Artificial Intelligence Capstone Project 2

Team name: #8 Potatoes Killer

Members: 110612117 張仲瑜 110550128 蔡耀霆 110652032 許元瑞

I. Introduction

The objective of this project is to develop a game-playing AI agent for a board game called “Battle Sheep”. In the implementation, we use the MCTS algorithm to help us find the best step and get a score as high as the agent can.

II. Algorithm and methods

The Monte Carlo Tree Search (MCTS) algorithm utilizes the current state information to perform four crucial parts to make the next decision. They are selection, expansion, simulation and backpropagation respectively.

- **Selection**:

  Starting from the root, the agent selects the optimal leaf node
  (The function is called recursively) currently in the MCTS tree with some strategy.

- **Expansion**:

  When the leaf node is reached, the agent will expand all the legal moves as the child of that node.

- **Simulation**:

  The agent will simulate all the added nodes in the expansion step with random moves until the game is over or the depth limit is reached.

- **Backpropagation**:

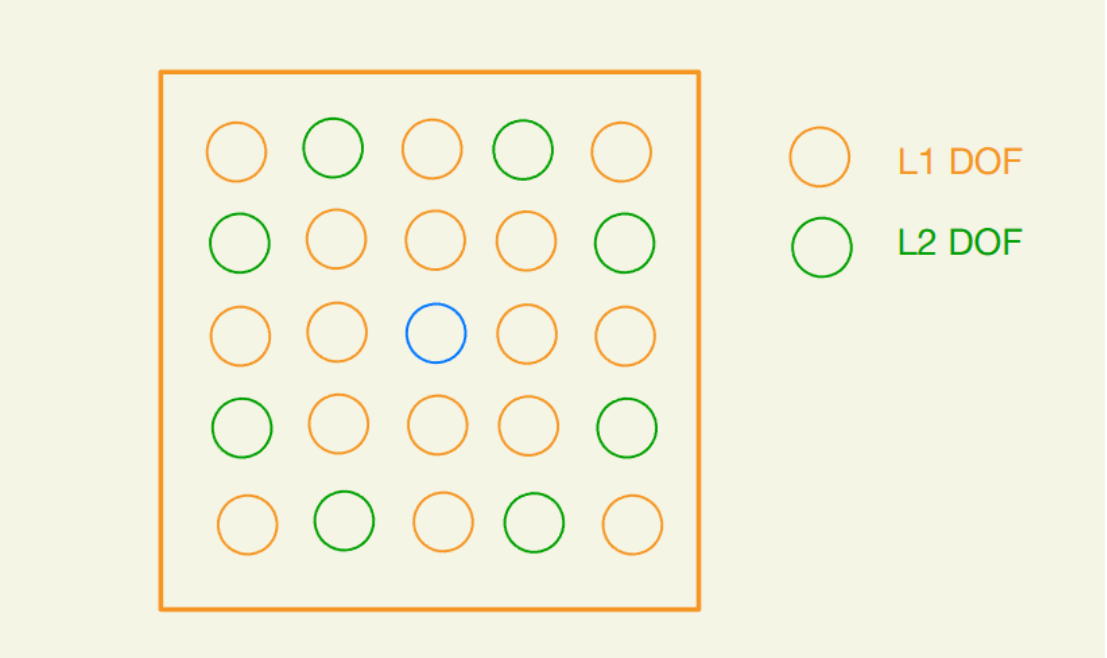
  Add the score of the simulation result all along from the leaf to the root. We do the back propagation to update nodes' variables including total visits and score to get a new uct score for the next selection.

- **Initial Position**:

```
328   def InitPos(mapState):
329       InitPos = [0, 0]
330       InitPos = calculate_DOF(mapState)
331       return InitPos
```

InitPos() will get the position that has the most advantage.

First, we define the (L1)DOF(degree of freedom) of a block as the empty blocks in all 8 directions from that block. On top of that, we then define L2_DOF of a block as the summation of the DOF of all the blocks in its DOF.



