AI Final Project Summary

Luna Dana - 260857641

April 13, 2021

1 Motivation and Explanation of Program

1.1 Motivation

PentagoTwist is a fully observable game which consists of two player playing against each other. Furthermore, it is a deterministic game. Its purpose is to align 5 balls in a row (vertical, horizontal or diagonal).

When reading this final project, my first thought was to implement a Minimax algorithm. However, Minimax is known to be very expensive since the search space is very large even if we make Alpha Beta pruning. Moreover, It would require to design strong evaluation function to estimate the utility of each board since every board is stored in a double array which is a very space taking data structure in Java.

I then chose to use Monte Carlo Tree Search since its evaluation function only depends on the observed outcomes of the simulations. As we have 2 seconds per turn, my Algorithm has the time to make a lot of simulations and is hence quite precise and relatively fast.

In a usual Monte Carlo Tree Search, you store all the the nodes in a data structure (e.g a tree or a hash-map) and keep the entire structure throughout an entire game. I decided to create a new tree at each turn to avoid the cost of storing the previous nodes. If all the board states were to be stored in a HashMap it would take O(n) at each turn to see if the node has already been simulated with the HashMap.containsValue() Method in Java. That is, since I don't store the tree, I chose to skip the expansion part of the Monte Carlo and directly simulate the node with the best UCT. With this change, every possible new BoardState is simulated at least 300 times.

When starting to implement my algorithm, I needed to balance my node selection to give the same chance to all the moves at first. My first strategy was to look at the best UCT but the agent would then only do a simulation on the first child which had won the first simulation. Secondly, I chose to make the node selection a little bit more random by selecting a random child 1 every 10 selections in order to enhanced the exploration phase and not get stuck in the exploitation phase.

1.2 My Program

My program is based on a Monte Carlo Tree Search with some extra attack and defense strategies. I used the Object oriented structure of java to store the information of each state and each move.

1.2.1 Node

In order to use the Monte Carlo Algorithm, I created a subclass `Node` inside `MyTools`. This class has different attributes and stores

1. The Parent of the Node of type `Node`
2. The BoardState of type `PentagoBoardState`
3. The Children of type `ArrayList<Node>`
4. The Children to expand of type `ArrayList<Node>`
5. The number of Visits of type `int`
6. The number of Wins of type `int`

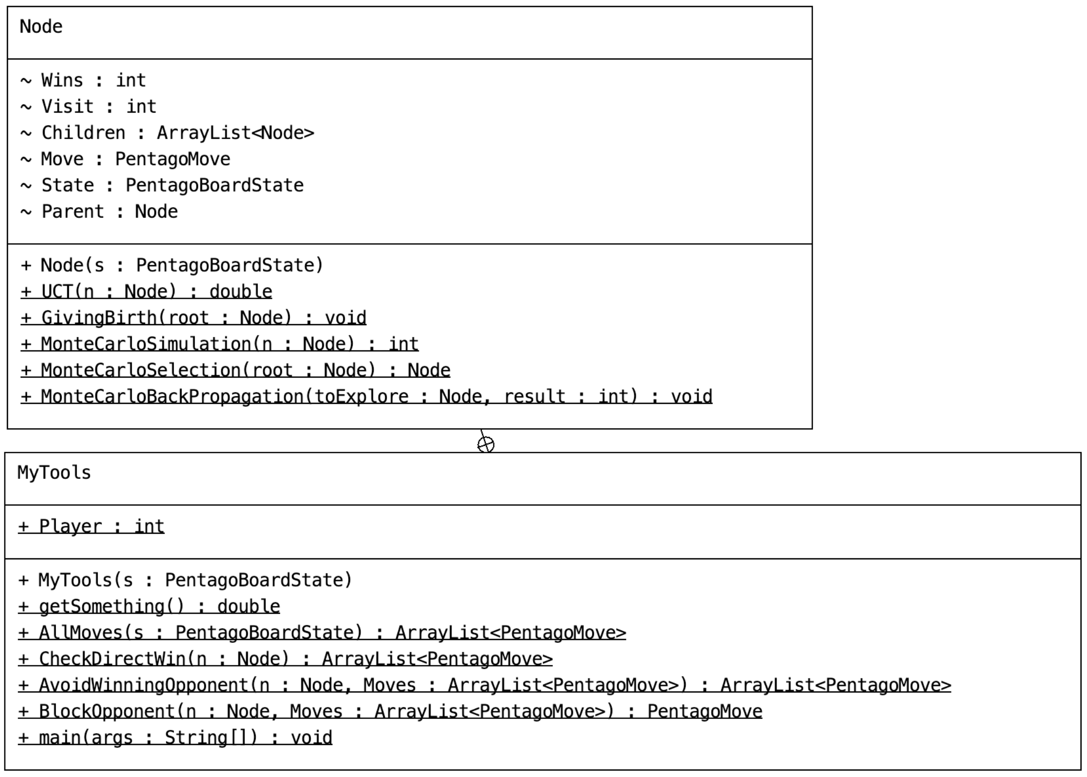
Node
~ Wins : int
~ Visit : int
~ Children : ArrayList<Node>
~ Move : PentagoMove
~ State : PentagoBoardState
~ Parent : Node
+ Node(s : PentagoBoardState)
+ UCT(n : Node) : double
+ GivingBirth(root : Node) : void
+ MonteCarloSimulation(n : Node) : int
+ MonteCarloSelection(root : Node) : Node
+ MonteCarloBackPropagation(toExplore : Node, result : int) : void
MyTools
+ Player : int
+ MyTools(s : PentagoBoardState)
+ getSomething() : double
+ AllMoves(s : PentagoBoardState) : ArrayList<PentagoMove>
+ CheckDirectWin(n : Node) : ArrayList<PentagoMove>
+ AvoidWinningOpponent(n : Node, Moves : ArrayList<PentagoMove>) : ArrayList<PentagoMove>
+ BlockOpponent(n : Node, Moves : ArrayList<PentagoMove>) : PentagoMove
+ main(args : String[]) : void

