COMP 424 Final Project Report

Kaustav Das Sharma, 26077292
April 13th, 2021

I. Approach and Motivation

THE motivation for the agent was based on other simple, classic AI programs of similar characteristic environments. These environments generally consist of zero-sum games where the gained utility of one player or group results in the lost utility of another player or group [1]. This immediately strikes as a problem for the Minimax algorithm to solve. The Minimax algorithm is a dynamic backtracking algorithm that is used in-game decision making to find the optimal move for a player. The major assumptions here are that the opponent also plays optimally and that both sides have an equivalent perception of utility. Other games which are well-suited towards Minimax algorithms are player turn-based games such as Tic-Tac-Toe, Chess, etc.

Therefore, the agent is based on the Minimax algorithm with the exception that it follows a logical opening strategy conjured through expertise built playing the game. Furthermore, many different heuristics/evaluation functions were implemented to get an optimal utility function, and after rigorous testing, the agent uses a heuristic based on the calculation of the number of adjacent common pieces. Additionally, alpha-beta pruning was added to ensure that paths of the Minimax search that will not be traversed are not expanded upon. Since there is a limit to the computational and memory usage, the depth was trialled at different levels to find the maximum depth which conforms to the usage limits. At depth 2, which means that the tree search evaluates one maximal move and one minimal move, all moves were well under the 2-second limit averaging around 0.5 seconds per move. At depth 3, which means that the tree search evaluates two maximal moves and one minimal move, the Minimax search was often not able to compute all possible states under the allotted time. Therefore, a depth of 2 is used until a point at which computing 3 layers deep is manageable (determined experimentally), since the number of possible moves decreases exponentially after each move is played, and switched to 3 thereafter. Conclusively, each move consists of an evaluation of game state (opening or not) and depending on the state of the game, the agent chooses to follow the opening strategy or Minimax with the aforementioned heuristic and depth respectively. The opening strategy is based on occupying the centre tile of each quadrant as this allows for the greatest number of attacks (getting chains of 5 tiles).

II. Theoretical Basis

The theoretical basis of the agent is best evaluated by an overview of the alternate algorithms that were not chosen. In the context of this class, that would be Monte Carlo Tree Search (MCTS). Monte Carlo leverages the power of randomness in decision making. MCTS, just like Minimax, also builds up a game tree and does computations on it to find the path of the highest utility. However, the key difference is that it does not need an evaluation function to find the utility. Instead, MCTS simulates random games from a given state/position to find the success rate. In probability theory, the law of large numbers states that as the number of identically distributed, randomly generated variables increases, their sample mean (average) approaches their theoretical mean [2]. According to the law, the average of the results obtained from a large number of trials should be close to the expected value. Thus, in the absence of a concrete evaluation function MCTS aids in decision making.

In the case of Pentago Twist, the objective of the game is fairly straightforward. It is like many games under the Moku family (Connect4, Tic Tac Toe) in which the goal is to connect, horizontally, vertically or diagonally, a sequence of pieces belonging to a player before the opponent. Thus, the game presents two challenges. The first is to connect as many pieces as possible, and the second is to block the opponent from doing the same. Therefore, given these challenges, it is clear that the evaluation function should target stringing pieces together. This is why the heuristic of counting adjacent pieces was used. If the agent evaluates positions by the maximum number of adjacent pieces, then the probability of finding a winning move increases, thus making the heuristic optimal.

Thus, given that the evaluation function is optimal and logically coherent, Minimax seems to be the better algorithm to use. However, there are computational reasons why Minimax is also preferable. For Minimax, the time complexity is $O(b^m)$,where b is the branching factor of the game-tree, and m is the maximum depth of the tree. The space complexity is $O(bm)$ [3]. The runtime of the MCTS is $O(mkI/c)$ where m is the number of random children to consider per search and k is the number of parallel searches, and I is the number of iterations and C is the number of cores available (since C is 1, the complexity would be $O(mkI)$. The memory complexity is $O(mk)$ since in each iteration mk states are evaluated [4]. It is difficult to compare

these complexities since they involve different parameters, however, making a random move takes $5 * 10^{-5}$ seconds (calculated experimentally on a 2.6 GHz Dual-Core Intel Core i5). Since the total allotted time per move is 2 seconds, one simulation can run 40000 random moves. The total possible moves for a game of Pentago Twist is $8^{36} * 36!$, but most games don't use up all possible moves. Instead, a game consisting of random moves takes about 12 moves (also calculated experimentally), giving about $9.88 * 10^{29}$ moves. This means that a simulation of a game would only encompass $1/(4.04 * 10^{26})$ of possible moves. The same constrictions apply towards the Minimax algorithm, hence why it is depth limited, however since Monte Carlo relies solely on random simulation, the number of simulated games/moves needs to be far beyond that of 2 layers of Minimax search (extended to 3 at move 10), since the evaluation function provides meaning which the same level of random games can not.