The graph above shows the L1_DOF and L2_DOF of the blue block. That is, the L1_DOF of it is 16, and L2_DOF of it is 24 if all the blocks were available.

Second, the other factor when considering the initial position is the distance between that block and the others' initial position since chances are good that you can build your own kingdom when nobodies are nearby. We simply calculate it by the summation of the Euclidean Distance between others' initial distance.

- **Game Environment**:

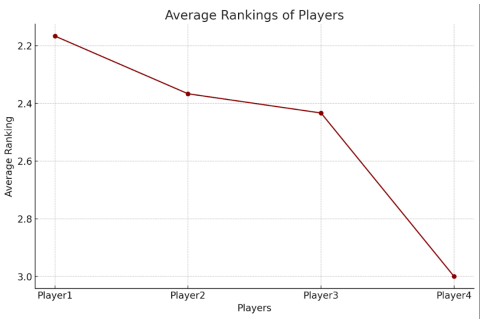
Class GameEnvironment will connect our program to the game server,which makes the action made in MCTS can be successfully passed to the server.

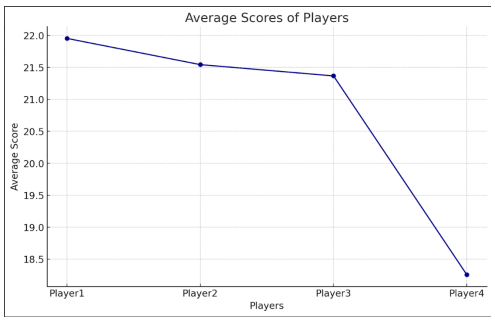
III. Experiment,Observation and Discussion

A. Impact of order(sequence)

- **Game1 : The basic form**
  (4 players, 16 sheep per player, 64 cell playing field)

Four L2_DOF players (no distance), 30 games :





100Samples	P1	P2	P3	P4
Avg_Score	21.95	21.54	21.37	18.26
Avg_Rank	2.17	2.37	2.43	3

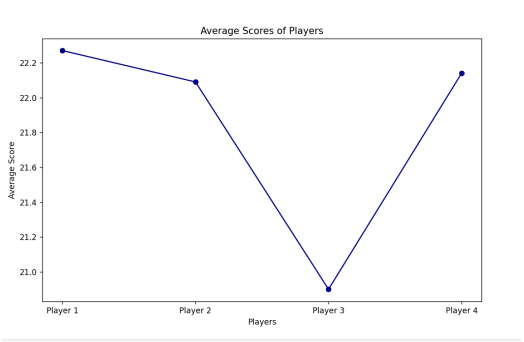
- **Observation:**

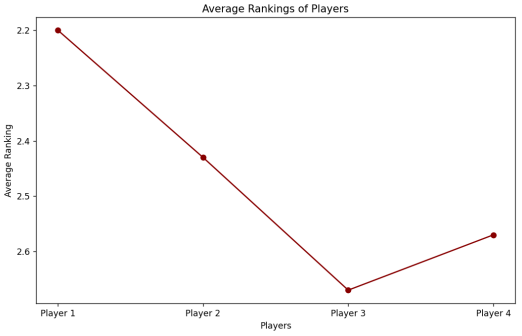
We suppose that because the fourth player has the last select order. In the game, the players in the front can choose the better DOF first. The DOFs of the positions selected by the players decrease due to the position. Thus, we can infer that the low select priority player has the worse performance due to the selection priority difference than other players.

