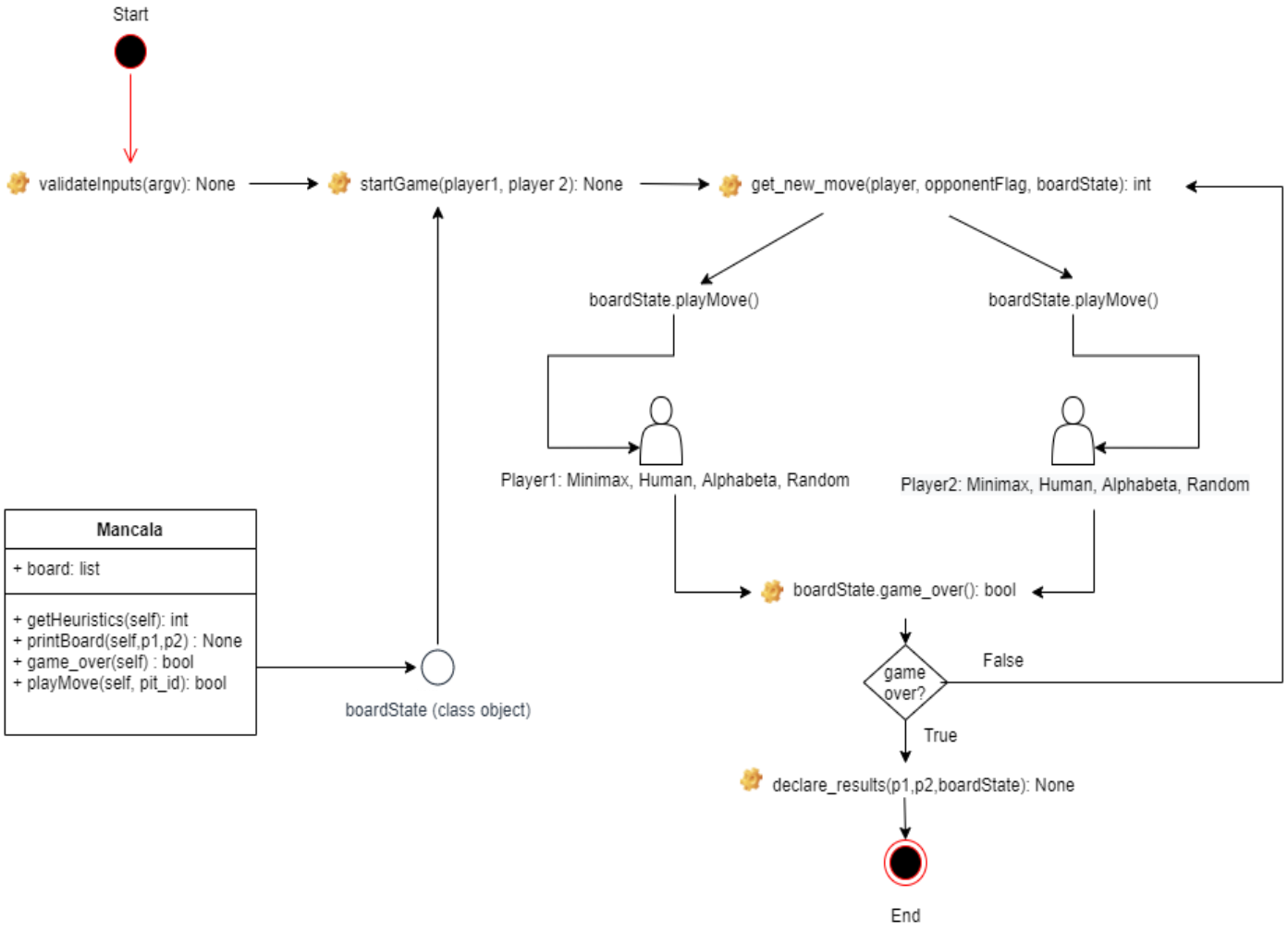
CSC442 - Introduction to Artificial Intelligence

Project 1: Mancala

Submitted by Jayant Rohra and Ajinkya Deshmukh

Program Design and Implementation



State Space

State Representation: The Mancala board is represented as a list of length 14, where indices 0 to 5 correspond to the pits of player 1, index 6 corresponds to the store of player 1, indices 7 to 12 correspond to the pits of player 2 and index 13 corresponds to the store of player 2.

