AI Capstone Project2 Report

- **Team name: Whatever**
- **Team ID: 25**
- **Team members**
  - 110550036 張家維
  - 110550014 吳權祐
  - 110550100 廖奕瑋

Game Agents Implementation (How Our Game Agents Work)

In the final version of our game agents, we utilized Monte Carlo Tree Search (MCTS) to play the games.

Game Environment

Since we had to implement several agents with different reinforcement learning algorithms, we built a common convenient environment to represent the state of the game and to make agents interact with it efficiently.

We implemented this environment as a class called `GameState` . And there are several attributes and functions in this class.

Here are some attributes to record the information of the game.

- `mapStat`: used to record current map state from the input
- `sheep`: used to record current sheep state from the input
- `scores`: used to save the scores from all players, it could be updated by the function `_calculateScore()`

```
def _calculateScore(self):
    for i in range(self.playerNum):
        id = i + 1
        connectedRegions = findConnected(self, id)
        self.scores[i] = np.sum(len(region) ** 1.25 for region in connectedRegions)
```

- `playerNum`: used to record the total number of players
- `maxSheep`: used to record the initial number of sheep
- `DIRECTION` : a tuple of all directions

```
DIRECTION = ((-1, -1), (0, -1), (1, -1), (-1, 0), (0, 0), (1, 0), (-1, 1), (0, 1), (1, 1))
```

In all functions, we used `numpy` to accelerate them. There were three important functions we implemented.

- `getLegalMove(self, id) -> list`

In this function, it iterated through all positions in the map and sheep state $M_{(r,c)}$ and $S_{(r,c)}$, where r stood for row, c for column. If $M_{(r,c)} = id$ & $S_{(r,c)} > 1$ and there was space on any d in `DIRECTION` , we put three legal moves $[(r,c),d,1],\ [(r,c),d,\frac{S_{(r,c)}}{2}],\ [(r,c),d,S_{(r,c)}-1]$ in the list.

```
def getLegalMoves(self, id):
    legalMoves = []
    for row, col in np.ndindex(self.mapStat.shape):
        # Select cells with more than one sheep
        if self.mapStat[row, col] != id or self.sheep[row, col] <= 1: continue
        for dir_i, dir in enumerate(DIRECTION):
            if dir_i == 4: continue
            if 0 <= row + dir[0] < len(self.mapStat) and \
                0 <= col + dir[1] < len(self.mapStat[0]) and \
                self.mapStat[row+dir[0], col+dir[1]] == 0:
                possible_moves = set([1, int(self.sheep[row, col] // 2), int(self.sheep[row, col]) - 1])
                legalMoves.extend([((row, col), m, dir_i + 1) for m in possible_moves])
    return legalMoves
```

- `getNextState(self, move, id) -> GameState`

  It applied the move of the specific player and return the new game state.

```
def getNextState(self, move, id):
    newState = copy.deepcopy(self)
    pos, split, dir_i = move
    row, col = pos
    if self.mapStat[row, col] != id or self.sheep[row, col] < split:
        raise("State error")
    dir = DIRECTION[dir_i - 1]
    newState.sheep[row, col] -= split
    end = False
    while not end:
        if 0 <= row + dir[0] < len(self.mapStat) and \
            0 <= col + dir[1] < len(self.mapStat[0]) and \
            newState.mapStat[row + dir[0], col + dir[1]] == 0:
            row += dir[0]
            col += dir[1]
        else:
            end = True
    if newState.sheep[row, col] != 0 or newState.mapStat[row, col] != 0:
        raise("Move error")
    newState.sheep[row, col] = split
    newState.mapStat[row, col] = id

    return newState
```

- `evaluation(self, id) -> float`

  The function calculated the evaluated score of the player from different aspects. Details could be checked in Evaluation Score subsection.

```
def evaluate(self, id):
    self._calculateScore()
    rank = self.getRank(id)
    sheeps = self.sheep[self.mapStat == id]
    score_part = self.scores[id - 1] / (self.maxSheep ** 1.25) # 16 ^ 1.25 = 32
    rank_part = 1 - rank / 4 # prefer higher rank
    sheeps_part = (-np.max(sheeps) - 1) / (self.maxSheep - 1) # avoid sheep to be too concentrated
    moves_part = len(self.getLegalMoves(id)) / (np.count_nonzero(self.sheep[self.mapStat == id] - 1) * 8 + 1) # prefer more legal moves
    rewards = np.asarray([score_part, rank_part, sheeps_part, moves_part])
    rewards /= np.linalg.norm(rewards)
    return np.dot(rewards, np.ones(4) / 4)
```