Lastly, since Minimax equates to playing random moves at the beginning of the game (at least with the agent's current heuristic function), there is an opening strategy built-in. The agent attempts to place tiles in the centre of each quadrant. The strategy is that if the central tiles of each quadrant are occupied, there are the most possible attacking opportunities. A key factor behind Moku games is building multiple chains such that an agent checking for a winning move one layer deep cannot prevent a win since there are multiple winning moves. Using this strategy in tandem with the heuristic function allows for the greatest winning opportunity for the agent.

III. Advantages and Disadvantages

The key advantage of the agent and the Minimax approach is that it is complete, optimal (if both players play optimally) and delivers good results given the time and memory constraints compared to other algorithms. The heuristic constantly evaluates and plays moves that either increase the length of existing chains, creates new chains or blocks the opposing player from doing the same.

The disadvantages are that the completeness and optimality only extend to the depth of the tree traversal. This is quite significant as the Minimax search is highly dependent on the number of layers it can traverse. An example from Chess would be that if the Minimax evaluates two layers deep, then sacrificing or even trading a piece would never be considered the best move as the utility is not evaluated beyond a move ahead. However, sacrificing and trading are commonplace in chess, and so the depth hurts the optimality and completeness of the entire game. There are also issues with the heuristic function. Since it doesn't evaluate the length of chains, just pieces adjacent to other pieces, the agent will often play moves next to other pieces which are permanently blocked off and can never reach the goal of 5 in a row. This is damaging as it is a wasted move and one which at best does not increase the utility of the player.

IV. Attempted Approaches

The first evaluation function that I attempted for the Minimax was based on whether the player is in a winning position or not. Although this can beat a random player most of the time (85%), the agent is still playing random moves most of the time since there are very few circumstances in which a state is reached at which a winning move is one move away. Thus, the current agent was able beat this agent 100

Another possible opening strategy is placing the tiles at the corners of the game board. This allows for potential chain sizes to be longer since the board is 6x6. However, rotations easily defeat this strategy and therefore the newer opening strategy is superior. With the new opening strategy, since the tile is at the middle of the quadrant, any rotation does not affect the position of the piece.

Another strategy was to set a depth of 3 for the entirety of the Minimax search and return a random move if the time is about to expire. Unfortunately, the first 10 moves mostly take beyond the allotted time and there is far too much utility lost by playing 6 random moves at the start of the game.

As mentioned in the previous section, a heuristic that accounts for the length of the chains is important. When this heuristic function was implemented, it performed as it was supposed to, making the longest chains possible to win the game. However, somehow, it is ignorant of good moves that the opposition player plays. Playing the new heuristic against this old one, resulted in a 100

V. Improvements

The most promising improvement for the current agent would be to somehow combine multiple heuristic functions according to the state of the game. From testing, it seems as though the current heuristic function performs well in the middle game whereas the chain heuristic performs better during the end games. Thus, with a more expert understanding of what constitutes a middle or an end game, the agent should switch heuristic functions to maximise performance. Furthermore, since there are going to be discrepancies between the utilities for these functions, the agent could play different versions of the agent with different weights (utility values) to determine the optimal weights.

Another possible improvement is to accurately assess the depth of the Minimax search based on the complexity of the current state. Since a greater depth will always yield better moves, the agent should assess the complexity of a given state and determine how much time it would take to perform Minimax and choose the depth of the search accordingly.

One last potential improvement, to cut down on the branching factor, would be to only apply Minimax to moves (children) that have neighbouring pieces of the same colour.

VI. References

[1]: https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/zero.html [2]:

https://www.britannica.com/science/law-of-large-numbers
[3]: https://www.javatpoint.com/mini-max-algorithm-in-ai
[4]: http://stanford.edu/~rezab/classes/cme323/S15/projects/monte_carlo_search_tree_report.pdf