COMP424 Final Project Description

Jenny Cao
School of Computer Science
McGill University
McGill id: 260917694

***Abstract*—This report is a description of my approaches and solution to the game Pentago-Twist. Throughout the course of exploration, I mostly focused on the alpha-beta pruning algorithm and the Monte-Carlo tree search algorithm, combined with a series of pruning techniques. Alpha-beta pruning turned out to have a much better win rate under the current implementation, thus selected at the end. However, many ideas about future improvements are explored and discussed here.**

Keywords—decision-making, Pentago-Twist, Alpha-beta Pruning, Monte-Carlo tree search

I. PROGRAM EXPLANATION

Pentago-Twist is a two-player game on a 6x6 board, which consists of four 3x3 quadrants. The first player will play as white, and the second player will play as black. In order to win, players need to have 5 pieces of their color aligned. The current player can choose an empty space for each move to place the piece and choose to flip or rotate a quadrant. Here I present my ideas for an Intelligent program to play Pentago-Twist.

My program's current working solution uses the alpha-beta pruning algorithm as the core deciding scheme, since it evaluates and finds the optimal move based on utilities of terminal states. Since the rules and setup of Pentago-Twist are clearly given to us, I borrowed the code from the function getAllLegalMoves() and implemented my function to generate all possible moves but prunes the ones that lead to the same result. Under the given decision time, the alpha-beta pruning algorithm could not run through every possible branch. Therefore, I used a transposition table to store states encountered and designed utility functions to help evaluate the states more likely to win.

Some strategic decisions were implemented for the first few moves. According to the game setup, it is more optimal if we place our moves in each quadrant's center. Center pieces are more likely to be involved in a winning alignment and are less likely to be influenced by the opponents' interruption. Therefore, we would check whether the center of each quadrant had been occupied. If not, then an empty central cell will be our next move.

Although our first few moves could be easily determined, we will still run the alpha-beta search algorithm and use the decision time for the first few moves to explore more states. The more states we have encountered, the more data we have in our transposition table. The first 30s will be used as much as possible, which means we will explore one step deeper than normal move for the first move we play. Since our first move would definitely be occupying a quadrant's center, we will presume that we already made this move, and the opponent plays a random move. This helps us to explore deeper states, hence making better use of our transposition table.

One other popular adversarial search algorithm that I considered and implemented was Monte-Carlo Tree Search algorithm, which combines tree search expansion with random simulation results. Many may choose this algorithm since it precisely captures the termination time of the program, which is very important considering our time constraint. However, I choose alpha-beta pruning as the final algorithm for submission because of the significant difference in win rate between current implementations of these two algorithms. I will also discuss some ideas that could help improve these two algorithms.

II. THEORETICAL FOUNDATIONS

A. Alpha-beta Pruning Algorithm

The implementation of alpha-beta Pruning mostly referenced the textbook [1].

In chooseMove() within the StudenPlayer class, we create an alpha-beta Search instance and uses it to call the alpha-beta pruning function. Using the current Pentago board state, we can extract information such as current player, turn numbers, and etc. These information help us to set the max and min player, which refer to our agent and our opponent here. In the alpha-beta search function, we start from the moves our agent can make and simulate each possible move our opponent may make by calling the function min_value(), then call max_value() to simulate our reactions, and vice versa. While searching, we also keep track of our depth at each level. Since we have a limited time constraint, we would only explore certain levels of the tree and use an evaluation function to decide which state is more worthy.

The Minimax algorithm is similar to our previous explanation of recursively evaluating opponent's decision and establishing a decision tree. Comparing to Minimax algorithm, alpha-beta pruning uses alpha and beta to indicate if we can prune a certain branch. Alpha stores the current maximum value, and beta stores the current minimum value. In max_value() function, if current value is greater or equal to beta, it will not be chosen; in min_value() function, if the current value is less or equal to alpha, it will not be chosen as well. This means that we only need to examine only $O(b^{m/2})$ nodes to pick the best move, instead of $O(b^m)$ for Minimax. Moreover, we would prune some moves with the same result before starting the searching, further limiting our chances.

One key strategy was adding a Boolean variable to indicate if this is the first move we will make, since we have 30s to plan and prepare. Using this information, we provide different strategies for the first few moves, which were explained in the previous part.

