COMP424 - Final Project: Pentago Twist

Frederic Beaupre, 260807622

April 13, 2021

1 How the Project Works and Motivation for my Approach

1.1 Overview

In completing this project, many agents were considered prior to settling on one for the final submission. I will discuss all these agents and different approaches in detail in section 4. For this section, I will be brief and say that I started with a simple alpha-beta pruning algorithm for the search. Due to its high computational cost and lack of speed, I tried my hand at the negamax algorithm with alpha-beta pruning. Still unsatisfied with the level of play of the agent, I implemented a sort of hybrid agent, which uses Monte Carlo simulations to filter through all the legal moves from a given board state and assign UCT values to them. At simulation's end we are left with a HashMap of moves and UCT values as (Key, Value) pairs. The algorithm then sorts this HashMap in order of increasing UCT value, and a top-k sampling function then crops down this list to the best k, where I used k = 50. This list of size k is the set of legal moves I use as game tree for the alpha-beta pruning algorithm. The latter is run with an initial depth limit of 2 which is incremented every 10 moves. As you can see in Table 1, adding the Monte Carlo simulations to Negamax and alpha-beta considerably increased both their performance. This was cause for investigating whether or not a simple Monte Carlo Tree search algorithm on its own would perform better than the previous algorithms. But as you can see again in the table below, this was not the case, so my submission presents our agent as the MiniMax algorithm with Alpha-Beta Pruning and Monte Carlo simulations, a hybrid, initialized with search depth = 2. Throughout this report I refer to minimax with alpha-pruning as simply alpha-beta, and to negamax with alpha-beta pruning as simply negamax.

	Depth	
Search Algorithm	2	3
	Win Percentage	
Negamax with MC Simulations	**82.5**	**90**
Negamax without MC Simulations	64.75	62.5
Alpha-Beta with MC Simulations	**87.75**	Timeout
Alpha-Beta without MC Simulations	77	Timeout
Monte Carlo Tree Search	57.83	

Table 1: Search Algorithms' win percentage against a random player. Note that while a varying number of games were played for each algorithm, a minimum of 200 games were played played for all of them.

I used this first round of tests to narrow down my options for best search algorithm to 1) Negamax with Monte Carlo Simulations, depth = 2; 2) Negamax with Monte Carlo Simulations, depth = 3; and 3) Alpha-Beta with Monte Carlo Simulations, depth = 2. I then tested those three agents on 700 games against the random player.

Search Algorithm	Negamax w/ MC		Alpha-Beta w/ MC
Depth	2	3	2
Win Percentage	81.1	86.0	**90.5**

Table 2: Search Algorithms Championship Round (700 games played)

We see that with the same hyper-parameters (depth limit and time allocated to MC simulations and alpha-beta search) alpha-beta pruning performs considerably better than negamax with alpha-beta pruning. It is also stronger than negamax even when negamax has a larger depth limit for its search (3 vs. 2). Thus the agent we submit for the class tournament is the alpha-beta with MC simulations hybrid, with a search depth of 2.

1.2 Evaluation Function

The evaluation function used was the same for negamax and alpha-beta. It checks for occurrences of marbles that are placed three, four, or five-in-a-row along the verticals, horizontals and diagonals of the given board state for a given player. Each occurrence of such a streak is then multiplied by its corresponding, pre-defined weight. The evaluation function also counts the number of centre marbles for a given player. By centre marble I mean any marble that is not set along the edges of the board.

However small the weight assigned to centre marbles is, I implemented this "centre play" logic onto my evaluation function because I believe that centre marbles are stronger than edge marbles, as they can combine with other marbles to create k-in-a rows in many more ways. This is discussed further in subsection 1.3. The evaluation function is summed up below, and was inspired by the evaluation function described in [1].

