COMP424 Pentago-Swap Report

Kyle Rubenok - 260667187 - kyle@rubenok.ca

Introduction and Background

As stated in the project description, the goal of this project is to create an AI agent that can play, and hopefully win a game of Pentago-Swap as implemented by the Project TA, Matt Grenander. Pentago's original form is very similar to Tic-Tac-Toe with the modification on a six by six board and requiring a line of length 5 to win. The "swap" is a wrench thrown into the game to make it significantly more complex. In addition to placing a piece, Pentago-Swap requires the player to exchange two quadrants of the board increasing the breadth of the game tree significantly.

In order to implement this AI agent, we are provided a working game implemented in Java. The game implementation features a server that allows students to simulate competitions between our AI's to tune their performance. We rely heavily on the `PentagoBoardState` class to communicate the status of the board to our method. Additionally, we are provided a player which will make random moves to test our agent against.

Technical Approach

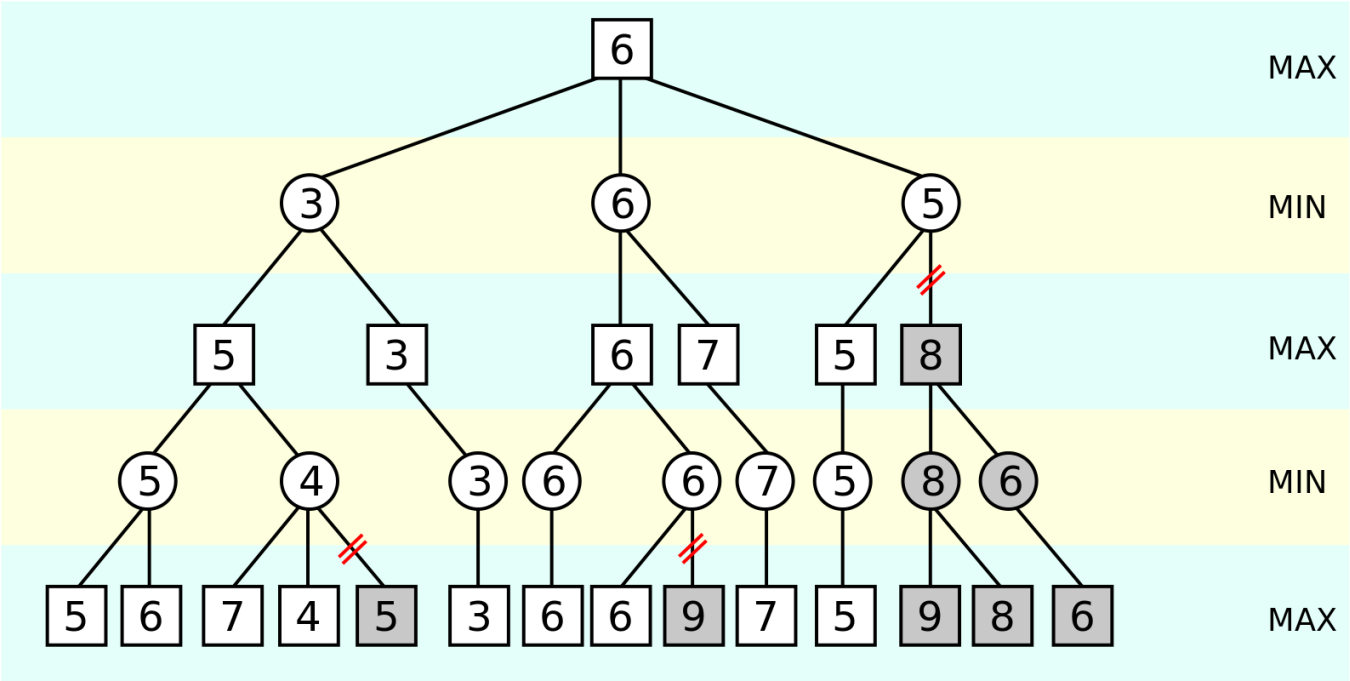
When faced with the problem of building this AI, I began to research implementations of AI agents in similar games to gain some inspiration for how I could implement something similar and what approach would be most effective. I settled on reseaching Tic-Tac-Toe and Connect4 implementations and evaluation functions for inspiration. I came across an implementation of Tic-Tac-Toe that used various different AI techniques seen in class. I started with the `MiniMax.java` implementation from LazoCoder [1] as it was very simple to implement and it would allow me to test the validity of this family of algorithms to confirm viability while temporarily ignoring the runtime disadvantages that it presents.

