Mancala

1 Program Design

As a team, we choose **Python** as the programming language of our project to solve this Mancala (Kalah version) game problem.

1.1 State-Space Representations

In this project, we use several data types in Python to represent states and actions in Mancala game.

1.1.1 State representation

We use *list* data type to describe the board state and show the number of stones in each pit and store. Notice that this list is a combination of four sub-lists: two pit lists and two store lists. Figure1.1 shows the representation of the initial board state. The first 7 elements in this list represent the board state on the Player1's side and the last 7 elements represent the board state on the Player2's side.

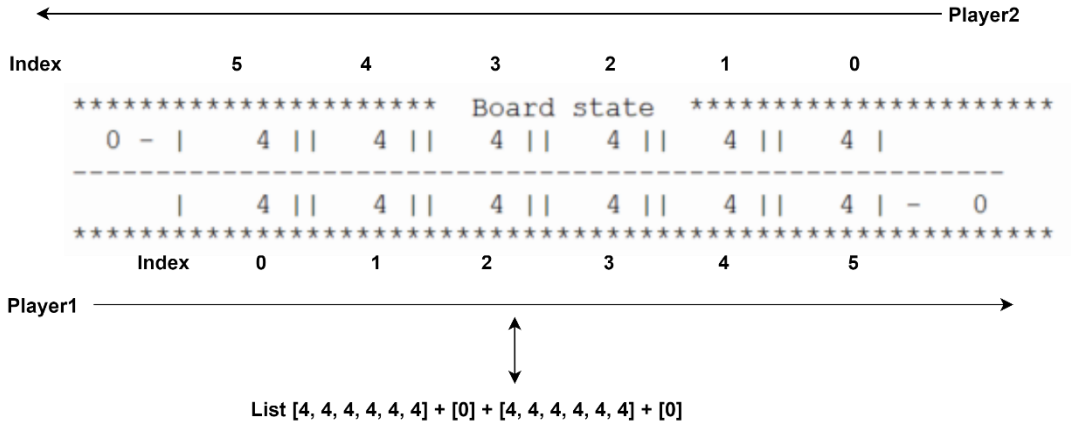

Figure1.1 Representation of the initial board state

According to the game rules, the final board state is that there is no stone in the pits on either side of players. In out project, the function of `check_end_game` in the `Mancala` class is used to check if the game reaches the final state.

1.1.2 Action representation

We also use *list* data type to represent actions in the Mancala game. Each element in this list represents the index of a pit on the active player's side. Each time one player moves, the player pick up all the stones in a pit and distribute them in a counterclockwise direction to the next pit, so a move is *int* data type indicating the index of a pit that the active player selects. For each player, the value of a move is from 0 to 5 as is shown in Figure1.1.

What is more, the action list should be filtered each turn to obtain all the indexes of the **non-empty** pits before the active player chooses a move. In out project, the function of `filter_actions` in the `Mancala` class is used to remove the indexes of the empty pits.

1.1.3 Transition model

The transition model in this Mancala game is to take a state, distribute stones and toggle

which player is next to play.

(1) Get a move for four kinds of players
There are four kinds of players in this game: Random Player, Human Player, Minimax Player and Alphabeta Minimax Player, so there are four ways to take a state and get a move. In our project, Player1 is always the first one to take a move and the function of `get_move` is used to get a legal move from these four kinds of players.
For Random Player, we use `choice` function in Python to choose a random move from the `filtered_actions` list. For Human Player, we use the `ask_human_move` to get an input or an index of a pit on Human Player's side, and then use the `check_illegal_move` function to check if the move or the input is an index of the non-empty pits on the player's side.
For Minimax Player and Alphabeta Minimax Player, we use the function `max_value` and `min_value` in the *Player* class to find the min or max value and return the value and the corresponding move. We set a flag called `ab_flag` to turn on or turn off the alpha-beta pruning.

(2) Sowing step
In our project, the `sowing` function in the `Mancala` class represents distributing stones and updating the board state after the active player chooses a legal move. The critical point in this function is to find the index of the pit where the last stone landed on (`last_pit`) and the number of complete counterclockwise circles of the sowing step (`circle`).

(3) Turn taking step
After the sowing step, the algorithm needs to check who is the next one to take a move, which is the `turn_taking` function in our project.
According to the game rules, the active player can move again **if and only if** the last stone of a move landed on the player's store (`if last_pit == self.numPit`).
Otherwise, it is time for the opponent to take a move.
Besides, if the last stone of a move landed on an empty pit on the active player's side (`if last_pit < self.numPit and self.p_pits(player.index)[last_pit] == 1`) and there are some stones in the opposite pit, then the stones in these two pits will be captured in the active player's store.

