COMP 424 Final Project Game: *Colosseum Survival!*

Authors: Maggie Shao and Lin Yuan

Colosseum Survival is a two-player strategy board game where players move and place a barrier on a side of the cell after each move. The goal is to separate the two players into two different zones by connecting the barriers. The player with a bigger area, that is, the larger number of blocks in the region, wins the game. This final project consists of creating a student agent capable of playing the game and making decisions based on AI algorithms that we discussed in class.

1. Explanation of Program and Motivation of Approach

The architecture of our program is as follows. We have 4 classes: Move, State, MCTNode and StudentAgent. A move object represents a move that a player can perform, described by the following attributes: position and barrier. A state object represents a state of the game and is defined by a chess_board field, both players' positions and a field indicating the player's turn. Then, a MCTNode represents a node in our search tree. It consists of a state of the game, a heuristic score, the move performed to reach the present state and references to adjacent nodes. Finally, it is in the StudentAgent class that we return the best move calculated using the previous classes.

After trying out a couple of approaches, such as Minimax with alpha-beta pruning and Monte Carlo Tree Search, we decided to apply a much simpler method, which is to use a traversal tree search with heuristics. We were inspired by our previous Monte Carlo method to apply backpropagation to update the heuristics. As for our heuristic function, we chose to compute a final score using a rules-based algorithm. We originally came up with a set of useful strategies, but weren't able to implement them in our code as they weren't complete and would subsequently mislead our agent. Hence, our heuristic is only composed of a single function win_next() that determines whether a state is a winning state, a losing state or a normal one. The function does so because it is a symmetrical heuristic, meaning that if our agent wins on the current state of the board, a score of +1000 will be assigned, while a score of -1000 is returned if the adversary wins the current game. Finally, a normal board state will not generate any score.

Our algorithm is based on 3 steps: selection, expansion and backpropagation, which are executed repeatedly until the time limit is reached. For the selection step, we start

with the root node R and do a traversal tree search, more precisely a Depth-First Search with a depth limit, that selects a leaf N at when the maximum depth is reached. For the expansion step, we will look through all the child nodes of the selected node N. The child nodes are all the state of the board resulting from the execution of a valid move generated by the function get_legal_moves(). Next, we compute the heuristic score for all children and aggregate them by the maximum score. Then we backpropagate to update the scores of all nodes from N to R.

We used this approach as we had to make a trade-off between the accuracy and the speed of the program. In fact, the time limit will restrict the number of children we select and expand on, so that our agent does not exceed the 2 seconds limit. In addition, we set a depth limit as well to the search in the selection and expansion part. This prevents the algorithm from expanding a root node till an endgame node, as it will take way too much time to finish visiting a single child of the root node and will reduce the number of children we can successfully visit within the time limit. Hence our heuristic search will provide a good step to take, despite it not necessarily being the optimal one. This algorithm allows our agent to have a win rate of 92.5% over 1000 games against the random agent, which is satisfiable.

2. Theoretical basis of approach

The theoretical background of our process is inspired by the Monte Carlo Tree Search, as it roughly follows its selection, expansion and back propagation steps. MCTS uses policies, which is a mapping to select actions.[1] The selection phase uses a tree policy, usually the UCB formula, near the root in order to select a path. Unlike the default policy, which can be a random move, the tree policy ensures a good coverage over possible states. The expansion step of a MCT expands the tree until a leaf node is reached. The backpropagation step consists of updating the heuristic score of all states visited during the previous two steps.

As mentioned in Professor Cheung's lecture on Informed Search [2], a heuristic represents an intuition that can estimate the real cost to a goal. There are many search algorithms that are based on heuristics such as Best-First Search that expands on the most promising node with respect to the heuristic evaluation. However, it is too greedy and doesn't consider the accumulated cost so far. As a solution, our heuristic search works because we backpropagate the maximum aggregated score from children of a selected node up to the root.

3. Advantages and Disadvantages

Advantages

Our approach allows the agent to predict a few moves ahead instead of acting in a greedy way, since the root node will choose to take a step according to the children with the best heuristic, which is calculated by expanding each child and aggregating the heuristics of their own children. Furthermore, our symmetric heuristic function win_next() that identifies a decisive game state is efficient and inexpensive. In fact, it allows our agent to make informed decisions as it allows it to determine the winning potential of choosing a child over another, hence highlighting an obvious advantageous move.