Number of occurrences x_i	Weight w_i
3 Marbles in-a-row	100
4 Marbles in-a-row	1000
5 Marbles in-a-row	100 000
Centre Marble	5

Table 3: Weights for a given occurence in the evaluation function. Note that 5 marbles in-a-row is basically a win, yet we still check if there are multiple such occurrences, as a position with multiple possible wins is stronger than a position with a single possible win.

$$Evaluation(board) = \sum x_i \cdot w_i \tag{1}$$

I also considered other possible occurrences to add in our evaluation function, but as they ended up hurting overall performance, I removed them from evaluation and left them only to be mentioned in section 4, under "Endgame Tactics".

1.3 Opening Moves

I believe that in the opening moves of a Pentago Twist game, a basic and solid strategy (regardless of what color the pieces you play with) is to try to control the centre, either by expanding towards it or by preventing the opponent from doing the same. Thus, the centre marble weight in my evaluation function will guide the agent towards centre play until 3 marbles in-a-row start to appear on the board, for either side. At that point, since k in-a-rows have far larger weight than centre

marbles, the agent will try to build on his k in-a-rows or prevent the opponent from expanding his own.

More specific to the very first moves, I believe that the centre squares of each quadrant are the strongest squares at the beginning of every game, because they are squares that work in favour of the "centre play" strategy, and because they are the only squares on the board that are never affected by a flip or a twist. They thus can be referred to as positional anchors: the player who controls them has, in my amateur's opinion, an early positional advantage.

Given those strategic guidelines, I implemented a function which guides the first five moves played by the agent. The function works in the following way. Note that below I identify the first move as move 1, not as move 0.

For Agent Playing White:

- **Moves 1-2**: Look for positional anchors and occupy the first available one. At least one is available for certain.
- **Move 3**: Find a positional anchor which is occupied by one of our pieces (there exists one for certain). Expand towards the centre by creating two marbles in-a-row with one of the anchors we occupy.
- **Moves 4-5**: Verify that the opponent does not have a 3-in-a-row on the board. If he does, find the best move using our search algorithm. If not, loop through the positional anchors. If the current anchor is not occupied, occupy it. If it is occupied by one of our pieces, expand towards the centre as in move 3. If it is occupied by the opponent, check the next anchor.

The logic is the same for black with the only difference being that we start to look out for the opponent's 3-in-a-rows one step earlier, at move 3. This is because I felt that otherwise, my agent could be caught "asleep at the wheel", looking for anchors and centre expansion while giving white 4 "free" moves along the edges of the board, where he could easily build a four-in-a-row and already be close to winning. This is not a problem when playing as white because we then have the extra tempo, the luxury of having the first move.

2 Theoretical Basis

2.1 Pentago Twist

Pentago Twist is a zero-sum, 2-player, turn-taking, deterministic game with perfect information, which forms our basis for formulating it as a search problem. While not as large as that of the game Go, it has a very large branching factor. The original game Pentago, which is slightly different than Pentago Twist (change the flip move for a counter-clockwise rotation and the two games are the same), has 288 possible first moves and an average branching factor of 97.3 over all states [2]. Those values are very similar, if not the same, for Pentago Twist.

2.2 Alpha-Beta Pruning

As per theory, we know that alpha-beta pruning is only optimal when playing against optimal opponents who are playing according to the same evaluation function as our agent, and only when search proceeds to the end of game [3]. There may be stronger strategies for sub-optimal opponents, and for games with large branching factors such as Pentago Twist, the search depth is too limited. It is also very expensive, even with the pruning. It is for those reasons that I went for a hybrid, hoping that it would make my agent 1) faster, 2) better at handling those sub-optimal opponents and those using a very different evaluation function, and 3) that it would alleviate the weight of the problem surrounding our search not reaching the end of games.