Running my own initial implementation of minimax seemed effective against the Random Player and against me as an admittedly poor player of Pentago-Swap. Due to time constraints, I decided I would not test alternative families of AI algorithms such as Monte Carlo Tree Search, or machine learning techniques such as Reinforcement Learning. I decided my primary goal would be to improve the efficiency of my MiniMax implementation by adopting AlphaBeta Pruning to reduce the number of computations required and stay within the 2 second limit imposed by the competition.

Alpha Beta Pruning Theory

Alpha Beta Pruning is an evolution of the Minimax Algorithm that allows several of the nodes in the game tree to be pruned and not evaluated. Depending on the complexity of the evaluation function and the order in which the tree is expanded, pruning presents a very significant speedup to the minimax technique that can result in more efficient run times. Given the two-second limitation imposed by the competition, any time savings that pruning could allow for would allow my implementation to evaluate the game at a lower depth giving more precise results.

As illustrated in the below diagram, alpha beta pruning does not expand the sections of the tree that are greyed out because it can be determined that no possible result will be selected in a subtree eliminate the need to calculate the value of that node.



Depth Variability

To take advantage of this new found efficiency allowed by Alpha Beta Pruning, I decided that I would experiment with the maximum depth allowed by my implementation. After some experimentation on the Trottier computers, I determined that with the initial board having 200+ initial move options, I could safely compute to a depth of size 3 within the two second time frame. One technique to improve this could have been to more intelligently select the order to evaluate the nodes of the tree so that I can prune more branches from the tree but at a glance I was not able to think of any suitable techniques given my mostly self imposed limited time frame. Instead, I decided that as the number of possible moves decreased with the progress of the game, I would start to increase the depth allowed to increase the accuracy of my player.

```
public static int getDepth(PentagoBoardState board) {
    int depth = 3;

    if (board.getAllLegalMoves().size() < 50) {
        depth = 6;
    }
    else if (board.getAllLegalMoves().size() < 80) {
        depth = 5;
    }
    else if (board.getAllLegalMoves().size() < 180) {
        depth = 4;
    }
    return depth;
}
```

Evaulation Function

For my evaluation function, I read various StackOverflow posts about Connect4 Heuristics thinking that it would likely be fairly similar for the sake of my game and time frame. Unfortunately, in my caffeine fueled late night I forgot to keep track of them so I can't make proper references to them. I ended up settling on counting the number of continuous streaks in rows, columns and diagonals. I summed all of these into a score that was ten to the power of the length of the line, expect in the case of the diagonal which was twenty to the power of the length because I found that my player didn't do well against players that went for diagonals.

Disjoint Sets

Another feature of my heuristic was a poor attempt at trying to count the number of disjoint sets in a row, column or diagonal as a quadrant swap could likely complete them quickly and I wanted to incetivize the creation of these. The simplest way I could think of to do this was to count the number of white or black pieces present in a row, column or diagonal and increase the score by 5 to the number of pieces. This did help the performace of my agent against quadrant-swap victories but I believe that there is much more that could be done here to refine it.

One weakness of this implementation is that is it has no concept of being blocked by a piece of the opposite colour. For example, a row that has perfectly alternative black and white pieces should not be worth much to any player, but the heurisitic valued that positively for both players and there were three of their pieces on the same row despite no opportunity to turn this configuration into a win. Refining the concept of disjoint sets would be a very clear area for improvement.

Instant Wins

In addition to counting the length of lines, my heuristic did also prioritize finding immediate wins above all else. Unfortunately, my player was entirely offensive and had no concept of defence. This proved effective against the random player and less sophisticated AI agents as my offensive player was often able to win before the other, it was not particularly effective against agents that were able to balance offence and defence well.

No knowledge of swaps

Finally, my heuristic lacked any knowledge of the board swaps and how to take advantage of them intelligently. I considered working on this but the implementation of it seemed quite complex for the time I had remaining so I was not able to execute on it. I would hypothesize that giving the evaluation function and understanding of the quadrant swaps so that I could take advantage of building two halves of the lines on separate quadrants to bring them together later would be very effective in improving it's performance.

References

[1]: https://github.com/LazoCoder/Tic-Tac-Toe "LazoCoder's AI Agent for Tic-Tac-Toe"

[2]: images/AB_pruning.svg.png "Jez9999 [CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0/)]"