Actions: The action of the agent/player involves the selection of a non-empty pit from 1 to 6. The number of stones from the selected pit are then distributed one by one until they are empty towards the pits in the right-hand side of the agent. The agent can put the stones in the opponent's pit except for the opponent's store. The agent gets an extra chance if the final stone lands in its own store.

The agent can capture the opponent's stones and add it to its store if it lands in an empty pit that is in the agent's own domain and of course if the opponent pit's stones are non-empty.

Goal: The goal of the player is to have the maximum stones in its pits along with its own store to win the game.

State Transitions: The get_player_move() and playMove() methods handle legal state transitions for all agents. The implementations of each of these methods is displayed in the write up below.

Game Play

Board Display: The 1st row of the board displays the store and the pits for player 2 and the 2nd row of the board displays the pits followed by the store of player 1. The pits are numbered 1 to 6 for the reference of the user. Below is a sample display of the board:

```
PS C:\Users\jayan\Documents\mancala_jayantrohra> python playMancala.py random minimax

Welcome to the Mancala Game!

pits :            6   5   4   3   2   1
                  -----------------------
minimax     --> 0 | 4 | 4 | 4 | 4 | 4 | 4
                  -----------------------
                  | 4 | 4 | 4 | 4 | 4 | 4     0 <--  random
                  -----------------------
pits :            1   2   3   4   5   6
```

Invoking the Game: The program is invoked with the command *python playMancala.py player1 player2* where player1 and player2 must be chosen from a valid list of input arguments: ['random', 'minimax', 'human','alphabeta']. If two same players are chosen, for example: *python playMancala.py human human*, then the game will automatically rename these players to human_1 and human_2.

```
PS C:\Users\jayan\Documents\mancala_jayantrohra> python playMancala.py human human

Welcome to the Mancala Game!

pits :            6   5   4   3   2   1
                  -----------------------
human_2     --> 0 | 4 | 4 | 4 | 4 | 4 | 4
                  -----------------------
                  | 4 | 4 | 4 | 4 | 4 | 4     0 <--  human_1
                  -----------------------
pits :            1   2   3   4   5   6
-----------------------------------------
```

Illegal Moves: The player/agent always selects a nonempty pit from its own valid range of pits. If the human player selects an illegal pit number or an empty pit, the program prompts the user with the message "*Illegal Move! Pick a valid pit:*". When the random player is taking its turn, it loops through a random choice from 1 to 6 until it fetches a non-empty pit.

Heuristics: We selected the difference between the scores of the players as our heuristics. We did this because, for the agent to win, it must essentially maximize the number of stones in its store. And the path it takes should basically be in the direction of maximizing its score – which means the higher the difference in scores for the maximizing player, the larger would be the heuristic.

Turn Taking: The game always starts off with player 1 taking the 1st turn. If the last stone lands in either of the player's own store, the respective player gets a chance to pick a move again, otherwise the opponent gets its turn and this continues until the game terminates(either of the player has all its pits empty)

Finishing the Game: When either of the user empties their pits (i.e. they do not have any further moves to play), the game comes to a halt, the final points are tallied by taking a sum of all the leftover stones in the non-empty pits and the scores of both the players are displayed. The outcome (winning player name/draw) is also printed to the output.

```
minimax  played move: 6
pits :            6   5   4   3   2   1
                  ------------------------
minimax     --> 40  0|  0|  0|  0|  0|  0
                  ------------------------
                  0|  0|  0|  0|  1|  0     7 <--  random
                  ------------------------
pits :            1   2   3   4   5   6
-----------------------------------------


random  score:  8

minimax  score:  40

minimax Wins!
```

Implementation

Global Constants:

N = 6 (Number of pits for each player)
DEPTH = 6 (Cut off depth for minimax and alphabeta)
STONES = 4 (Number of stones in each pit)
P1_STORE = N (Index of the player 1's store)
P2_STORE = (2*N) + 1 (Index of the player 2's store)

Class Mancala:

- *Parameterized constructor*: If the board input parameter is non-empty, it initializes the board with the input otherwise initializes a fresh board with 4 stones in each pit and the respective player stores as 0.
- *printBoard(self, player1, player2)*: Prints the current state of the board with the player stores

