Pentago Swap Write-up

Rajae Faraj

July 15, 2021

Explanation

The project implements minimax with alpha beta pruning to select the moves to play. The selection is based on an evaluation function that considers the current pieces on the board. Due to computational limitations, the depth of the algorithm is of two during the early moves but is upgraded to three during the mid-game since less possible moves are available. The evaluation function is very extensive as it covers a broad range of pieces' combinations. The function penalizes more for optimal opponent moves than it rewards the player, since playing defensively tends to reward more than playing offensively. For example, three opponent pieces on the same row reduces the evaluation total by 2100, whereas they only add 350 to it when the player follows the same sequence. My AI seems to occasionally ignore opponent wins on diagonals, thus, a higher number is subtracted from the value of a move when there are three opponent pieces on a diagonal. The agent loops through the board quadrant by quadrant, row by row, column by column, and then piece by piece. In doing so, it calculates the number of pieces in each row, column, diagonal, and semi-diagonal (i.e: two pieces on (0,1) and (1,2)). Based on the number of each element, the evaluation total is adjusted. The agent considers moves that result in four / five opponent pieces placed on diagonals, rows, and columns as deficient, since

they are guaranteed to result in a win, and thus, their occurrence hinders the value of the move further as to make it less likely to be picked. It also penalizes for trios of pieces. During the early game (i.e: turn > 5), the agent is warier of the opponent's pairs of pieces on rows / columns / diagonals, as they are more likely to turn into sequences of three pieces, which helps the opponent in obtaining a quick win. Basically, the evaluation function determines if the moves to be picked are favorable or not, and the best move among all the possible moves is then chosen. No starter moves were implemented since all the moves seemed to be able to score a win and hardcoding a sequence of moves can be disadvantageous if the opponent is cautious.

Motivation

I changed the algorithm implemented from minimax to Monte-Carlo tree search (MCTS), and vice-versa multiple times before settling on the use of minimax. Minimax seemed more foreseeable of optimal opponents' moves, whereas MCTS acted in a somewhat random way in the same context. I presume that many students will build their agent to be optimal, and thus, minimax is better fitted. Although minimax is characterized with high branching factors (upper bound of 216 in this case) which is a concern computationally wise, combining alpha beta pruning and a decent heuristic helps in lowering the time needed to visit the available moves. None beneficial branches are immediately pruned as their value estimate is weak. This increases the accuracy of some moves at the expense of others, since the evaluation function can overestimate the quality of states. Overall, since the best strategy against minimax is usually minimax, and since I believe that many students will adopt it, I settled on minimax.

Theoretical Basis

Minimax is a recursive algorithm that holds a minimizer and a maximizer player. It computes a utility value for both. During a min turn, the player is assigned

the worst score among children; the best score is assigned during max turns. Minimax expands all the nodes available in the search tree up to a certain depth or up to the end of the tree, assigns a value to each node based on a heuristic, and backtracks while allocating min and max values. In this case, alpha beta pruning was utilized in combination with minimax to reduce the number of branches visited. It does not alter the functionality of minimax, it rather prunes branches that cannot influence the optimal move to be picked in any way. However, minimax assumes that the opponent is also playing optimally, thus sub-optimal or unpredictable opponents can hinder its performance. Nonetheless, minimax is guaranteed to function until the end of the game since PentagoSwap is finite and has a set number of turns (complete), and thus, can make up for random moves throughout the game.

Pros and Cons

There are many advantages to minimax. Minimax is easy to implement. The evaluation function is also easy to upgrade (writing a very fit heuristic is however difficult). The utility function was tailored for the PentagoSwap, and thus is better in approximating the possible outcomes of moves. Another advantage of minimax is that it always leads to an optimal state against optimal opponents, which helps in winning the game and minimizes the probability of loosing.

Minimax's biggest flaw is also its biggest advantage, which is how pessimistic it is. An ideal world where the opponent always picks the best moves is not always achievable. The agent can lose to players implementing MCTS or using randomness in their strategies, as it cannot predict unlikely moves and win positions. A major problem with this implementation is the computational power. Due to the time limit, I was only able to reach depth two in the first ten moves, which hinders the early-game significantly. In fact, this AI is more likely to lose in the early-game than in the late-game. Thus, algorithms such as MCTS

or agents that use powerful starter move sequences have a higher probability to win at the start of the game. My agent is also un-biased towards diagonals, which I tried to fix in the evaluation function, but failed to do so appropriately, thus, it is more likely to lose against players with a diagonal win preference.

Other Approaches

As mentioned above, I attempted the MCTS algorithm. No tree policy was used, every move was played up to the end of the game. The evaluation function was very simple: WIN = 11010, DRAW = 0, LOSS = - 11000. Multiple rollouts were then performed, and statistics were averaged for each move. The best performing move, i.e. the one with the highest value mean, was then picked. It did not perform as well as my minimax version as it didn't have a tree policy and only branched once. Although it still won occasionally against the minimax version since minimax is sub-optimal against sub-optimal agents, the current agent won more frequently. There is an element of luck in this game, and also a bias toward starting first, thus, it is inconceivable to always win against other smart agents.

Improvements

The agent could be improved by introducing a matrix that contains all possible board state layouts that qualify for a win, and by storing the current board state in an array. Then to save time, arrays from the matrix can be compared to the board state formation to determine whether a move is valuable or not. Implementing a better evaluation function is also needed. Accounting for draws in the evaluation function would also be helpful. I attempted to do the former, but the agent would sometimes pick a draw when it was able to win, so I removed it. A better heuristic allows for a depth of three from the start of the game which is very advantageous. A better heuristic will also allow to suppress the diagonal un-bias, and to improve the mid-game. Forming a heuristic based

on deep learning would be ideal as it leads to a high probability of winning because it is able to learn from previous played games the best strategies to score a win. Adding some randomness to the agent could also prove valuable against optimal opponents, as it holds a disconcert potential. Also, my agent almost always lost to the 2 1 2 (x2 pieces in 2 quadrants, 1 piece in a different quadrant) diagonal formation as it had a hard time predicting it, thus, the evaluation function must be further adjusted to account for this approach.