B. *Pruning before Searching*

Here, we borrow the idea of forward pruning discussed in the lecture. Before we begin searching, we need to first get a hold of all possible moves that could be done to the current board. Although we were given the function getAllLegalMoves() from class PentagoBoardState, some moves may lead to the same result. Therefore, getting rid of the moves that produce the same results could help us limit our state space significantly.

In class MyTool, I implemented the function pruneSymmetricMove() that produces legal moves while prunes some that lead to the same state. PruneSymmetricMove() borrows from the given function getAllLegalMoves() to get all empty cells. After getting an empty cell, we clone the current PentagoBoardState and mark the empty cell with current color. We then calls the helper method isSymmetric(), which checks whether flipping or rotating a quadrant would make a difference. If not, then this flip or rotation will not be counted as a possible move.

C. *Transpostion Table*

During our search, same states may be discovered more than once. Using a transposition table, we could store reusable information thus making the program more efficient.

A transposition table usually takes the form of a hash table, as is the case in our current implementation. The key to of the hash could take various forms, and some explorations include the Zobrist Hashing [2] will be discussed later on. In the current implementation, we simply use deepHashCode() function to simplify the hashing process.

Two classes were used for generating the transposition table. The class TranspositionData employs the Singleton design pattern to instantiate the transposition table, which means only one instance of the transposition table will be created within one game. Each time we evaluate a Pentago board state, we will check if this state and its utility are already stored within the transposition table. If not, we will store it with the key being the result from calling deepHashCode() function and value being TranspositionState object containing the Pentago board state and its utility. The next time we encounter this state, we can directly use the result we have calculated before.

D. *Utility Functions*

Since the tree grows exponentially, reaching the end requires too much space and time. Therefore, an efficient utility function is crucial to making a wise decision.

The Utility() function evaluates whether we have a winner or a draw. If we have neither, then we start to evaluate by calling the function InLineScore(), which provides current board's score.

If the state leads to a win for our agent, it will give the max value possible, minimum value if opponent wins, and 0 if draw. The InLineScore() function calculates how many agent pieces and opponent pieces are within each row/column/diagonal, which grows exponentially. The greater the number is, the more likely that player will get an alignment for the row/column/diagonal. We also keep track of how many are aligned within each row/column/diagonal which also grows exponentially. This number also indicates how close the player is about to win. The overall score will be the sum of agent scores minus the opponent scores.

Here we need to discuss two important strategies for our utility function. First, the exponential growth of scores when calculating the alignment score. For example, 1 white piece that doesn't connect any other white piece has a score of 10^0. If 2 white pieces are aligned, they have the score (10^1+10^0); If 3white pieces are aligned, they have the score (10^2+10^1+10^0). This helps us to pay more attention to states that are almost winning/losing for our agent. Second, we put more weights on blocking the opponent, since putting the same weight on opponent and our agent sometimes does nothing and leads to the opponent winning if it plays first.

III. Result and analysis

A. Result

Currently, my alpha-beta pruning algorithm is able to beat the random player in most cases. (Fig.1) However, exceeding time limit is still a huge problem for alpha-beta pruning. In my current program, I check if we are close to exceeding the time limit. If we have less than 50-100 ms, our program will return what we have found currently even though it might not be optimal.

TABLE I. ALPHA-BETA PRUNING RESULTS

My player	Wins	Total rounds	Average timeouts
white	926	1000	2.94
black	853	1000	2.51

Fig. 1. Result of alpha-beta pruning against the random player. Table includes winning rate, total rounds played, and average number of moves in each round that exceeds time limit and gets a time-out. The time-limits here are to break if we have less than 20-50s, thus leading to different outcomes. The more time we have before timeout, the less likely we are to encounter a timeout. To guarantee less timeouts, the submitted code offers a more lenient bound, thus is also more likely to be not optimal.

Evidently, the first player gains a bigger advantage, but it also leads to large state space to be explored, hence higher chances for getting timeouts.

B. Advantages vs. Disadvantages

In general, alpha-beta pruning search guarantees to return the optimal state. However, in the current implementation, a time constraint was set up to avoid automatic loss of the game. This leads to uncertainty of the outcome, since the optimal move might not be explored before we forcefully break the loop. With a huge branching factor in Pentago-Twist, our search depth is very limited.