B. Different methods to select initial position

- **Game1 : The basic form**
  (4 players, 16 sheep per player, 64 cell playing field)

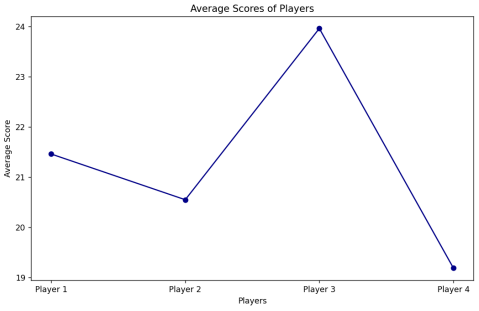
  4 players: DOF, DIS, DOF, DOF (DIS as first is skipped):

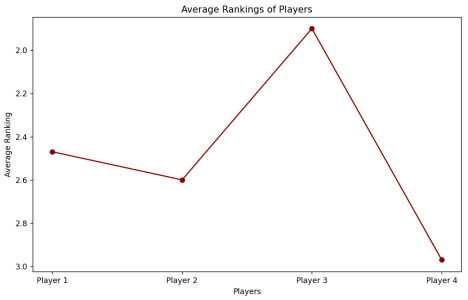




Avg_Score	22.27	22.09	20.9	22.14
Avg_Rank	2.2	2.43	2.67	2.57

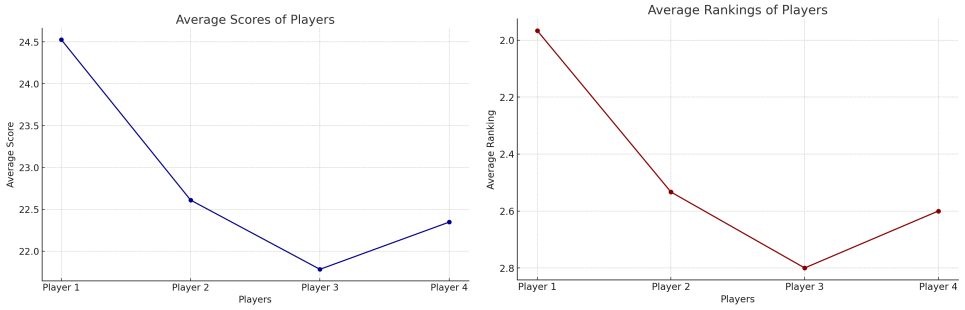
  4 players: DOF, DOF, DIS, DOF





100Samples	P1	P2	P3	P4
Avg_Score	21.46	20.55	23.96	19.19
Avg_Rank	2.47	2.6	1.9	2.97

4 players: DOF, DOF, DOF, DIS:



100Samples	P1	P2	P3	P4
Avg_Score	24.53	22.61	21.78	22.35
Avg_Rank	1.97	2.53	2.8	2.6

- **Observation**:

  In these three experiments, we placed DIS in different positions to compare with 4 DOFs and see if this new algorithm could improve the scores or rankings of the original algorithm.

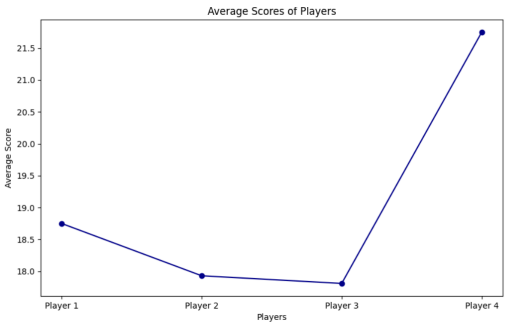
  The results show that when DIS is in the second or third position, it performs better than DOF. We believe this is because it obtains enough positional information to compensate for its disadvantages in freedom of choice. In the second position, since it only obtains the position of the first place, this indeed causes DIS to deviate from the best position for freedom. With such a trade-off, DIS achieves a performance slightly inferior to that of DOF.

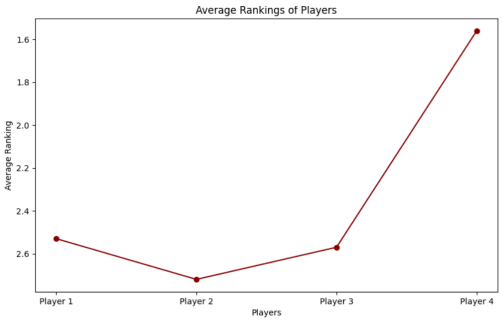
C. Impact of distance factor

- **Game1 : The basic form**

  (4 players, 16 sheep per player, 64 cell playing field)

