COMP 424: Project Report

Pentago-Twist

Kua Chen 260856888

1. Explanation and motivation

Evaluation strategy

An evaluation function plays an important role for our AI agent. An evaluation function *v(s)* estimates the expected utility of a state.[1] It should assign a value (so-called utility) to a game state so that the AI agent can choose the action that is the most promising. The first step of the design process is developing an evaluation function. Since the ultimate objective of Pentago-Twist is to have 5 pieces in a row before the opponent does, intuitively, the more our pieces in a row, the closer we are to a victory. At the same time, the more opponent's pieces in a row, the closer we are to a defeat. Therefore, my evaluation strategy is based on the how many groups of pieces in a row. Technically, it is called features of the state. First of all, if this action leads us to an immediate victory, the evaluation function will assign *Integer.MAX_VALUE* to it. On the other hand, if this action leads us to an immediate defeat, the evaluation function will assign *Integer.MIN_VALUE* to it. If immediate victory or defeat does not happen, the evaluation function will count the number of groups of 2 pieces in a row, count the number of groups of 3 pieces in a row, count the number of groups of 4 pieces in a row. These features of the game state are independent, and a weighted linear function is used (i.e. multiply their scaling factors respectively and them sum up). 4-pieces in a row is more promising than 3-pieces in a row. 3-pieces in a row is more promising to 2-pieces in a row. Therefore, we defined scaling factor for 4-pieces to be 30, scaling factor for 3-pieces to be 20, scaling factor for 2 pieces be 5.

There is an additional policy: if a piece is in the center of a quadrant, it will gain extra 2 utility score. This policy is usually aimed for the first turn. Due to personal experience, I noticed that placing the piece in the center of a quadrant at first turn could increase the possibility of winning. With this policy, the AI agent will choose to place the piece in a center. As mentioned above, the more opponent's pieces are in a row, the closer we are to a defeat. Thus, our evaluation function should also take opponent's pieces into account. For each group of 3 opponent's pieces in a row, we will have a penalty of -20 utility score. For each group of 4 opponent's pieces in a row, we will have a penalty of -100 utility score. This high penalty is because 4-opponent-piece-in-a-row is a highly risky situation for us, and it is just one-step away from the defeat. So, our AI agent should prevent this situation from taking place.

Search Algorithm

After developing the evaluation function, we can head to develop the algorithm that pick the most promising action (move) for current game state. There is an evolution process of developing the search algorithm.

The first simple strategy I developed is picking the action that would lead to the largest utility calculated by the evaluation function (1-depth minimax). The algorithm gets all legal moves first and use a for loop to evaluate the game state created by applying each particular legal move. It will find the game state with highest utility and therefore find the most promising move. This is basically a minimax cutoff algorithm but with a depth of only 1 and it only considers current move without taking next opponent's move into account. This strategy can beat random player

(see result in table 1), but its performance was far from satisfactory when competing with real human.

Testing (10 games for each trial)	RandomPlayer wins	Simple strategy (StudentPlayer) wins
Random first	0	10
StudentPlayer first	0	10

Table 1: Simple Strategy VS RandomPlayer

After that, I integrated this simple strategy into minimax cutoff algorithm. Minimax search is an algorithm for finding the optimal strategy: the best move to play at each state(node).[1] Its mechanism is expanding the complete search tree until terminal states have been reached, computing utilities of the terminal states and finally backing up from the leaves towards the current game state.[1] At each min-node (opponent perspective), back up the worst utilities among its children. At each max-node (our AI agent perspective), back up the best utilities among its children.

We implemented minimax cutoff search algorithm in a recursive way and introduced an argument called *depth*. This variable controls the level of the search tree (not necessary find the terminal states) and in turn controls the computation time. We also need a Boolean variable *max* to indicate whether this is max node or min node. If max is true, this is a max node. Otherwise, it is a min node. The pseudocode this algorithm is shown below:

```
Minimax (int depth, GameState state, Boolean max) {
        If (depth == 0) evaluate game state and return

        For each legal move m:
                Get the game state tmp after processing move m
                Current Utility = Minimax (depth - 1, tmp).          // Recursion

        If (max) Return the move with highest utilities.
        Else Return the move with lowest utilities
}
```

This algorithm works well with depth of 2 under the 2-second constraint. However, as the depth increased to 3, the computation time just skyrocketed to go over 20 seconds. Therefore, I would further upgrade this algorithm again to Alpha Beta Pruning algorithm.

