us may not form a good state for their own sate, thus wasting the opponent one piece to put into some place that may cause a useless piece for them. Then we can use this fact to have more chances to put our pieces as a better state.
6) 4 in a row plus 2 empty pieces
   This is just the case after we put our piece at the place where we have the 3+3 cases above. Since the opponent did not put anything in this row, even though they want to block us, it will be too late for them to do so. Thus among all states, this case have the highest weight.

For both black and white pieces, we keep track these 6 cases and store them both in two lists.

whiteScore:[4+2, 4+1, 3+3, 3+2, 2+4, 2+3]
blackScore:[4+2, 4+1, 3+3, 3+2, 2+4, 2+3]

Situation	Weight
4+2	w0 = 10
2+1	w1 = 7
3+3	w2 = 5
3+2	w3 = 3
2+4	w4 = 2
2+3	w5 = 1

The equation for value of white score and black score can be calculated:

$$W = \sum_{i=0}^{5} wi * W[i] \tag{1}$$

$$B = \sum_{i=0}^{5} wi * B[i] \tag{2}$$

Case we are white:

$$Value = W - 0.8 * B \tag{3}$$

Case we are black:

$$Value = B - 2 * W \tag{4}$$

Where we use W represent white's score and B represents the black's score. The reason for this setup is that, when we

are white (move first), we would like our player more likely to focus on getting better state for itself instead of defending others. However, when we are black (move last), we would like our player more like to defend, since the opponent always has one more chance than us. The weights for the equation are calculated by several trails. Since learning is not allowed, there could be a big improvement if we could learn for the weights of evaluation function.

D. Hard Coding

For the first two steps, we choose to hard code the center coordinates and put the first two steps in the middle, even the third one if the opponent did not do so. In the case that we are white player, we even hardcore the third step that let the player form a 3 plus 3 in a diagonal.

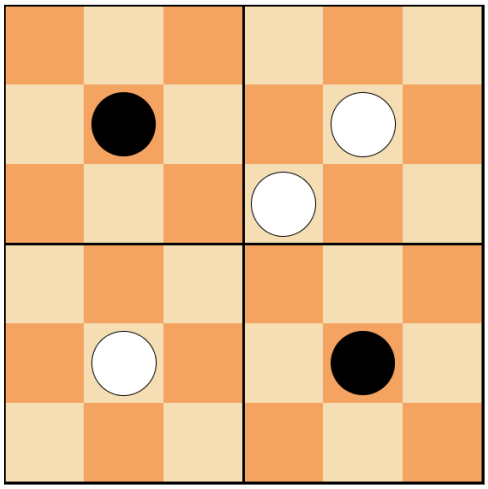

Fig. 5. First three steps hard coding for player-0

III. Advantages and Disadvantages

A. Advantages

1) Minimax strategy with alpha beta pruning will help us reduce time and increase efficiency. Thus we could go as deep as possible.
2) Different evaluation functions that consider whether we move first or not can help our player always choose the situation that is better for us.
3) Iterative deepening return the best move with time limit can let our player search as deep as possible.
4) Score calculation that only evaluate 18 different rows can be very efficient and save time for traversing the board, since we never do the same job twice.
5) Hard code the first three steps can help us to gain as much as win states as possible, since each center coordinates can control four rows. And a win states will never form without have pieces on the center coordinates.

B. Disadvantages

1) When considering how many pieces are in a row and how many pieces are available, we did not consider how does their position impact the result, for example although our piece already have 4 in this row, but their piece are located at the center of this line, even if we tried to put another one to form a 5 in a row plus 1 opponent piece. We could not form a win state. This case can be see as following graph.

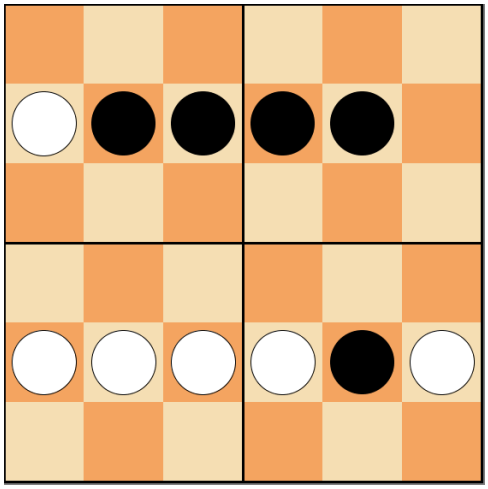

Fig. 6. Special Case

2) The minimax function works best when both our max player and min player plays optimal, consider our opponent is not optimal, the last move may be not the best choice for our player.
3) The draw state could form when both players use the optimal strategy.
4) Even the different moves could cause same result due to the fact that we could swap the board. In this case iterative deepening may repeatedly evaluate the same state that can waste some time.