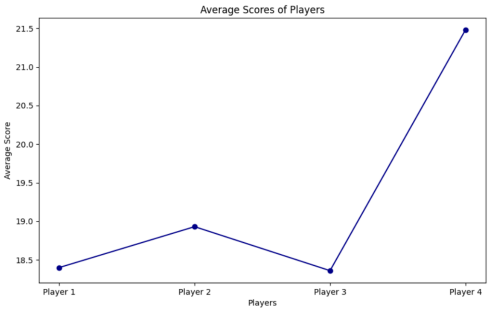
  Distance factor: 5

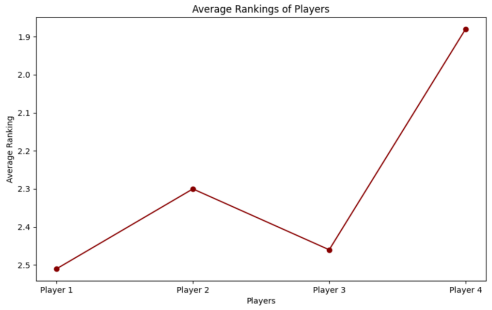




100Samples	P1	P2	P3	P4
Avg_Score	18.75	17.93	17.81	21.75
Avg_Rank	2.53	2.72	2.57	1.56

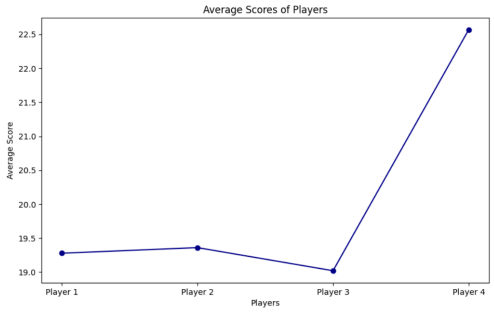
  Distance factor: 10

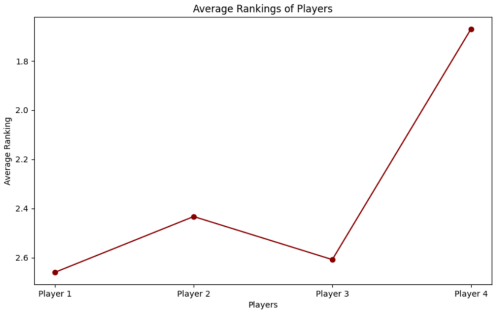




100Samples	P1	P2	P3	P4
Avg_Score	18.4	18.93	18.36	21.48
Avg_Rank	2.51	2.3	2.46	1.88

  Distance factor: 15





100Samples	P1	P2	P3	P4
Avg_Score	19.28	19.36	19.02	22.57
Avg_Rank	2.66	2.43	2.61	1.67

- **Observation**:

In our experiment, we attempted to find the optimal ratio of the weights for degrees of freedom and distance from enemies when selecting initial positions. A smaller parameter value indicates a greater emphasis on degrees of freedom, while a larger value means more emphasis on distance from enemies.

If the parameter is small, the distance from enemies essentially has no influence on the choice of initial positions. In scenarios where points of high degrees of freedom are clustered, if everyone considers degrees of freedom, their initial positions will be grouped together, which can limit the development of agents. If the parameter is large, the distance from enemies becomes the sole reference, and our agents might choose positions with limited development potential in order to avoid enemies.

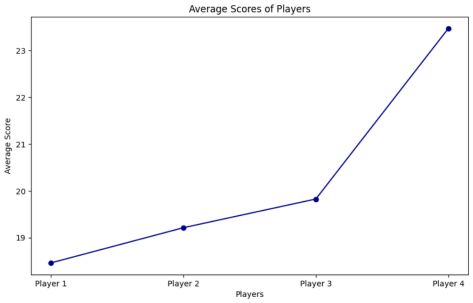
From the results, setting the distance weight to 5 appears to be a good ratio, followed by 15, and lastly 10. This part just involves adjusting the parameters to strike a balance on DOF and distances from enemies.