Figure 1: UML Diagram of my code structure

1.2.2 Methods

1. `GivingBirth(Node root)` Create the tree structure by giving birth to the `Children` nodes and setting the root as the `Parent` node. I first clone the board state because we should not change the initial object and then I apply all the possible moves to the temporary board. For each new board, I create a new child with its associate move and board.

2. `AllMoves(PentagoBoardState s)` takes as input a BoardState and return all the possible moves that can be executed from this specific board. If multiple moves lead to the same boardstate (for example placing a ball and then rotating or flipping an empty quadrants will lead to the same state), the method only keeps one of the move. This technique allows me to gain in space efficiency by reducing the number of moves consequently. Moreover, it allows the list of move to be shorter which helps for time efficiency since travelling a list of move is minimum O(n) of time complexity.

3. `CheckDirectWin(Node n)` takes as input a `Node` and check all the moves that can lead to a direct win. During each iteration of Monte Carlo simulation, this method will be called and if a winning move exists, it will directly return it in `chooseMove(PentagoBoardState boardState)`.

4. `AvoidWinningOpponent(Node n, ArrayList<PentagoMove> Moves)` takes as input a Node and an `ArrayList` of possible moves. It checks for each of my move, all the next possible moves from the opponent. If the opponent have a winning move, it adds my initial move to a list of bad moves that should absolutely not be played. I will then remove the list of bad moves from

1.2.3 Monte Carlo Simulation

1. `MonteCarloSelection(Node root)` takes a input a Node and check if all the children have been expanded. If one child c is not expanded yet, it removes c from the list of child to expand and adds it to the list of Children. If all children have been expanded, he select the child with the best Upper Confidence Bound. `UCT(Node root)` takes a node and returns a `double`.

2. `MonteCarloSimulation(Node n)` takes as input a Node and simulate a random playout by applying random moves consecutively until there is a winner.

3. `MonteCarloBackPropagation(Node toExplore,int result)` takes as input the Node explored and the result of its random playout. It then propagates the result to his parents. For this I use a while loop with a pointer which updated the attributes of the object until the root is reached.

1.2.4 Super Structure

In the StudentPlayer class, a while loop runs for 2 seconds which is the allowed playing time. During this 2 seconds, the 3 steps of the Monte Carlo Simulation are played (with 3 function calls). After the while loop has been executed. The child with the best wins over the nuber of visits is selected. I could also have chosen to only count the number of visits since the AI agent usually choose to visit the Node with the highest UCT. However, I have chosen to implement a

2 Theoretical Basis

We can formulate our problem as a seach problem.

1. States which are the boards.
2. Operators which are the moves.
3. Transition functions that defines the result of a move and check in the state is a terminal state that is if the move led to a winning board.
4. Terminal States as winning boards
5. Utility function which is the number of wins on the number of visits. We are looking to maximize this ratio during the simulations.

Monte Carlo Tree Search (MCTS) is a famous algorithm introduced in 2016. it is very famous in Game Playing and especially in Board Games such as Chess, Bridge or Shogi. MCTS uses a Upper Confidence Bound to calculate the best child to expand. It then simulate a random playout on the child and propagate the result to the node's parents.
It has been proven that Monte Carlo Search Tree converges to Minimax

2.1 Steps of the Search

1. Selection. Start from the initial position and select the best child according to the UCT. However, all the child need to be visited
2. Expansion. Expand the tree by giving birth to the next node.
3. Simulation. Simulate a random playout and store the result.
4. BackPropagation. Propagate the result of the simulation to the child and its parent until reaching the root.

