COMP 424 Final Project Report

Author: Claire Feng

1. Abstract

1.1 Explanation of the Program

In this project, I chose to implement the algorithm of moves of my agent using a pure Monte Carlo tree search with some heuristics for choosing the possible moves for expansion. The student_agent.py file includes the MCTS class definition, and some helper functions necessary for the implementation. When the step function is called from world.py, A MCTS instance is created, and the find_move() is called on it to choose the next best move. I reused three helper functions from world.py: random_walk(), set_barrier(), and check_endgame(). Other helper functions I defined include:

- remove_barrier(opposite to set_barrier()).
- rollout(), which takes turns randomly moving the two agents starting from an initial board state and runs until the game ends, and returns the final number of scores/blocks of two players.
- surrounding_barriers(chess_board, pos), which calculates the number of barriers that are close to(within a quarter of the area of the board size) the agent at a certain position on the chess board.
- MCTS.get_move(), which uses two greedy heuristics for choosing the most promising move among 50 random walks.
  - The first heuristic is straightforward, it always avoids having more than two barriers enclosing itself, and it tries to enclose the adversary with more than one barrier.
  - The second heuristic is more general, intuitively, the more barriers there are in an area close to an agent, the more it tends to lose, especially for a random agent it's easier for it to trap itself.

In the MCTS class, the find_move() function first calls the expand_children() functions, which uses the get_move() method to choose the 10 most promising moves for expansion, then performs 20 rollouts on each of them. The utility of each move is calculated as the sum of $(myscore - advscore)/boardsize$ of the 20 rollouts. Finally, the find_move() method returns the move with the highest utility.

1.2 Motivation

The motivation for this approach is the high level of uncertainty in playing this game. For example, the size of the game board, the positions of the initial barriers and the two players, and the maximum number of allowed steps are all randomly chosen for each game. Also,

there is no clear winning and losing position in the middle of gameplay. With each step, the situation can change dramatically and turn from almost losing to winning. Thus, due to the high flexibility of the game created by the game rules, it's hard to develop an accurate evaluation function for predicting how far the current state is toward the goal state, so I choose to use MTCS, which depends less on the evaluation function, and more on randomly simulating plays, which suits the flexible nature of this game better.

2. Theoretical Basis

Referencing the textbook (Artificial Intelligence: A Modern Approach, 4th US ed. by Stuart Russell and Peter Norvig) Chapter 5.4

The Monte Carlo Tree search was designed to play games that have a large branching factor and highly flexible gameplay that minimax with alpha-beta pruning cannot handle, like Go. A basic MTCS does not rely on an evaluation function, instead performs simulations (rollouts) that take turns as both players until the game ends, and selects the final move with the highest winning percentage. To get the optimal play, for most games it's important to balance exploration (selecting states that haven't been explored much) and exploitation (selecting states that have enough simulations and have a good winning percentage). The most used selection policy is the UCT (upper confidence bounds applied to trees), which ranks the moves based on the formula

$$\frac{U(n)}{N(n)} + C * \sqrt{\frac{logN(parent(N))}{N(n)}}$$

, where the first term is the exploitation term(the winning percentage of the node n), and the second term is the exploration term. C is a constant exploration weight that balances the exploration/exploitation tradeoff. In MCTS, the growing of the search tree follows four steps for each iteration:

1. Selection: starting from the root node, select a child node according to the selection policy, repeating this process until reaching a terminal node (game ends).

2. Expansion: generating a new child node of the selected node.

3. Simulation: perform a playout starting from a node by taking turns to play for each player until the game ends, not recording the moves. The time complexity is linear since only one move is taken for each step.

4. Backpropagation: the result of a simulation is used to update all the parent nodes along the path to the root node. The playout number N is incremented by 1 for all nodes, but only the utility (U) of the nodes that are on the same side as the winning player are incremented.