Alpha Beta Pruning is built on top of the minimax algorithm with an extra pruning step. The core idea is that if a path looks worse than what we already have, then discard it (prune).[1] In other words, there is no need to search further if the best move at this node cannot change. In order to identify when the pruning should take place, two integer variable alpha and beta will be introduced to track the bounds of the utilities. The pseudocode is shown below:

```
A_B_Pruning (int depth, GameState state, Boolean max, int alpha, int beta) {
        If (depth == 0) evaluate game state and return

        For each legal move m:
                Get the game state tmp after processing move m
                Current Utility = A_B_Pruning (depth - 1, tmp, !max, alpha, beta)

                If(max) update alpha if needed
                Else update beta if needed

                //pruning
                If (alpha >= beta) break

        If (max) Return the move with alpha utility.
        Else Return the move with beta utility.
}
```

The algorithm has less computation time compared to Minimax Search. It can finish computation for a depth of 3 usually within 3 seconds. Unfortunately, the time constraint for the tournament is 2 seconds. Even though Alpha Beta Pruning with a depth of 3 only exceeds 2-second limit for a very few of cases, I personally would not take this risk. The interesting fact is that it makes no difference if both Minimax Search and Alpha Beta Pruning have a depth of 2. They will return the same result.

2. Theoretical Basis

The theoretical basis of evaluation function, Minimax Cutoff Search and Alpha Beta Pruning is from the lecture and lecture slides. The details of evaluation function policy (defining features and its weight) is derived from intuition and personal experiences of playing this game. The evaluation function is the foundation of minimax cutoff search and Alpha Beta Pruning.

The Minimax Cutoff Search evaluates the utility of leaves (depth == 0) by evaluation function and then back up to the parent who will pick the most favorable one from its self-interest. So do the parent of the parents and so on. For a max node, it wants utility to be as large as possible, whereas for a min node, an optimal opponent does not want our agent to win the game, so it wants the utility to be as small as possible. A graph illustration of the algorithm is shown below:

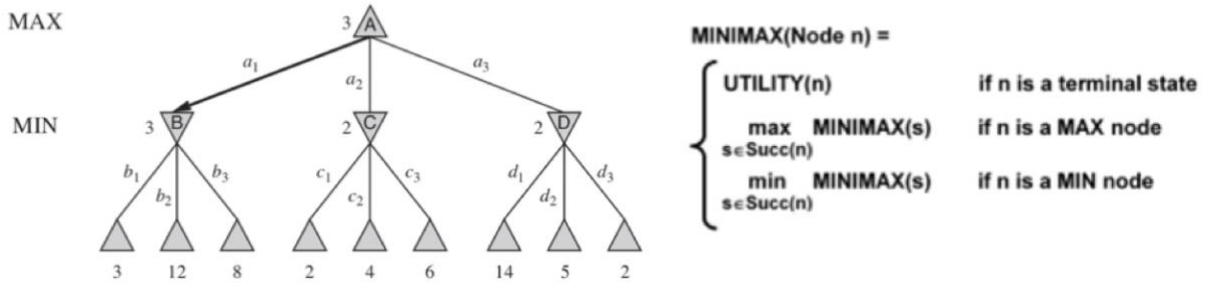

Figure 1: Minimax Search Algorithm[1]

I did not really understand max node and min node at first because I was thinking, for my opponent's turn, it would also want to maximize its utilities and at some time, my agent could be min node as well. In other words, I was stuck in the flipping of perspectives. Later, I realize that my agent is always a max player, and my opponent is always a min player no matter whether my agent moves first, or my opponent moves first.

Minimax search is complete since our game tree is finite. However, minima search is optimal only when it is against an optimal opponent and we have no guarantee for what kind of opponents we will have during the tournament. There might be better strategies for suboptimal opponents[1]. In terms of time complexity, given a branching factor of *b* and a depth of *d*, the time complexity is $O(b^d)$[1]. For Pentago-Twist, there are maximally 36 possible of pieces and 2 choices (rotation/flip) for each quadrant. The branching factor is maximally 288. As stated before, in order to meet the time constraint, the depth can maximally be 2.