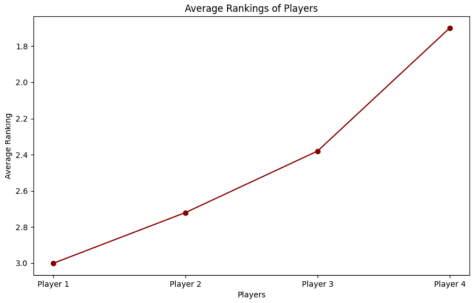
D. Iteration times

- **Game1 : The basic form**

  (4 players, 16 sheep per player, 64 cell playing field)

  Four L2_DOF players (no distance), 100 games :





100Samples	P1	P2	P3	P4
Avg_Score	18.46	19.22	19.83	23.47
Avg_Rank	3	2.72	2.38	1.7

- **Observation:**

  In this experiment, we try to find the importance of the number of iterations to our model. More iterations mean we have simulated more endgames, which allows us to predict with more complete and accurate data. When selecting nodes, we have more comprehensive data to evaluate which node to choose next. As expected, more predictions are advantageous, and indeed, the results show that 500 iterations yield the best outcomes.

IV. Thought

In the beginning, we tried to train a model with DQN to play the game, but it turned out to be a failure due to a couple reasons. First, the sparse reward question. At that time, we did not come up with more little reward steps like the distance between the wrong selection and the block we do have sheep in to help the model learn efficiently. Second, the learning rate and the

exploration rate should be higher in this situation, or the training time will be too long.

During the implementation of MCTS, questions came thick and fast. We found that the agent kept selecting the same node even though some of its children hadn't been visited, which shouldn't happen since we set the score of those to be infinity when simulating. After thoroughly checking, there was a bug in the backpropagation that we didn't trace back to the root and it starts from the leaf in every iteration. It took us lots of time to solve this.

When it comes to the MCTS experiment, We find that the state of the map will not be updated. Once the agent with the automatic code loses, the score will be calculated at that moment, which will cause the score of that agent to be higher in average. We were forced to run the AI_game.exe and record the score manually over 300 times, which almost drove us crazy.

Despite facing the obstacles above, we still think that this is a fantastic and meaningful experience to try and select different methods to create such an agent. Hope next time we can successfully train a model to achieve better performance!

v. Appendix

```
190    def mcts(root, playerID, iterations=1):
191        now_node = root
192        for iter in range(iterations):
193            while len(now_node.children):
194                now_node = select_best_child(now_node)
195            expand(now_node, playerID)
196            result, now_node = simulate_random_game(now_node, playerID)
197            now_node = backpropagate(now_node, result)
```

191: Use now_node to traverse the MCTS tree.

192~197: In every iteration, the 4 following steps will be executed.

```
135    def select_best_child(node):
136            if len(node.children)==0:
137                return None
138            total_visits = sum(child.visits for child in node.children)
139            best_child = max(node.children, key=lambda child: child.uct_score(total_visits))
140            return best_child
```

138~139 :

Here we use UCT (Upper Confidence bounds applied to Trees) as the evaluation score of the node.

```
53     def uct_score(self, total_visits, exploration_weight=1.44):
54         if self.visits == 0:
55             return float('inf')
56         average_score = self.score / self.visits
57         uct = average_score + exploration_weight * (np.log(total_visits) / self.visits) ** 0.5
58         return uct
```

54~55 :

If a child node hasn't been visited, the uct score will be very large to let the unvisited node have a chance to be picked up, which is an important part for exploration.(exploration_weight is an important parameter to balance exploitation and exploration.)

```
125  def expand(node, playerID):
126      possible_moves = node.generate_legal_moves(playerID)
127      for move in possible_moves:
128          new_mapStat, new_sheepStat = node.simulate_move(move)
129          node.children.append(MCTSNode(new_mapStat, new_sheepStat, parent=node, move=move))
```

126~129 :

Get all the possible moves from generate_legal_moves, and simulate them to get related mapstate and sheep state.