2.2 Upper Confidence Interval (UCT)

MCTS uses a Upper Confidence Bound to calculate the best child to expand. The formula I am using is adapted from the one seen in class (Lecture 10, slide 43 from class). The first part of this addition corresponds to the exploitation phase. That is the ratio will be higher for nodes with many wins. The second part takes into account the exploratory phase.

$$UCT = node.Wins/node.Visits + sqrt(2) * log(node.Parent.Visit/node.Visit) \tag{1}$$

3 Strengths and Weaknesses

3.1 Strengths

1. Efficiency. My AI does in average 300 simulations per child. The minimum visit in average is 47. Hence, even the less visited child still got visited a lot of time. It removes the chance of not seeing a loosing branch.

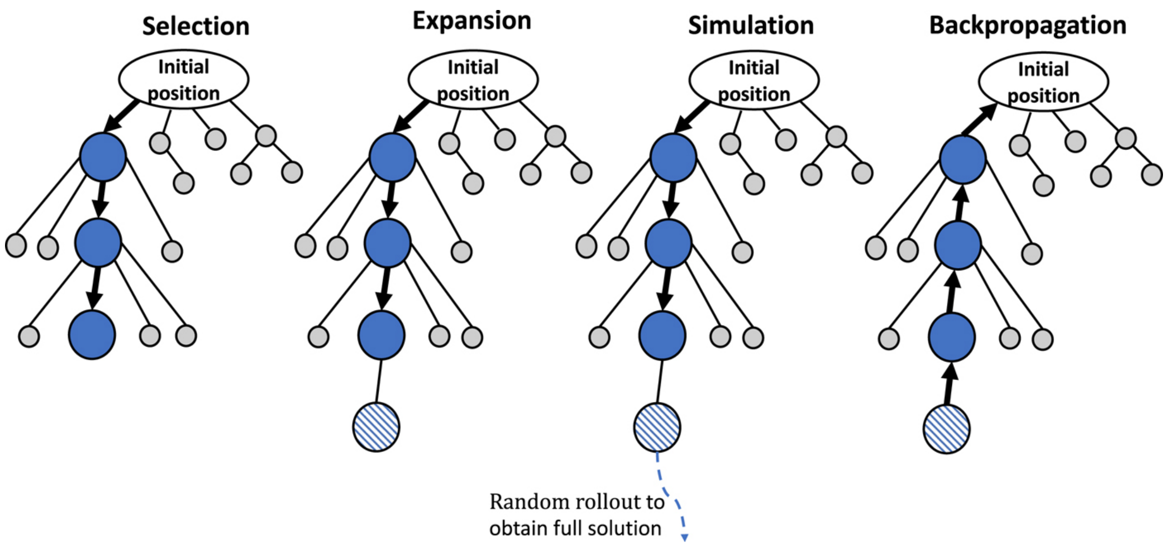

Figure 2: Steps of the MCTS

2. Defense. My AI always checks the opponent's winning moves at each turn in order to block him. That is, it counters famous PentagoTwist winning strategies like THE STRAIGHT FIVE, THE TRIPLE POWER PLAY or MONICA'S FIVE. I have tried using these strategies as a human player and my AI agent won each time. When I play against my player I get a draw 98 percent of the time, showing my agent knows has a good defense.

3. Attack. My AI checks at every turn if there is a winning move after turn 2 since there can't be a winner if only 4 balls have been played the board. If a winning move is found, it is directly returned.

3.2 Weaknesses

1. Heavy Data Structure. Since we use object oriented with Java and each Node have a lot of attributes, it is very costly to store an entire tree. That is, if a Node have been visited in previous turns, my AI agent can't gather its information's. For this reason, I have removed the expansion phase of the Monte Carlo Tree since at each turn, a new tree is built.

2. Originality. I guess my AI agent is efficient but maybe not precise and strategic enough to beat very good/professional players. It is the theoretical disadvantage of the algorithm : a single branch could lead to a direct loose and the algorithm can't notice it. Moreover, a lot of other students may have chose the same strategy.

4 Other Approaches

4.1 Alpha Beta Pruning

Alpha Beta pruning was the first idea that came to my mind because I immediately knew that a Minimax algorithm won't be efficient enough and Alpha Beta pruning is its improved version. However, it require a fixed tree depth since it costs to much to search the entire tree. It also calculates the value of all the legal moves which is not optimal. When I tried to implement it, it could not look ahead after 4 or 5 moves. With MCTS, we do not have to fix a number of simulation since we can just make as many simulations as possible during the 2 seconds.

5 Improving my Player

1. In order to improve my player, I would like to store the nodes visited into a data structure (like a HashMap) in order to keep track of a state if this state appears again in another turn. I did not implement it since it was not efficient enough but with a good implementation it could make the agent more precise.

2. I would design some heuristics with the goal to reduce the number of possible moves and make the search space smaller.

6 References

1. ML — Monte Carlo Tree Search (MCTS) https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
2. Pentago, the mind twisting game https://www.fgbradleys.com/rules/Pentago.pdf
3. Monte Carlo tree Search Wikipedia Page https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
4. Slides on Game Playing and MCTS from class
5. Monte Carlo Tree Search https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa