Game Project1 – Battle Sheep

Implementation

MCTS is implemented as the main algorithm for Battle Sheep. The meta below determines how the tree evaluates explore-exploit trade-off. More details can be found in the following readings.

```
class MCTSMeta:
    EXPLORE_WEIGHT = 2.0 # for ucb exploration
    H_WEIGHT = 0         # for heuristic
    H_DECAY = True       # gradually reduce weight of heuristic as game goes on
```

MCTS

The class **MCTS** implements MCTS algorithm with some extra modifications. The tree stores only the game state at root for better efficiency. The explored states can be reached by traversing the nodes and following the corresponding **Move**s defined in children **Node**s.

```
self.root_state = deepcopy(state) # game state at the root
self.root = Node(                 # root node of the tree
    **self._get_meta_pack(),
    owner=state.current_player,
    plays=0,
)

self.node_cnt = 0     # total number of nodes explored
self.simulate_cnt = 0 # number of simulations
self.reset_cnt = 0    # number of resests due to unreached state
```

During experiments, however, the other agents may play some moves which the tree has not explored yet. Therefore, we added an additional class function **_reset** to handle this situation (set the root state of the tree to the current game state and restart the algorithm).

The 4 classes below help **MCTS** analyze and respond to the environment

1. **Node**

   The node of MCTS. It keeps track of important properties for the algorithm.

   ```
   self.owner = owner # current player of the node
   self.move = move   # how to move from parent to here
   self.parent = parent
   self.plays = plays # number of plays of the player (owner) so far
   self.children = {} # { child.move: child, ... }
   self.N = 0         # times this position was visited
   self.Q = 0         # total points of all leaves stemming from here
   self.H = None      # heuristics
   ```

   Explore-exploit trade-off is evaluated as below (The idea of the design is elaborated in **Experiments & Experiences** -> **2.Heuristics**):

```
# exploitation + exploration
ucb =  self.Q / self.N + self.explore_weight * sqrt(log(self.parent.N) / self.N)

if self.h_decay:
    # rely less on heuristics as gane goes on
    # +1 to avoid zero division error
    return ucb + self.h_weight * self.H / sqrt(self.plays+1)
else:
    return ucb + self.h_weight * self.H
```

2. **State**
   - keeps track of the environment
   - return valid moves for current player
   - calculate the points of each player if game over

3. **Move**
   - Define 3 types of moves:
   - initiate the 1st move (for **InitPos**)
   - normal moves (for **GetStep**)
   - not moving at all when there is no valid move (for **GetStep**)

4. **Group**
   Given **mapStat**, return connected regions of a player for **State** to calculate scores.

InitPos & GetStep

Finally, MCTS takes care of both following these steps:

1. Update the state (play what the other 3 players just played)
2. Expand the tree within time limit
3. Choose the best move based on the number of times the node was visited (**Node.N**)

Experiments & Experiences

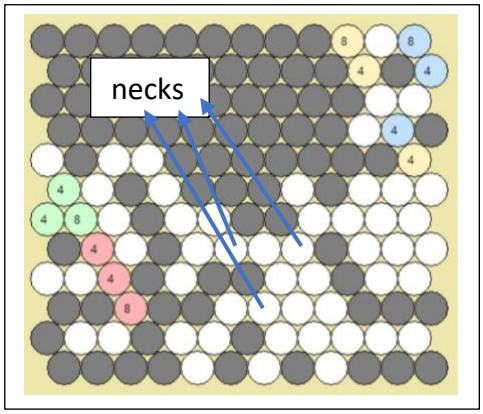
1. InitPos

Originally, we did not want to complicate the design so **MCTS** was not responsible for **InitPos**. Instead, a function named **evaluate_init_pos** (not included in the submitted code because it was not used in the end) was defined to make the decision. It evaluates a given position by summing up the farthest distance it can go in all 6 directions (the distance is divided by 2 if there is an opponent in the direction). It returns $-\infty$ if the position is invalid. The higher the sum, the better the position.

However, it turned out that this can lead to some undesired outcomes. When the map is long and narrow, it tends to choose positions having fewer (2~3) free directions. It may have a large sum according to **evaluate_init_pos** but this is probably not a wise move. Such a position will soon be blocked by other players and end up with a low score.