Algorithm Implementation

- **Init Position**

  For initial positioning, we introduced a novel method utilizing 2d convolution that could determine the initial position efficiently. This was implemented in the `weightedMap` method.

```python
def weightedMap(mapStat, kernel=(5, 5), weights=None, filling=-1):
    kw, kh = kernel
    weights = np.ones(kernel) if weights is None else weights
    assert weights.shape == kernel
    mapStat = np.array(mapStat)
    mapStat = np.pad(mapStat, ((kw // 2, kw // 2), (kh // 2, kh // 2)), "constant", constant_values=filling)
    mapStat = np.abs(mapStat)
    sub_matrices = np.lib.stride_tricks.sliding_window_view(mapStat, kernel)
    return np.einsum("ij,klij->kl", weights, sub_matrices)
```

  The idea behind this was that if we treat every grid as a number, e.g., 0 for space, 1 for obstacle, we then could use 2d convolution to compute the "crowdedness" with respect to its surrounding neighbors. Since opponents' sheep could split into multiple grid of sheep in most time, we treated them as a scalar to be computed in convolution.

  After computed the values, the agent would choose the grid with minimum number of sheep, and has at least one obstacle in its four neighbors as its starting position. This way, no matter which order would our agent be, it can avoid to be too closed to other players and obstacles while choosing a legal position.

```python
def InitPos(mapStat):
    init_pos = [0, 0]
    """
    Write your code here
    """
    mapStat = np.array(mapStat)
    available = mapStat == 0
    neighbors = weightedMap(mapStat, kernel=(3, 3), weights=np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]]), filling=0)
    available[neighbors == 0] = False
    weighted_map = weightedMap(mapStat)
    weighted_map[~available] = np.inf
    init_pos = np.unravel_index(np.argmin(weighted_map), mapStat.shape)
    print(weighted_map)
    return init_pos
```

- **MCTS Algorithm**

  For implementing MCTS, we used an open source python script as our base code. Since the script only provided abstract classes and methods, we wrote derived classes to inherit them on our own.

  Most of our code lay in the derived `SheepGame` class. We implemented each abstract function in the `SheepGame` class, such as `find_children` and `is_terminal`. It was notable that we used another `gameOver` method to determine whether the game ended in `is_terminal` method but not by checking whether each player had moves. This was because the ego player might get the highest score at the time it couldn't move any further, but not the highest score until the end of the game.

  - **Simulation**

    For the consideration of performance, we removed `simulate` method in the base `MCTS` class. If we simulated till the end of the game in every rollout stage, it would took too much time

to do simulations and performance of the agent would be very poor. So we removed this function to have a higher number of iterations. However, this caused another problem. That is, we could not know who were going to win in the end and set this as reward after doing rollout. So we used the function `evaluation` to estimate the reward of current state to solve this issue.

```
def do_rollout(self, node):
    "Make the tree one layer better. (Train for one iteration.)"
    path = self._select(node)
    leaf = path[-1]
    self._expand(leaf)
    self._backpropagate(path)
```

- **Backpropagation**

  Because it was egocentric in the first three games, we defined the reward values Q in MCTS with respect to the reward of the final node R and the reward of the ego player R_E during backpropagation:

$$Q = \begin{cases} R(R_E) & \text{, if current player is ego player} \\ R_E - R & \text{, otherwise} \end{cases}$$

```
def _backpropagate(self, path):
    "Send the reward back up to the ancestors of the leaf"
    ego_reward = path[-1].state.evaluate(path[-1].ego_player)
    reward = path[-1].reward()
    for node in reversed(path):
        self.N[node] += 1
        self.Q[node] += reward if node.player_id == node.ego_player else ego_reward - reward
```