3. Advantages and Disadvantages

3.1 Advantages

The program almost always wins against the random player (100%), because it has some ability to foresee the future outcomes to make the decision. However, it is still not intelligent enough and can still lose to a human player. Using greedy heuristics when choosing possible next moves for expansion, this agent is not trapping itself and can make obvious moves to try to trap the adversary agent. So it's harder to capture and has the intelligence to some extent, making it more fun to play with compared to a random agent.

3.2 Disadvantages

However, because of the limitation of computational resources (2s timeout), it cannot foresee all the future states of the game boards and perform enough simulations that give a perfect movement at each step. Also, compared to humans, who can look at the game board visually, the heuristics that can be used by the agent is not perfect and tends to be short-sighted. It can rule out irrational random moves that trap itself most of the time, but it's not perfectly capturing the best move that is sometimes quite obvious to human eyes. In summary, the agent is intelligent to some extent but is still not competitive enough to always win against a human player.

3.3 Expected Failure Modes

Although this agent will not trap itself immediately like a random agent, it can still move around stuck in a local optimum and fail to escape a large trap which takes multiple steps and requires looking more steps into the future. Besides, one of the greedy heuristics makes the agent try to reduce the number of barriers in the nearby surrounding area, which can also contribute to the issue of being stuck in a local optimum and failing to escape the larger trap.
Another problem with my agent is that it cannot accurately approximate the number of blocks itself and the adversary is trying to gain on the two sides of a longer barrier. In the situation that the opponent is close to the barrier but has more blocks behind it, it is possible for my agent to close off the opponent according to the "trap the opponent" heuristic, which causes itself to lose.

4. Comparison with Previous Approaches

Before implementing the heuristics, the previous agent simply chooses 10 random moves and does 10 rollouts on each of them, and returns the one with the highest number of wins, and it loses to the current agent 90 percent of the time. I think these three differences make the current more intelligent:

1. The heuristics make the most difference, as it adds some intelligence to the agent and saves computational resources for simulating the most promising moves.

2. The calculation of utility. Compared to simply calculating the number of wins out of all rollouts, the calculation using the formula $(myscore - advscore)/boardsize$ is more

informative and accurate, thus having a better prediction of which move to choose as a final decision.

3. The number of potential moves chosen for doing simulation and the number of simulations performed on each move. With a larger board, the max_step is usually higher, so there will be more possible moves starting from one state, so naturally, the number of potential moves chosen should be more than on a small game board. Instead of fixing it to 10 moves, the current agent makes it a function proportional to the board size, and inversely proportional to the number of simulations. This approach can also balance the split of computational resources spent between choosing potential moves and simulating playouts on them, similar to the exploration/exploitation tradeoff. However, as it cannot accurately control the time it takes for these two processes, in some rare cases the time it takes can exceed 2s, causing it to lose one move.

I've also tried implementing the selection policy using the UCB1 formula and using it for selecting the next node to perform a rollout. However, by autoplaying this agent with the current agent, the current agent always achieves a higher win percentage, so in the end, I chose not to use this formula. It is possible that the number of simulations that are allowed in the limited time is not large enough to reflect the advantage of balancing exploration and exploitation.

5. Possible Improvement

Although I've implemented some heuristics, it can still be improved in logic and implemented in more detail to make the agent more "intelligent". For example, the agent could try to identify long barriers on the game board by simulating moves along the barrier until it reaches a position where it can extend the barrier. Then it needs to decide whether it can reach that position, which can be done by taking multiple random walks with the fixed step number, and checking whether the position is in the result set. Moreover, it needs to check if extending the barrier will increase the number of its potential winning blocks, which can be approximated by the position of the barriers.
Also, as mentioned in the expected failure mode section, the current agent is still too short-sighted and can stuck in the local optimum and fail to escape a larger trap. Since the current implementation is just a simplified idea of MCTS that looks no more than one step into the future, it still makes moves that look random to humans. To improve this, it is necessary to introduce the tree structure to properly implement the complete Monte Carlo Tree Search algorithm with backpropagation. By keeping track of the current node that represents a board state and continuing to expand the tree at every call to the step function, we can store multiple MCTs of different board sizes in the memory and speed up future response time and save time for more exploration and exploitation for better results.