Instantiate those expanded nodes and append them to the current tree.(those children's parent will be the node we pick in selection part)

Find all the possible moves with the generate legal moves function.

```
66      def generate_legal_moves(self, playerID):
67          moves = []
68          for x in range(12):
69              for y in range(12):
70                  if self.mapStat[x][y] == playerID and self.sheepStat[x][y] > 1:
71                      for idx, (dx, dy) in enumerate(all_move_dir):
72                          nx, ny = x + dx, y + dy
73                          if 0 <= nx < 12 and 0 <= ny < 12 and self.mapStat[nx][ny] == 0:
74                              for split in range(1, int(self.sheepStat[x][y])):
75                                  moves.append(((x, y), split, idx + 1 if idx < 4 else idx + 2))
76
77          return moves
```

70:

Iterate all the map's cells, if the player occupies that cell and the player has at least two sheeps on it, that block may be the legal position we can take the action.

71~75:

For every possible position, we check its surrounding cells. If the cell is empty and in the map, we can generate all the legal moves on that cell, and those possible actions will be appended to the move list.

```
79      def simulate_move(self, move):
80          x, y = move[0][0], move[0][1]
81          dx, dy = all_move_dir[move[2] - 1 if move[2] <= 4 else move[2] - 2]
82          nx, ny = x + dx, y + dy
83          while 0 <= nx < 12 and 0 <= ny < 12 and self.mapStat[nx][ny] == 0:
84              nx += dx
85              ny += dy
86          nx -= dx
87          ny -= dy
88          new_mapStat = self.mapStat.copy()
89          new_sheepStat = self.sheepStat.copy()
90          new_mapStat[nx][ny] = new_mapStat[x][y]
91          new_sheepStat[nx][ny] += move[1]
92          new_sheepStat[x][y] -= move[1]
93          return new_mapStat, new_sheepStat
```

80~87:

We get the legal move in that direction. The while-loop in 83~87 finds the farthest legal position in that direction.

88~92:

We compute the updated game state of that move.

```
182 def simulate_random_game(node, playerID):
183     if(len(node.children)==0):
184         return calculate_score(node.mapStat, playerID), node
185     startNode = random.choice(node.children)
186     simulate_tree_root = MCTSNode(startNode.mapStat, startNode.sheepStat, parent=node, move=startNode.move)
187     while not is_game_over(simulate_tree_root.mapStat, simulate_tree_root.sheepStat, playerID):
188 
189         legal_moves = simulate_tree_root.generate_legal_moves(playerID)
190         if len(legal_moves)==0:
191             continue
192         move = random.choice(legal_moves)
193         new_mapStat, new_sheepStat = simulate_tree_root.simulate_move(move)
194         simulate_tree_root.mapStat = new_mapStat
195         simulate_tree_root.sheepStat = new_sheepStat
196         simulate_tree_root.move = move
197 
198         for i in range(1,5):
199             if(i == playerID):
200                 continue
201             legal_moves = simulate_tree_root.generate_legal_moves(i)
202             if len(legal_moves)==0:
203                 continue
204             move = random.choice(legal_moves)
205             new_mapStat, new_sheepStat = simulate_tree_root.simulate_move(move)
206             simulate_tree_root.mapStat = new_mapStat
207             simulate_tree_root.sheepStat = new_sheepStat
208 
209     return calculate_score(simulate_tree_root.mapStat, playerID), startNode
```

183:

If the node has no child, we don't need to simulate that node to the final game(because that node is the final game).

185~186:

Choose a random child to simulate, we make a new tree here to do the simulation part. The while-loop we continue the simulation until the end game, which is an iteration.