1.2 Classes and Functions
There are two classes in our project: `Player` class and `Mancala` class. The Player class includes several properties and methods of two players in the game. In this class, we label the Player1 as `index`1 and the Player2 as `index` 2. We design methods to simulate four kinds of players in the player class including minimax algorithm, alpha-beta pruning, and heuristic functions. The Mancala class includes several properties and methods of the board state. In this class, we use `numPit` to represent the number od pits on each side (the default value is 6) and use `stonesInPit` to represent the initial number of stones in a pit (the default value is 4). Functions like `filter_actions`, `check_end_game`, `check_illegal_move`, `sowing` and `turn_taking` are included in this class.

1.3 Maximum depth for Minimax
Considering the time complexity, we set a maximum depth for minimax algorithm and

alpha-beta pruning algorithm to limit the response time in a few seconds. After experiments, we set a default maximum depth value for minmax as 6 and alpha-beta pruning one as 8 to react in a similar time. To make the algorithm more powerful, the maximum depth can be larger, which may cause longer response time in minutes or hours.

2 Utility Functions

As how to determine the score of the board state can be a heuristic problem, we design three utility functions in the `score` function in the `Player` class for this project.

First, if we just want to win, only who wins matters. Or we consider that it is a big win when one player wins the game with more stones. Therefore, the number of stones should be considered. Moreover, the depth of minimax algorithm also matters if we want to win the game with less steps.

Also, before the game is over, whether it is objective to evaluate the state only with the number of stones in the store is equally worth discussing, as the number of stones in the pits will have an influence on whether the game will end and other situations. In this project, we choose one of three heuristic functions with parameter `h_flag`.

First, we decide the player with more stones in the store wins.

When we only consider who wins (e.g., set `h_flag` as 0):

$$\begin{aligned} \text{score} &= \quad 50 \quad \text{when win} \\ \text{score} &= -50 \quad \text{when lose} \\ \text{score} &= \quad 0 \quad \text{when draw} \end{aligned}$$

When we want to get a big win with more stones in the player's store:

$$\begin{aligned} \text{score} &= \quad |\text{difference of stones in stores}| \quad \text{when win} \\ \text{score} &= -|\text{difference of stones in stores}| \quad \text{when lose} \\ \text{score} &= \quad 0 \quad \text{when draw} \end{aligned}$$

When we want to get a big win with less steps:

$$\begin{aligned} \text{score} &= \quad 50 - \text{depth} \quad \text{when win} \\ \text{score} &= \quad \text{depth} - 50 \quad \text{when lose} \\ \text{score} &= \quad 0 \quad \text{when draw} \end{aligned}$$

When both the number of stones and steps are considered:

$$\begin{aligned} \text{score} &= \quad |\text{difference of stones in stores}| - \text{depth} \quad \text{when win} \\ \text{score} &= \quad \text{depth} - |\text{difference of stones in stores}| \quad \text{when lose} \\ \text{score} &= \quad 0 \quad \text{when draw} \end{aligned}$$

Furthermore, when assuming that both the number of stones in the stores and in the pits matters, it becomes a tricky question of finding the winner before the game ends. A few experiments and evaluation are taken but the effect is still unclear. Maybe the minimax algorithm needs to change into another algorithm.

3 Experiments and Evaluation

We evaluate the performance of our algorithms through a series of four experiments: Minimax Player vs. Random Player, Human Player vs. Random Player, Random Player vs. Minimax Player, Alphabeta Minimax Player vs. Minimax Player. There are three results in the Mancala game: Player1 wins, Player2 wins or draw. We create the *p1_VS_p2* function to perform the experiments. Table4.1 shows the results of experiments.

Table4.1 Results of experiments and evaluation

Players		Game results		
Player1	Player2	Player1 wins (times)	Player2 wins (times)	Draw (times)
Minimax	Random	99	1	0
Human	Random	10	0	0
Human	Minimax	1	8	1
Minimax (*maximum_depth = 6*)	Alphabeta Minimax *(maximum_depth = 6*)	100	0	0
Alphabeta Minimax *(maximum_depth = 6*)	Minimax (*maximum_depth = 6*)	100	0	0
Minimax (*maximum_depth = 6*)	Alphabeta Minimax *(maximum_depth = 8*)	0	100	0

As shown in the comparison between minimax and alpha-beta minmax in the same depth, Player1, who plays first, always wins because of the same algorithm and same choice. In order to make a difference, a random exploration can be added. And as the alpha-beta minimax is more time-efficient than minmax, with the same response time, it can go deeper. And with deeper exploitation (depth), alpha-beta minimax can beat minmax easily.