- **Select**

  Given a node v, and v_i are the children nodes of v. And $N(v)$ means visit times of the v and $Q(v)$ means the reward of v. To select the child node to do rollout with balance of reward and visit times, we picked the node v_i with highest UCT score which calculated from the formula $UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{log(N(v))}{N(v_i)}}$.

```
def _uct_select(self, node):
    "Select a child of node, balancing exploration & exploitation"

    # All children of node should already be expanded:
    assert all(n in self.children for n in self.children[node])

    log_N_vertex = math.log(self.N[node])

    def uct(n):
        "Upper confidence bound for trees"
        return self.Q[n] / self.N[n] + self.exploration_weight * math.sqrt(
            log_N_vertex / self.N[n]
        )

    return max(self.children[node], key=uct)
```

- **Expansion**

In expansion stage, we utilized `getLegalMove` and `getNextState` to expand the children nodes from the current node, and the player id of children would be the next one of parent.

```
def _expand(self, node):
    "Update the `children` dict with the children of `node`"
    if node in self.children:
        return  # already expanded
    self.children[node] = node.find_children()
```

```
def find_children(self):
    next_player_id = self.player_id % self.player_num + 1
    children = [SheepGame(self.state.getNextState(move, self.player_id), next_player_id, self.ego_player, move)
                for move in self.state.getLegalMoves(self.player_id)]
    random.shuffle(children)
    return children
```

Evaluation Score

Our evaluation score in the final version had four individual parts:

- **Score**

  $$\text{Score} = \frac{s}{s_{\max}}$$

  - s: the score of current player
  - $s_{\max}$: the maximum score, which were `maxSheep ** 1.25`

  ```
  score_part = self.scores[id - 1] / (self.maxSheep ** 1.25)
  ```

- **Rank**

  $$\text{Rank} = 1 - \frac{r}{r_{\max}}$$

  - r: the rank of current player
  - $r_{\max}$: the maximum rank, namely 4.

  ```
  rank = self.getRank(id)
  rank_part = 1 - rank / 4
  ```

- **Sheep**

  Attained by dividing the maximum number of current sheep in the cells minus 1 with the maximum number of all sheep minus 1, namely 15, which was negated afterward. By doing so, an agent was discouraged to have sheep too crowded.

  Here is the formula:

  $$\text{Sheep} = -\frac{\max(S_{(r,c)})}{S_{\max}}$$

  - $S_{(r,c)}$: the values of sheep at (r, c).
  - $S_{\max}$: the maximum value of sheep, namely 16.

  ```
  sheeps = self.sheep[self.mapStat == id]
  sheeps_part = (-np.max(sheeps) - 1) / (self.maxSheep - 1)
  ```

- **Moves**

$$\text{Moves} = \frac{\text{number of legal moves}}{\text{number of occupied grids}}$$

```
moves_part = len(self.getLegalMoves(id)) / (np.count_nonzero(self.sheep[self.mapStat == id] - 1) * 8 + 1)
```

Each part was normalized to the range of $[0, 1]$, and concatenated as a reward vector. The reward vector was normalized to an unit vector first. Its mean was then computed as the return value of the `evaluate` function.

```
rewards = np.asarray([score_part, rank_part, sheeps_part, moves_part])
rewards /= np.linalg.norm(rewards)
return np.dot(rewards, np.ones(4) / 4)
```

Other Designs

Since game 4 is a cooperative game, we slightly modified the Q values in MCTS with respect to the reward of the final node R and the reward of the ego player R_E during backpropagation to fit the condition:

$$Q = \begin{cases} R & \text{, if current player in ego player's team} \\ R_E - R & \text{, otherwise} \end{cases}$$

Experiments and Experiences

Algorithm Choices

- **Experiments Result**

  All experiments were done in game 1.

Algorithms	Random Action	Minimax	MCTS (classical)	Q-learning	MCTS (ours)
Result ($\frac{win}{total}$)	3 / 10	2 / 10	4 / 10	4 / 10	**7 / 10**

- **Random Action**

  Choosing the move randomly from the list of legal moves was the easiest method. We have tried this method and observed its performance. The agent with random action was uncontrollable, sometimes it could win the game perfectly, while at most time it got very low score. After seeing the result from this experiments, we realized that this Sheep Game is very uncontrollable, there are too many factors in the game we need to consider, so we started studying how to use reinforcement learning algorithms to build a powerful agent.

- **Minimax**

  Concept of minimax algorithm is to maximize the evaluation score when egocentric agent moves and minimize the score in other players' round. In our experiments, minimax got the worst performance, even lower than the random action agent. This is because of the following reason:

- **Limited depth of the tree**

  Since it needed to consider all situations from current state, the agent was going to simulate all moves from all players, this limited the depths of minimax tree and made the performance worse. This is why we don't use it, but the concept of evaluation score were applied to our MCTS agent.