187~207:

The main part we do the simulation, we will do the simulation on that node based on everyone doing random legal moves.

```
79      def simulate_move(self, move):
80          x, y = move[0][0], move[0][1]
81          dx, dy = all_move_dir[move[2] - 1 if move[2] <= 4 else move[2] - 2]
82          nx, ny = x + dx, y + dy
83          while 0 <= nx < 12 and 0 <= ny < 12 and self.mapStat[nx][ny] == 0:
84              nx += dx
85              ny += dy
86          nx -= dx
87          ny -= dy
88          new_mapStat = self.mapStat.copy()
89          new_sheepStat = self.sheepStat.copy()
90          new_mapStat[nx][ny] = new_mapStat[x][y]
91          new_sheepStat[nx][ny] += move[1]
92          new_sheepStat[x][y] -= move[1]
93          return new_mapStat, new_sheepStat
```

```
95  def explore_region(x, y, mapStat, visited, playerID):
96      if not (0 <= x < 12 and 0 <= y < 12) or visited[x][y] or mapStat[x][y] != playerID:
97          return 0
98      visited[x][y] = True
99      size = 1
100     for dx, dy in findBarrier_dir:
101         nx, ny = x + dx, y + dy
102         size += explore_region(nx, ny, mapStat, visited, playerID)
103     return size
104
105 def calculate_score(mapStat, playerID):
106     visited = [[False] * 12 for _ in range(12)]
107     total_score = 0
108     for x in range(12):
109         for y in range(12):
110             if mapStat[x][y] == playerID and not visited[x][y]:
111                 region_size = explore_region(x, y, mapStat, visited, playerID)
112                 total_score += (region_size ** 1.25)
113     return round(total_score)
```

95~103:

We do DFS to find the area of the connected region.

105~113:

We add up all the connected region's scores to know the total score. We will use the score to update the node's uct score.

```
159     def backpropagate(node, result):
160         while node.parent is not None:
161             node.visits += 1
162             node.score += result
163             node = node.parent
164         return node
```

160~163:

Update node's information, visit times, total score, parent.

Game environment:

```
19  class GameEnvironment:
20      def __init__(self):
21          self.mapStat = None
22          self.sheepStat = None
23          self.id_package = None
24          self.playerID = None
25          self.done = None
26
27      def reset(self):
28          self.id_package, self.playerID, self.mapStat = STcpClient.GetMap()
29          init_pos = InitPos(self.mapStat)
30          STcpClient.SendInitPos(self.id_package, init_pos)
31          return self.id_package, self.playerID, self.mapStat, self.sheepStat
32
33      def step(self, action):
34          STcpClient.SendStep(self.id_package, action)
35
36      def update(self):
37          (self.done, self.id_package, self.mapStat, self.sheepStat) = STcpClient.GetBoard()
38          return
```

20~25:

__init__():records the basic game information, map state, sheep state….

27~31:

reset():actively obtains map initial information for MCTS and triggers the calculation of positions, sending the initial positions to the server.

33~34:

step():sends the best move outputted by our MCTS to the server.

36~37:

update():actively obtains the current map status for MCTS calculation each time it is our turn to move.

```
323  if __name__ == "__main__":
324      env = GameEnvironment()
325
326      env.reset()
327
328      while True:
329
330          env.update()
331          if env.done:
332              print("Game over or connection lost.")
333              break
334
335          step = GetStep(env.mapStat, env.sheepStat, env.playerID)
336
337          env.step(step)
```

'env' is an instance of GameEnvironment. 'reset' is triggered on the 'env' instance to obtain initial map information. While-loop continues until the end of the game.

At the start of each round, 'update' is triggered to actively obtain game information, which includes map data, sheep data, and a completion status (done determines whether to end the game loop).

If the game is not over, 'GetStep' is triggered to prompt MCTS to calculate the current best move, which is then sent back to the server using 'step'.