IV. Other Approach and Comparisons

A. Evaluate by quadrants

One other approach that we tried is instead of evaluating by the board, evaluating by quadrants. Dividing the board as four quadrants, then for each quadrant, evaluate the pieces as three-in-row, two-in-row and so. However, there is one certain issue with this evaluation. When we calculating the value of moves, we will definitely give the value that could connect three in a row with a very high priority then the player will more like to perform this step. But if none of the quadrants could connect this three in a row as a win state, even if we fill one row with all our pieces, it may be useless. Same theory could be applied to the opponent, when we want to block the opponent, we need to consider whether it is already impossible for the opponent to win. If so, we should not waste our move and block that way. And after combining these ideas, we changed our strategy by just evaluating the whole board when we evaluate the move.

B. Non linear function

For the evaluation function, instead of linear one, we also tried the non-linear one. By multiplying all scores in the evaluation can give us the final result. However there is a certain issue with this way. Consider we use a large weight on some situation, whenever this situation happens we will only tend to form this result instead of thinking about other situations. But it is possible for assign relatively small weight to control, considering multiplication involves all part for each step, a lot of trails will be taken for get the relative good results. We could try this way for further improvement and experiments.

V. Discussion and Improvement

Although there are many advantages, there are still some disadvantages with this method. In order to perform a better agent, there are some other improvements we could try later.

A. Monte-Carlo Algorithm

One big improvement could be tried in the future is that using the first step's 30 seconds to form the monte-carlo that generate the best moves for the first few steps, then combine this with minimax to perform better when we go to the final moves. The focus of Monte Carlo tree search is on the analysis of the most promising moves, expanding the search tree based on random sampling of the search space. Although this method is that is not efficiency compared to Minimax algorithm, we could also solve this problem by using the alpha-beta pruning strategy to minimize the space. In particular, Monte Carlo tree search does not need an explicit evaluation function. Simply implementing the game's mechanics is sufficient to explore the search space (i.e. the generating of allowed moves in a given position and the game-end conditions). This could help us save our time. As such, Monte Carlo tree search can be employed in games without a developed theory or in general game playing.(5)

B. Priority Tree

Instead of just keeping the first best moves, we could use a priority queue to keep the best moves that will have a priority depend on the value of this move. Then the additional operation could be applied on this structure that may perform a deeper search in the tree.

C. Apply Learning

Another big improvement could be tried is that learn for the evaluation function. For the evaluation function, a slight change on the weight for each combination could cause a very huge effect on the performance. Also the learning could also applied on the monte-carlo search algorithm that can be very efficient.

Reinforcement Learning is an approach based on Markov Decision Process to make decisions. For reinforcement Learning we can pass a reward, positive or negative depending on the action the system took, and the algorithm needs to learn what actions can maximize the reward, and which need to be avoided. Deep Q-Learning increases the potentiality of Q-Learning, continuously updating the Q-table based on the prediction of future states. (4)For our evaluation function, by defining a loss function, in order to minimize the loss, the reinforcement learning strategy will provide us for the best solutions of the weights that could help us improve our performance.

References

[1] Wikipedia: Artificial Intelligence/Minimax
Retrived from *https://en.wikipedia.org/wiki/Minimax*

[2] Wikipedia: Artificial Intelligence/Alpha-beta pruning
Retrived from *https://en.wikipedia.org/wiki/Alpha-beta pruning*

[3] Retrived from *https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/*

[4] Mauro Comi (2018, November 5).
How to teach AI to play Games: Deep Reinforcement Learning
Retrieved from https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a

[5] Wikipedia: Artificial Intelligence/Monte Carlo Search Tree
Retrived from *https://en.wikipedia.org/wiki/Monte-Carlo tree search*