2.3 Negamax (with alpha-beta pruning)

Negamax is very similar to alpha-beta pruning. It relies on the fact that $\max(\alpha, \beta) = -\min(-\alpha, -\beta)$ to simplify the implementation of the minimax algorithm that is more commonly used with alpha-beta pruning [4]. The player on move looks for a move that maximizes the negation of the value resulting from the move. I hoped that the one fewer procedure would speed up my agent. This speed up was barely noticeable and the change in algorithm even caused a small dip in performance, as you can see in Table 1, so I quickly took Negamax out of consideration for the project.

2.4 Monte Carlo Tree Search

On the surface, Monte Carlo Tree Search seemed destined to solve most of the problems of an agent entirely based around alpha-beta pruning. MCTS is faster, unaffected by the branching factor and does not require careful design of an evaluation function [3]. However, it may miss optimal play because it won't see all moves at deeper nodes, and it trades relying on an evaluation function for relying on the policy used to generate candidate states. In the end, when testing my implementation of MCTS, I found that it performed far worse[1] than expected (see Table 1). I thus came to the conclusion that my best bet was to aim for a good trade-off between alpha-beta and MCTS, which resulted in the hybrid agent presented in this project.

3 Advantages/Disadvantages, Expected Failure Modes

3.1 Advantages

The main advantage of my agent lies in its versatility. Combining alpha-beta pruning with Monte Carlo simulations lets our agent draw strengths from both approaches while diminishing the extent of their shortcomings when implemented in a non-hybrid way. For example, while the alpha-beta part of the algorithm does not always reach end of games in its search, the Monte Carlo simulations that ran before explore further, so that knowledge that may be missed by alpha-beta at deeper levels will perhaps be explored by our MC simulations.
Also, while the agent cannot search as deep as a stand-alone MCTS agent, since it has an "alpha-beta side", it can search to greater extent along any given depth, thus giving it a better shot at

[1] It must be noted that my implementation of MCTS was quite basic due to time constraints, and that it could be improved significantly, although perhaps not to the level of the other hybrid agents.

optimality. Moreoever, thanks to the opening moves described in subsection 1.3, I believe my agent benefits from an early-game positional advantage against most agents that launch their search algorithms from move 1, as well as an awareness of the importance of centre play given by the weight of centre marbles in the evaluation function.

3.2 Disadvantages

It is very hard to find a good evaluation function for Pentago Twist. While my own works well against a random player, it is still unclear how it would compare to stronger agents' evaluation functions as I mostly tested it against a random player or the other agents I designed, which as we have seen, are weaker. If one were to stick with an alpha-beta pruning approach, it would require more extensive knowledge of the game, especially of its tactics in the middle game and endgame, to come up with a stronger evaluation function.
Also, while the MC simulations did speed up my agent, those simulations still borrowed some of the time (800ms) that the alpha-beta pruning computation would have had for itself if implemented alone. Hence, while faster than a stand-alone alpha-beta agent, my hybrid agent will still probably not be among the fastest in the class tournament.
Finally, and as briefly mentioned in the advantages section, my agent cannot search as deep as a stand-alone MCTS can.

3.3 Expected Failure Modes

First of all, within the constraints of the class tournament, my search algorithm is in danger of disqualification when setting the search depth at 3, and is looking at certain disqualification for search depths that are beyond that value, because it is too slow.
Also, while it is not exactly an "expected failure mode", it is interesting to note that during the test on 700 games against the random player (Table 2), my agent (Alpha-Beta with Monte Carlo simulations) ended up with 348.5 wins (the .5 comes from a draw) out of 350 games as black which is a 99.6% win rate, and only 285 wins out of 350 as white, a 81.4% win rate. I have not had time to investigate this large discrepancy in strength of play that seems to depend on the colour of the pieces that the agent is playing with. One possibility is that that small difference in the hardcoded opening moves between the two colours has an effect, but it is extremely unlikely that this effect accounts for the totality of the 18% difference.
So as I said, it is not a failure mode, but my agent is significantly weaker when playing the white pieces.