With the obtained map information, an MCTS root node is instantiated. MCTS iterations are performed using this root, and 'select_best_step' is triggered to choose the move with the highest UCT score to send as our best move.

```python
def GetStep(mapStat, sheepStat, playerID):
    root = MCTSNode(mapStat, sheepStat)
    mcts(root, playerID, iterations=500)
    best_child = select_best_step(root)
    step = [(0, 0), 0, 1] if best_child.move is None else best_child.move
    return step
```

```python
def select_best_step(node):
    if len(node.children)==0:
        return None
    total_visits = sum(child.visits for child in node.children)
    best_child = max(node.children, key=lambda child: child.uct_score_no_inf(total_visits))
    return best_child
```

```python
def uct_score_no_inf(self, total_visits, exploration_weight=1.44):
    if self.visits == 0:
        return 0
    average_score = self.score / self.visits
    uct = average_score + exploration_weight * (np.log(total_visits) / self.visits) ** 0.5
    return uct
```

Here, it's noteworthy that the UCT calculation method triggered here differs from the one we use in 'simulate_best_child'. When selecting the actual move we want to make, we aim to choose the move that currently has the highest probability to achieve the highest score(exploitation), not just the most potential(exploration). Thus, moves that have not been explored have a UCT score of 0 to prevent unexplored moves from being selected as our actual move.

Initial Position:

```python
def calculate_DOF(mapStat):
    L1_DOF_array = np.zeros((12, 12))
    L1_CDOF_array = np.zeros((12, 12))
    for i in range(12):
        for j in range(12):
            if mapStat[i][j] == 0:
                L1_DOF = 0
                L1_CDOF = 0
                for dx, dy in all_move_dir:
                    nx, ny = i + dx, j + dy
                    while 0 <= nx < 12 and 0 <= ny < 12 and mapStat[nx][ny] == 0:
                        L1_DOF += 1
                        if (dx, dy) in findBarrier_dir:
                            L1_CDOF += 1
                        nx += dx
                        ny += dy
                L1_DOF_array[i][j] = L1_DOF
                L1_CDOF_array[i][j] = L1_CDOF
```

272~279: Iterate all the blocks in 8 directions to calculate L1_DOF

```
283     L2_DOF_array = L1_DOF_array.copy()
284     L2_CDOF_array = L1_CDOF_array.copy()
285     max_L2_DOF = 0
286     max_L2_CDOF = 0
287     for i in range(12):
288         for j in range(12):
289             if mapStat[i][j] == 0 and L1_DOF_array[i][j] != 0:
290                 for dx, dy in all_move_dir:
291                     nx, ny = i + dx, j + dy
292                     while 0 <= nx < 12 and 0 <= ny < 12 and mapStat[nx][ny] == 0:
293                         L2_DOF_array[i][j] += (L1_DOF_array[nx][ny])
294                         if (dx, dy) in findBarrier_dir:
295                             L2_CDOF_array[i][j] += (L1_CDOF_array[nx][ny])
296                         nx += dx
297                         ny += dy
```

293~295: Calculate the L2_DOF with given L1_DOF

```
396     for i in range(15):
397         for j in range(15):
398             if mapStat[i][j] != 0 and mapStat[i][j] != -1 and mapStat[i][j] != playerID:
399                 enemy_choice.append((i, j))
400 
401     Max_score = 0
402     for i in range(15):
403         for j in range(15):
404             if L2_DOF_array[i][j] != 0:
405                 print(f"enemy_choice:{enemy_choice}")
406                 distance_array[i][j] = 5 * sum(np.sqrt((i - x) ** 2 + (j - y) ** 2) for x, y in enemy_choice)
407                 if any(0 <= i + dm <15 and 0 <= j + dn < 15 and mapStat[i + dm][j + dn] == -1 for dm, dn in findBarrier_dir):
408                     if distance_array[i][j] + L2_DOF_array[i][j] > Max_score:
409                         Max_score = distance_array[i][j] + L2_DOF_array[i][j]
410                         init_pos = [i, j]
411 
412     print(np.round(distance_array).T)
413     print("\n")
414     # print(L1_DOF_array)
415     print(L2_DOF_array.T)
416     print("\n")
417     print(np.round(distance_array + L2_DOF_array).T)
418     print("\n")
419     print(init_pos)
420     return init_pos
```

396~399: Mark initial positions of others

406: The evaluation including distance relationship is

dis_factor * distance + L2_DOF

409: Choose the highest one as initial position

Thanks for TAs reading! I Hope you have a nice day!