Disadvantages

As mentioned previously, our heuristic evaluation is uniquely composed of a win_next() function. Although it is a good way to assign a score to a node, it is not enough to make the score more accurate. In fact, most of the time, there is no decisive outcome to the game, especially in a big board and towards the beginning of the game. In this situation, the heuristic function will not be very useful, causing the agent to make random moves. Furthermore, the size of the board affects the accuracy of our agent in another way. In fact, if it is too large, our search algorithm will not be able to traverse all the children of a node within our 2 second time limit. Hence, the agent will only look at the same first children and might neglect the ones with a good winning potential, making its decisions, to some degree, random.

4. Other approaches

At first, we considered using the Minimax algorithm with alpha-beta pruning. However, after doing some research, we concluded that, although it is a useful algorithm for simple games, it is not suitable for our case. To begin, Minimax assumes that both the player and the opponent are optimal, which does not meet our needs since our agent has to play against a random agent. Furthermore, it has a big branching factor [4], thus slowing down the process to reach the goal, making it not fit for the 2 second time limit for each turn, as well as the restriction on the memory usage.

We also started to implement a Monte Carlo Tree (MCT) Search, since as mentioned in professor Cheung's lecture on MCT Search[1], this approach is easy to apply for games

with a large set of states such as this one because it does not rely on a specific evaluation function. Therefore, we did not need to understand how to really play the game to make a decent agent. However, it is precisely because it relies on random simulations to choose the best steps that we decided to not move on with this algorithm. In fact we believe that, because the size of the neighborhood is too large, taking a random sample of it would dismiss many moves with great potential.

Finally, we tried implementing a rules-based heuristic by separating the strategies into three stages of the game: beginning, middle and end of the game. We can separate these stages by assigning them each a different weight. For example, the strategies useful for the start of the game will receive a smaller score compared to the ones for the middle and the end of the game. An example of early game heuristic would be assigning a score of +1 the closer the position is to the center of the board, with the goal of controlling the center, thus creating more opportunities for our agent. A mid-game strategy is being able to break through. If our agent finds himself in a disadvantageous position and wants to get away, it will be much easier to do so if it isn't next to a barrier that is connected on both ends. Hence, a score of +5 can be assigned if our agent moves to a state that allows an easy breakthrough and -5 otherwise. Finally, towards the end-game, we want to imitate how humans play the game by identifying a region that we can potentially control. To do so, we can implement an algorithm that checks if it is possible to separate the two players on the current board state with three or less extra barriers. Then, by obtaining and comparing the number of cells each player controls, our agent will decide whether it wants to avoid this outcome or play along with it. Of course, this evaluation will be assigned a much higher score than the previous two, as it will not affect the start and middle of the game, but will be important towards the end. We were unfortunately not able to successfully implement the last strategy, hence we decided to not include the previous ones either, as they will mislead our agent.

5. Improvements

As mentioned in the disadvantages, our heuristic function's accuracy can be improved by implementing more rule-based algorithms. We can finish implementing the heuristic evaluations mentioned in our other approaches.

Furthermore, we noticed that the Monte Carlo Tree (MCT) Search works independently of our approach, hence making it a good add-on to improve our agent. In fact, including a MCT simulation to our current heuristic based agent is a solution to one of the disadvan-

tages, as it reduces the number of nodes to traverse, thus making our agent more efficient and reducing the chances of it doing a random move. The heuristic can also filter out some bad moves at the simulation stage of the MCT Search. This will allow the MCT agent to select a less random sample and solve our initial issue with using MCT Search.

6. References

1. Cheung, J. (2022). Artificial Intelligence Monte Carlo Tree Search [Lecture Slides] McGill University, COMP424.

2. Cheung, J. (2022). Artificial Intelligence Informed Search [Lecture Slides] McGill University, COMP424.

4. ProfessionalAI. (23 Mar 2020). Minimax Algorithm. https://www.professional-ai.com/minimax-algorithm.html [Accessed 10 April 2022].