Building a Pentago-Twist AI Agent using Minimax and Alpha-Beta pruning

Alisa Gagina 260770497

alisa.gagina@mail.mcgill.ca

COMP 424 McGill University, QC, CA

1 INTRODUCTION

The code was initially forked from the Github repository [1], after which I edited the MyTools.java to hold all the functions I implemented, and Student.Java to hold all the calls to them.

1.1 How it works

To start off, the AI calls the method `StartingMoves` to check if any of the positions at (1,1), (1,4), (4,1), and (4,4) are open. These positions could be considered advantageous, as the opponent cannot mess with them by flipping / rotating the quadrant. This is also done to reduce the branching factor of our minimax algorithm.

After the AI runs the method `RootABMiniMax`, with depth of 3. This method returns a Pentago Move by evaluating values through the recursive method `ChildABMiniMax`, which has the same logic as `RootABMiniMax` but returns an integer. Before going into `ChildABMiniMax`, a check is performed to see if the move being considered is a winning one.

To avoid going overtime, a timer starts in `RootABMiniMax`. At 1.9s, if not done, the `RootABMiniMax` method is forced to quit the recursion and return its best calculation. With time, the allowed depth increases (becomes 6 after 8 turns, and 8 after 12 turns).

The values assigned to moves are calculated by the evaluation function `getValue` which holds the following heuristics:

- the number of consequent pieces horizontally, vertically, and diagonally (weight ~0-2000)
    - Weights are: two =2, three = 8, four = 30, five = 1000
- is the next move a winning move? (weight 5000)
- the number of four-piece combinations with a space in between (weight ~0-500)

These heuristics are first run for the AI. Then, they are run for the opponent, and the results are subtracted. For example, if the AI has four consequent pieces in a row somewhere (worth 1000), but the opponent is about to win next turn, the overall value is -4000.

The first heuristic accounts for the overall state of the game. The second heuristic is used to emphasize the selection of winning moves. Finally, the third heuristic is used to make up for the oversights of the first. For example, in a case where we have a row with *a black piece, two white pieces, a space, and another two white pieces*, the first heuristic would only assign it a worth of 4, even though only one move is needed to win.

1.2 Motivation

As the task is fully deterministic, with an adversarial opponent, I was stuck between two approaches: Minimax with Alpha Beta Pruning and Monte Carlo Tree Search. I decided to start off with Minimax, as I wanted to make my agent to find all the optimal solutions due to the consideration that the AI programs coded by the rest of the class will be more optimal than the RandomPlayer. I also thought that an evaluation function that would show how close an agent is to winning would be easier to design than a tree policy, as it is hard to tell what the winning moves would be at the very beginning (the games I ran required at least 6 moves to be won).

Of course, Minimax has problems, mostly due to the high branching factor. To this end, the depth that I assigned to the minimax algorithm at the very beginning was limited to 3. I also tried implementing a

Minimax & Iterative Deepening Search combo, which did not work out as I worsened the result against the random opponent, which will be further discussed in section 4.

2 THEORETICAL BASIS

The theories I used to build the Minimax Algorithm with Alpha-Beta Pruning followed the course material, as well as a YouTube video [2] I watched that had slightly clearer pseudo-code. The algorithm is split into two, as traditionally it should only return a number value, but we want a move instead. The only changes to traditional Minimax were the addition a timer check, as well as a manually chosen depth value that I got after experimenting. Minimax is a good choice when you have a good evaluation function. However, if some other agent has a better one, it might be beaten.

The board state evaluation function depends on three heuristics, where the weights that were assigned to the piece combinations were chosen after experimenting. As discussed in class, only the ordering matters, and the actual numbers not as much. I did, however, emphasize the five consecutive piece combinations (a winning move). Such a move is worth 1000, which far outweighs all the other configurations (for example, it would take 34 four-piece consecutives to outweigh a single five-piece consecutive). However, there is still a difference between winning combinations. For example, a case where we have a five consecutive and a four consecutive in a different spot is worth more than a case with a five consecutive and a two consecutive.

Alpha-beta pruning was used to reduce computation overhead as well as memory usage. This allows us to avoid some of the subtrees that have no chance of improving the result. However, as there is no special ordering to the moves explored, it is hard to explain just how many resources were saved. To further reduce computation, I manually limited the depth of the minimax search to make sure that we could go over all the branches rather than get stuck on one.

3 ANALYSIS

3.1 Advantages

Most of the innate minimax advantages hold: all the move combinations are checked until the timer runs out or the optimal move is found. It focuses on minimizing opponent's chances of winning, rather than considering only the agent's victory.

The program will never return a random move, as there is a timer making sure that the best possible move is saved and return even if the minimax calculations are not finished.

The algorithm improves as the game progresses. The increased allowed depth after the first eight moves allows the Minimax to get more specific board positions all the way until the end of the game.

The starting moves are already decided, and they prioritize taking positions that cannot be changed. Not only does this make the decision tree a bit smaller for when minimax runs, it also plays an advantage in terms of my evaluation function, as I did not include any turn / flip logic.

3.2 Disadvantages

A weakness of minimax is when playing against agents with different heuristics, optimal play is not ensured. Even more so, minimax assumes that the opponent is going for optimization, thus opponents that use completely random moves are a weakness. A good evaluation function is hard to design as the board state can be changed drastically each turn.

The algorithm does time out sometimes, which means that not all moves might be checked.

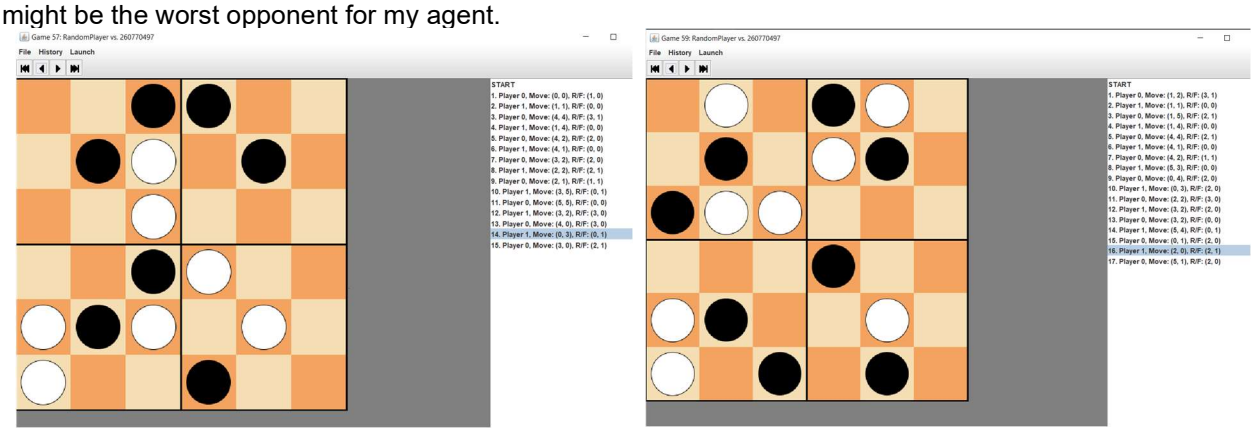
I manually chose the weight values as well as the times when the Minimax depth increases, there might have been better options out there.

Finally, the move ordering in the tree is randomized rather than calculated, which can cause loss of effectiveness with alpha-beta pruning, as it gives just average results.

3.3 Weaknesses

Specific weaknesses of my approach are the vulnerability to turns and flips, as I could not create a heuristic function that accounts for those. Figure 1 shows two games that my agent (black) lost. In game a), the opponent (white) only has to place a piece in slot (3,0) and flip that quadrant to win. My evaluation function would assign this board state a low value, as there are no four consecutive pieces anywhere on the board. In game b), my agent completely missed the two possibilities where the opponent can place a piece and win. It can either be done by putting a piece in spot (3,1), or by adding a piece to (5,1) and rotating that quadrant.