Nevertheless, the performance of normal MCTS still falls behind alpha-beta pruning search, for 2 seconds would not guarantee many useful data to be provided. Due to the help of forward pruning, transposition table, and utility functions, these disadvantages will not be evident when playing against a random player since our move is still backed up by many useful data.

IV. Future Improvements

A. Pruning Possible States

The current pruning function for possible moves only checks whether flip or rotate produces the same outcome. However, this does not check for moves that place the pieces differently but ends up with the same state after flipping or rotating. This could help us to further eliminate moves that would produce the same result.

B. Transposition Table

The current transposition table only uses the piece matrix and calls the deepHashCode() function within the Array class. This only guarantees the same state would lead to the same hash value but doesn't guarantee different states would all have distinctive hash values. Therefore, if we can improve on the distinctiveness of the key-PentagoState mapping, we could have a more accurate transposition table.

For example, we could convert the matrix into distinctive strings as keys, with each character representing a cell. However, the performance of String keys is significantly worse than that of int keys, especially with a 36-character long String. [3]

Another idea could be to use a 36-digit octal number to indicate board and each digit representing a cell. Furthermore, we could also use a 4*6-digit octal number to represent the black/white pieces in each row/column. Even though this is an improvement on the idea of String keys, it is still not as efficient and may have unexpected behavior due to internal math operations on binary circuits and integer overflow.

Further ideas to improve the transposition table could be borrowed from the Zobrist Hashing [2], which is to generate a distinct hashing value using XOR operations on random numbers.

C. Utility Functions

We could use machine learning to gather more data and evaluate the win rate based on the current board information. This could play a part in our utility function, making the evaluation function more accurate.

V. OTHER APPROACHES

A. Monte-Carlo Tree Search

A working example of a normal MCTS is implemented within the student_Player package, which referenced [4]. The java classes Node, State, UCT, and MCTS are all classes used for MCTS. MCTS includes four steps: selection, expansion, simulation, and backpropogation. In selection phase, we use the tree policy to choose the most promising child of each node until a leaf is reached. In the expansion phase, we create expand to the child nodes if the current node doesn't end the game. In the simulation phase, we randomly choose the moves from current node until the game ends. In the backpropogation phase, we update the current node paths with our simulation results.

Here we will also discuss further improvements on MCTS. The current normal implementation of MCTS with forward pruning doesn't perform as well as the current alpha-beta pruning algorithm. Its win rate against a random player is not significant. This may be because our dataset is limited and the rounds are not enough. Nevertheless, based on knowledge about Pentago-Twist, we can combine MCTS with transposition table and improve its performance.

The Naïve way to implement this is using transposition table and persisting the cumulative reward value of a state, which helps the exploitation part of the algorithm to make better choices. However, it skews the exploitation part of the algorithm in a potentially harmful way, since the child node may be explored more often than the parent node which should be technically impossible.

Complicated implementations of transposition tables and moving groups[5] proved to be useful, but will result in a much more complicated program.

B. Neuro Networks and Machine Learning

Some more advanced approaches could be to use Monte Carlo tree search algorithm to find its moves based on knowledge previously acquired by machine learning, which shares certain similarities with Monte-Carlo Tree Search combining with a transposition table. The state-of-the-art chess-playing AI Alpha-Go also employs neuro networks to find each move and predicts their outcomes.[6]

REFERENCES

[1] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Hoboken: Pearson, 2021, pp. 167-171

[2] L. Wächter, "Zobrist Hashing," *Medium*, 04-Aug-2020. [Online]. Available: https://levelup.gitconnected.com/zobrist-hashing-305c6c3c54d0. [Accessed: 13-Apr-2021].

[3] J. Dunstan, "String Keys vs. Int Keys," *JacksonDunstancom RSS*. [Online]. Available: https://www.jacksondunstan.com/articles/2527. [Accessed: 13-Apr-2021].

[4] *Monte Carlo Tree Search - About*. [Online]. Available: https://web.archive.org/web/20160307033754/http://mcts.ai/about/index.html. [Accessed: 13-Apr-2021].

[5] B. E. Childs, J. H. Brodeur, and L. Kocsis, "Transpositions and move groups in Monte Carlo tree search," *2008 IEEE Symposium On Computational Intelligence and Games*, 2008.

[6] "AlphaGo: The story so far," *Deepmind*. [Online]. Available: https://deepmind.com/research/case-studies/alphago-the-story-so-far. [Accessed: 13-Apr-2021].