As mentioned before, Alpha Beta Pruning is built on top of Minimax Search and have better efficiency. It has two additional integer variable alpha and beta to track bounds on the utility so that it can prune the path if a path looks worse than what we already have.[1] Theoretically, alpha should be initialized to negative infinity and beta should be initialized to positive infinity. For this Java project, we will initialize alpha = *Integer.MIN_VALUE* and beta = *Integer.MAX_VALUE.* At max node, alpha will be updated if the current evaluation value is higher. At min node, beta will be updated if the current evaluation value is lower. If alpha is larger than or equal to beta, then we prune the rest of the children. The order of game states matters. In worst case, the most promising node is ordered at the end and nothing is pruned. Then this algorithm will take the same computation time as Minimax Search $O(b^d)$. On the other hand, if the order is perfect, the time complexity will reduce to $O(b^{0.5d})$.[1] On average, the time complexity is $O(b^{0.75d})$ for randomized action ordering.

3. Advantages and Disadvantage

Advantages

My final submission will be using Alpha Beta Pruning. The advantage of this algorithm over simple strategy is that this one takes opponent's actions into account and have better performance as shown in the table below. Compared with minimax cutoff search, this algorithm has better efficiency in computation time. For example, for a depth of 3, Alpha Beta Pruning can finish computation on average 3 seconds, whereas minimax cutoff search needs more than 20 seconds. Another advantage is that our Agent relies on the evaluation function which could potentially highly accurate if it designed by an export on Pentago-Twist, whereas Monte Carlo Tree Search has certain randomness.[2]

Testing (10 games for each trial)	Alpha Beta Pruning (depth = 3) wins	Simple strategy wins
Alpha Beta Pruning first	10	0
Simple strategy first	10	0

Table 2: Simple Strategy VS Alpha Beta Pruning

Disadvantages

The major disadvantage is that this algorithm assumes the opponent is optimal and using the same Alpha Beta Pruning algorithm and same evaluation function. However, this is not true. There is no guarantee on what strategy that the opponent will use, and their evaluation function can evaluate game states with different policies. Therefore, the assumption that the opponent is optimal is invalid. Another disadvantage is that the depth is too limited for our case. Due to the fact that we only have 2 seconds at each turn during the tournament, the depth is highly limited. After sufficient testing, depth of 2 is safe and can guarantee not violate time constraint. For depth of 3, it will violate the time constraint for some cases. Minimax Cutoff Search can also work for a depth of 2 so basically there is no different between Minimax Cutoff Search and Alpha Beta Pruning. After experiment of competition between Alpha Beta pruning and Minimax Cutoff, I noticed that whoever goes first will win the game. In this case, a Monte Carlo Tree Search will be much more appreciated. Unfortunately, this is already the end of the semester and there are too many assignments to be done, therefore, I was able to allocate more time for developing a Monte Carlo Tree Search. I will discuss this in detail for next section.

Testing (10 games for each trial)	Alpha Beta Pruning (depth = 2) wins	Minimax Cutoff (depth = 2) wins
Alpha Beta Pruning first	10	0
Minimax Cutoff first	0	10

Table 2: Simple Strategy VS Alpha Beta Pruning

4. Further Improvement

The evaluation function can certainly be further improved. I am not an expert in Pentago-Twist, and I defined the features of the game state from intuition and personal experience of

playing this game. The evaluation function can be redesigned by an expert to better reflect the real situation of the game state. Moreover, an evaluation function with better efficiency can help increase the depth for Alpha Beta Pruning while obeying the time limit and then in turn improve the agent performance. Secondly, as mentioned above, ordering can influence the time complexity of Alpha Beta Pruning. Therefore, we can sort one node's children so that more unnecessary nodes can be pruned. Most importantly, I should try to implement Monte Carlo Tree Search algorithm which use random simulations of the game to help the agent determine what move to select.[2] As mentioned before, problems of Alpha Beta Pruning are that: high time complexity, reliance on a reasonable evaluation function and assumption of optimal opponent who is using the same evaluation function. Monte Carlo Tree Search can address these problems.

5. Reference

[1] Jackie CK Cheung, "COMP 424 – Artificial Intelligence: Game Play", 2021

[2] Jackie CK Cheung, "COMP 424 – Artificial Intelligence: Monte Carlo Tree Search", 2021