4 Other Approaches Explored

Most of the approaches below have already been discussed in other sections (e.g., negamax, MCTS). To avoid repetition in this section, I will therefore try to only go over new points, or to further those that were briefly mentioned.

4.1 Alpha-Beta Pruning and Negamax

I investigated standard alpha-beta pruning as well as alpha-beta pruning combined with Monte Carlo simulations to assign states their UCT values (UCT function described in section 4.2) and top-k sample the most promising ones before the alpha-beta search. Adding the Monte Carlo simulations greatly improved performance. I also considered depth limits of 1, 2, and 3 for the search. A depth of 1 resulted in unsatisfactory performance and a depth of 3 resulted in an agent that was far too slow. I thus set the depth limit at 2, and incremented that value every 10 moves, because the branching factor gets smaller as the number of moves played increases. I investigated the same approaches for the Negamax algorithm.

4.2 Monte Carlo Tree Search

As was briefly mentioned in a footnode, due to time constraints I was not able to fine-tune my MCTS implementation much. To quickly have something up-and-running with which to compare other agents, I went for a straightforward, pseudo-code like implementation with upper confidence trees using the following formula for the UCT value of a state:

$$\frac{\text{win score}}{\text{number of times visited}} + \sqrt{2 \times \frac{\log(\text{number of simulations})}{\text{number of visits}}} \tag{2}$$

Where I set the score attributed to a win as 10.

4.3 Endgame Tactics

The strategy guide to Pentago Twist [5] offers winning setups that are 2-3 moves away from winning if the opponent does not see what's coming, and fortunately for he who knows about these setups, they are not easy to see coming. They are called Monica's Five, The Middle Five, The Straight Five, and The Triple Power Play. In an attempt to strengthen the evaluation function, I had the latter look for those setups that are close to winning and assigned them huge weights so as to stay away from those board positions with such winning setups. However, when testing my agent with this enhanced evaluation function, not only did it make the agent slower, it also made its winning percentage drop. Perhaps that was due to an inadequate understanding of the winning setups, or to too large a weight being associated with them. In any case, I elected not to include this "endgame tactic" behaviour in my agent.

5 Potential Improvements

A strong improvement to my agent would be to implement a transposition table into the alpha-beta pruning algorithm that maps board states to their corresponding evaluation function values. With that table stored in memory, whenever the agent encounters a position on the board, it could look for that position in the transposition table and quickly retrieve its evaluation if it is indeed in the table (if not it would just perform the standard search algorithm to find the best move). That would save a lot of computation, and would allow our agent to "learn" a ton from each game.
Moreover, and as was briefly mentioned before, a stronger understanding of the game would allow us to design a better evaluation function and to properly incorporate into it the strategies and

tactics that work best at different times of the game (opening, middle game, endgame).
Finally there are a few hyper-parameters in my algorithm that could use a little bit of fine-tuning. For example, we could investigate the optimal value of k in top-k sampling, the best values for weights in the linear evaluation function, or see what MC-simulation time limit and move-time limit combination works best. Finally, as suggested in paper [2], a technique commonly used for solving chess positions, known as Retrograde Analysis, would be a promising avenue to look into for solving Pentago.

References

[1] On Solving Pentago, *Niklas Buscher*
https://www.ke.tu-darmstadt.de/lehre/arbeiten/bachelor/2011/Buescher_Niklas.pdf

[2] Pentago is a First Player Win: Strongly Solving a Game Using Parallel In-Core Retrograde Analysis, *Geoffrey Irving, OtherLab*
https://arxiv.org/pdf/1404.0743.pdf

[3] Artificial Intelligence: A modern approach,
Stuart Russel and Peter Norvig
Third Edition

[4] Negamax Algorithm,
https://en.wikipedia.org/wiki/Negamax

[5] Pentago Twist: Basic Play, Rules and Strategy Guide, *MindTwisterUSA*
https://webdav.info.ucl.ac.be/webdav/ingi2261/ProblemSet3/PentagoRulesStrategy.pdf