Some different ways we tried to select initial position:

- Instead of sum of distances, use sum of $log(distance)$ so that the number of free directions has more influence on the decision
- Simply the number of free directions
- Choose a neck on the map (positions which separates the map into 2 or more unconnected regions). In this way, the player can monopolize 1 of the regions if no other initiates there



Unfortunately, all of them could end up with extremely undesirable positions in some specific circumstances. In addition, the 1st player performed especially bad in most of the games. The main reason was probably that no function could anticipate where the other players will play in **InitPos**. However, MCTS could solve the issue by nature. Consequently, we modified MCTS and discovered that it indeed outperformed any of the functions above in most cases, which was why **evaluate_init_pos** was not used in the end.

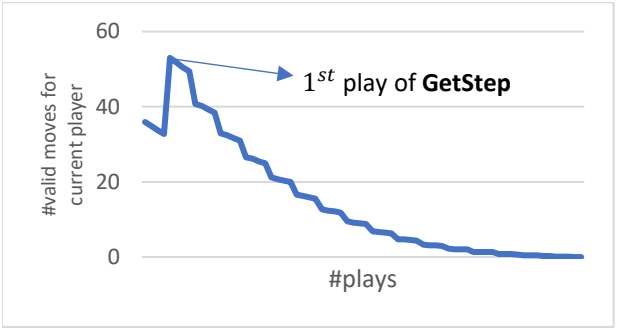
2. Heuristics

In the experiments, we found that MCTS on average resets (please refer to **Implementation -> MCTS -> _reset** for details) about 7 times per game. We believed that this can greatly harm the performance because all the previously explored states would be discarded. Supported by the statistics below, the more frequent MCTS resets, the worse the final results. Except for MCTS which ran out of valid moves early because fewer plays of course leads to fewer resets. Such cases (less than 10 plays) are excluded in the table below.

Rankings	1st place	2nd place	3rd place	4th place
Average number of resets	5.57	6.38	7.66	8.77

The graph below is plotted based on roughly 30,000 games played randomly. It illustrates the relationship between number of plays so far (horizontal axis) and the number of valid moves for current player (vertical axis).

As shown in the graph, the branching factor of game tree reaches its peak in the beginning phase, which is where most of the resets take place, and drastically declines as the game moves on.



Combining the facts above, it is most likely that additional efforts to help MCTS warm up can achieve tremendous improvements. As a result, we decided to combine heuristics and UCB to evaluate explore-exploit trade-off:

$$\boldsymbol{UCB + heuristics / decay\ factor}$$

Some constants are omitted here. The complete formula can be found in **Node**.

- heuristics = the number of valid moves of current player
- decay factor = $\sqrt{number\ of\ times\ the\ current\ player\ has\ played + 1}$
  (+1 to prevent zero division error)

The design intends to make use of heuristics to resolve the slow warm up weakness of MCTS. In the early game where resets happen frequently, heuristics should help the tree expand itself more "cleverly". While the game goes on, the decay factor gradually reduces the reliance of MCTS on heuristics to make sure that it can exploit the information gained from exploration.

To verify the correctness of the idea, we conducted 2 experiments.

Experiment 1 (all 4 MCTS without heuristics decay):
2 MCTS with heuristics vs 2 MCTS without heuristics
Experiment2 (all 4 MCTS implement heuristics):
2 MCTS with heuristics decay vs 2 MCTS without heuristics decay.

The tables below summarize the results.

Experiment 1 (147 games)

	average scores	**average points**
h_weight = 0.1 (using heursitics)	54.88	2.98
h_weight = 0 (no heursitics)	49.12	2.51

Experiment 2 (132 games)

	average scores	**average points**
h_weight = 0.1 h_decay = True	55.79	2.83
h_weight = 0.1 h_decay = False	50.43	2.66

(The playing order was determined randomly)

The statistics above indicate that MCTS implementing heuristics and using decay factors outplay those without heuristics or decay factors. In consequence, MCTS with the following settings is chosen as the final agent:

- explore_weight = 0.5
- h_weight = 0.1
- h_decay = True

Conclusions

MCTS is more powerful than anticipated. In the beginning, we mistrusted algorithm and attempted to use a separate function to select the initial position. However, all of them turned out to be less powerful than MCTS.

However, the slow warm up nature does cause some problem as shown by the relationship between number of resets and the game results.

Futures Works

1. We did not spend too much time on parameters tuning because each player has up to 5 seconds to think which is time-consuming. There can be some room for improvement here.
2. Basic MCTS was implemented as the main algorithm in the project. However, there are many other improved versions. For example, quality-based or RAVE MCTS. It will be worth it to try out some of them and compare their advantages and disadvantages.