I have already mentioned the weakness to random, suboptimal moves, so in a way this Random Player might be the worst opponent for my agent.



a) b)

Figure 1: Loosing situations: a) Player 0 Move (3,0), R/F: (2,1) b) Either Player 0 Move (5,1), R/F: (2,0) or Move (3,1), R/F (0,0)

4 THE PROCESS

I started by coding minimax with pruning and just one heuristic: the number of consecutive pieces present on the board. After running 100 instances against the random player, my win rate was at 89%. Most of the time, it was due to turning quadrants that connected pieces of two and three. My heuristic would give this occurrence a low score, not matter the presence of a winning move. From this I learnt two things: I needed a separate 'winning move preventor' and a heuristic more suited to accounting for gaps (ex. `_xx_xxx or xxxx_x_`). A heuristic for turns / flips would also be helpful but I had no idea on how to code one.

After adding the second heuristic, that reweighs the win State, I was at 97%-win rate, the best so far. At this point, I decided that it was time to test how the agent fares against a human. It turns out, that I could win somewhat easily as the agent would fail to block my winning moves.

Afterwards, I started off by coding a function that would look two moves ahead instead of one, to check if any of the opponent's moves after the move that my agent is considering could make my agent lose the game. It was put beside the function that checks if any of the current move possibilities will make the agent win. It turns out that making this function was a mistake. One version just made the agent skip the current move, which brought down the win rate to 78%. Another version, where we tried to put our piece on the spot where the opponent would presumably put theirs, returned a win rate of 82%.

After scrapping that idea, I added the third heuristic, which accounted for empty spaces as discussed above. The win rate did not change, it stayed at 97%. Interestingly enough, after another round of testing the agent against myself, it became less oblivious to my winning moves and started blocking them better. It might be because I favor the strategy of connecting two pairs of pieces by one in the middle, instead of building them up in a chain.

Then I started playing around with adding an IDS to my Minimax function, instead of manually picking the depth. I ran into a problem when writing a transposition table, as running IDS by itself is too costly in terms of space and memory. Looking up an already calculated value of a previously seen board state is easier than calculating it anew. I ran into difficulties with the implementation of said table, as the board configurations had to be hashed (using Zobrist hashing), but it seemed sometimes different positions were getting mapped to the same hash entry, thus causing collisions. Another issue that came up was storing the alpha and beta values. As the trees was randomized between different possible moves, the stored values only applied to the current move's tree, not all of them. Thus, no computation speed was saved as when compared to plain minimax.

I decided to scrap the idea of IDS, and instead experimented with randomly choosing the depths instead. Surprisingly, a series of smaller depth increases, (ie. depth 4 at 8 moves, 5 at 10, 6 at 12) returned a worse result (93%) than two steep increases (depth 6 at 8 moves and 8 at 12 with a 97% win rate). I have no idea as to why it happened, and it is not due to faulty data either, as I ran the variations 300 times each.

I also experimented with heuristic weight functions, for example by decreasing the difference between four piece and five-piece consecutives. Using the numbers I have previously mentioned, I've achieved up to 99% accuracy.

5 FUTURE IMPROVEMENTS

There are two main areas that I would want to improve on: the addition of a heuristic that would account for turns and flips, and the implementation of either Monte Carlo Tree Search or IDS to figure out which moves to prioritize for the Minimax.

A possible heuristic for flips and turns could be the calculation of just how many same colour pieces the horizontally and vertically adjacent quadrants have. It is uncertain if it would work though and gives less weight to the diagonal winning moves.

Using Monte Carlo would allow me to pick n best moves and run a full depth minimax on them. However, it would be hard to pick a tree policy which can pick what the best move combination is. It could, of course, be as simple as the number of moves needed to win, however this can change all too easily with the turning and flipping that's happening.

I could also run some calculations in the 30 second window given at the start, for example to build up a tree of all possible moves in advance, instead of relying on my own starting_moves function. This would not last for long though, as 30 seconds would not be enough to build out the whole game. So, we could theoretically know all the values for, say, five moves in, which is hardly enough to truly win the game. It would only give us more time to calculate in the next few starting moves, as tree look up is easy. Alternatively, I could do some supervised learning instead of minimax now, and store those results in a file, to check if there are any unbeatable winning move combinations.

6 REFERENCES

[1] S. Yeasar, "Pentago Twist," 2021. [Online]. Available: https://github.com/SaminYeasar/pentago_twist.

[2] S. Lague, "Algorithms Explained – minimax and alpha-beta pruning," 20 April 2018. [Online]. Available: https://www.youtube.com/watch?v=l-hh51ncgDI.