- *game_over(self)*: Returns true if either of the player has all of its pits empty else returns false
- *playMove(self, pit_id)*: Executes the move for the respective player and pit id and alters the state of the board. Handles the logic of capturing opponents stone if the current player lands among its own pit which is empty pit. Returns the boolean variable move_again as True, if the current player lands in its own store, else returns False.
- *getHeuristics(self)*: Returns the heuristic value as the difference between the the player's stores

Methods:

- *def isValidMove(obj, pit_id, opponentFlag):* Returns True is the player/agent is playing a valid and non-empty pit, otherwise returns False.
- *def declare_results(obj, p1, p2):* Calculates and Prints the final player scores and declares the game outcome (Draw/Winner)
- *def get_next_move(player, opponentFlag, boardState):* Returns the pit id for the move to be executed by the respective player
- *def startGame(player1, player2):* Initiates the game by creating a new state of the board and alternates the turns between the players by invoking the get_next_move and playMove functions
- *def validateInputs(argv):* Checks the validity of the input arguments passed to the script.
- *def minimax(boardState, depth, maximizingPlayer):* Executes the Minimax algorithm recursively and returns the heuristic value along with the most optimal move for the given cutoff depth
- *def alphabeta(boardState, depth, alpha, beta, maximizingPlayer):* Executes the Alpha beta pruning algorithm recursively, where it prunes the search space when alpha >= beta and returns the heuristic value along with the most optimal move for the given cutoff depth

Member Contributions

Member	Contribution
Jayant Rohra	• General Project Design • Game start and state transitions • Random • Minimax • Report
Ajinkya Deshmukh	• General Project Design • Player Move Execution • Human • Alpha Beta Pruning • Report

Experiments

Player 1	Player 2	DEPTH	# of Games	Player 1 Win Rate	Player 2 Win Rate	Draw Rate
Minimax	Random	6	100	99%	0%	1%
Minimax	Random	8	100	100%	0%	0%
Random	Minimax	6	100	0%	99%	1%
Random	Minimax	8	100	0%	100%	0%
Minimax	Alphabeta	6	10	100%	0%	0%
Alphabeta	Minimax	6	10	100%	0%	0%
Human	Minimax	6	10	10%	80%	10%
Minimax	Human	6	10	90%	10%	0%
Random	Alphabeta	10	100	100%	0%	0%

Multiple experiments were conducted to test the effectiveness of the alphbeta and minimax algorithms. In case of Minimax/Alphabeta VS Minimax/ Alphabeta, the player 1 always won the game 100% of the times with a score of 33-15 for depth = 6, 25-23 with a depth of 8 and 29-19 for a depth of 10. For depth = 12, player 2 won the game with a score of 23(P1) - 25(P2). The alphabeta algorithm runs smoothly until a depth of 10 and starts lagging from depth = 12 whereas the minimax starts slowing down after depth = 8. This proves that the alphabeta algorithm is successfully pruning the search space. Also, both the algorithms use the same strategy hence every combination of alphabeta/minimax vs alphabeta/minimax for the same depth yield the same results.

Skill of Minimax/Alphabeta V/S Random:

Minimax won ~100% of the time against the random move selector. Minimax with a depth of 6, gave 1 draw match out of 100 games played against the random player, whereas minimax with a depth of 8, gave a 100% win rate. We also allowed random to play first to see if random ends up selecting an optimal first choice and defeats minimax, but that did not happen at least in the sample of 100 games that we tested.

Discussion

- This project gave us an opportunity to build AI for a real-world scenario. While building the game, the first thing we learnt was that mapping the state space from a real-world problem to a data structure is not always easy. Initially, we selected a simple state space representation (list of tuples), but that made us realize that it makes the state transitions a bit complicated to code and deal with. Hence, we switched to a much convenient representation of the state space using a simple 1D array.
- Debugging minimax was the most complicated challenge. It printed so many possible states in one recursion. It was too overwhelming to even understand what's going on. However, we were able to get over this by recursing over a temporary copy of the board's state by copying the class object and searching in the direction in which our heuristic took us.
- Overall, we were able to get the game working and implement all the required functionality for the project.