- **MCTS (classical)**

  The classical MCTS algorithm took lots of time to do rollout and simulate till the end of the game to get the final state as reward, then did backpropagation to the node on the path. We had tried this method to build our agent. But we found out that this algorithm could not get better performance in the result, here were the reason we guessed:

  - **High computation cost of simulation**

    The computation cost of rollout stage was too high to make the agent having enough iterations to sample all possible moves. This made the agent to be uncontrollable since the performance of it was all depended on the quality of the sampled moves, which was highly random.

  Based on this result, we construct our agent with concepts of MCTS and minimax which didn't simulate till the end of the game, and took evaluation scores as reward.

- **Q-learning**

  Our Q-learning experiment was constructed with a Q-table containing Q-values in the format [row, column, moving direction], which was a $12 \times 12 \times 8$ matrix, and Bellman equation was used as the value function. But the experiments with Q-learning often showed undesirable results, which might result from the following reasons:

  - **Limited learning time**

    The learning time available for each move was merely 3 seconds. This limitation prevented us from making a larger Q-table that could provide more information for each state, which might enhance strategy selection for each move.

  - **Sub-optimal Q-table design**

    The design of our Q-table could be sub-optimal. The fact that learning time was strictly limited lead to smaller Q-table, which couldn't contain enough information unless a better or denser state representation format was used. However, since MCTS and other tree-shaped algorithms already performed far better than Q-learning, this part was deprecated.

Design of evaluation score

How to design the evaluation score is an important things in our algorithm, since we need to use this to estimate the reward of current state. We have tried lots of metrices as a part of score and decided the final version of our evaluation score by observing how these sub-scores affect the agent's behavior.

- **Current score**

  Obtained by first calculating the current score earned by an agent at the moment. This was a very important part to make the agent get higher score.

- **Score ranking**

This sub-score was related to the rank of the player in the game, which encouraged the agent to have a higher rank.

- **Crowding of sheep**

  Attained by dividing the maximum number of current sheep in the cells minus 1 with the maximum number of all sheep minus 1, which was negated afterward. By doing so, an agent was discouraged to have sheep too crowded.

- **Number of legal moves**

  Since more legal moves meant more possible ways to expand, the number of legal moves of an agent was chosen as an evaluation method of a state.

- **Number of neighbors**

  This part would make the agent to divide the sheep very dispersion, so we decided not to use it.

- **Statistics of connected sheep distribution**

  We have tried to use variation and mean as a part of the score, but this was hard to normalize and it made the agent out of control.

We eventually use the first four sub-scores as the evaluation score in our implementation.

Similarities and Differences in the Different Game Settings

Game 1

The agent in game 1 (also does those in game 2-4) used the evaluation score introduced right in the previous section. And then it used the below algorithm, which was specified before to evaluate its backpropagation:

$$Q = \begin{cases} R(R_E) & \text{, if current player is ego player} \\ R_E - R & \text{, otherwise} \end{cases}$$

Game 2

Since game 2 could be seen as the expanded version of game 1, the agent in game 2 utilized the same algorithm as game 1.

Game 3

Since we couldn't find a good settings for the agent in game 3, we used the same agent as game 1 and game 2 used. In this agent, since we didn't know the values inside of each grid, we assumed that our opponents split their sheep equally. That is, the distribution of their sheep is uniform, and every position could split as long as the number of grid occupied by any opponent was less than 8.

Game 4

The agent in game 4 used the algorithm similar to game 1, but the evaluation of backpropagation is slightly different, as shown in the following formula:

$$Q = \begin{cases} R & \text{, if current player in ego player's team} \\ R_E - R & \text{, otherwise} \end{cases}$$

Contributions of Individual Team Members

- 張家維
  - Implementation and experiments of Q-learning and MCTS(classical)
  - Design of evaluation score
  - Agent4 implementation
  - Report writing
- 吳權祐
  - Basic GameState implementation
  - Implementation and experiments of minimax and random action agent
  - Design of evaluation score
  - Agent3 implementation
  - Report writing
- 廖奕瑋
  - Basic MCTS algorithm implementation
  - Design of evaluation score
  - Agent1&2 implementation
  - Experiments of our agent
  - Report writing