MANCALA PROJECT WRITEUP

By: *Mohammed Ibrahim Khan*

Program Design

The game of Kalah comprises of two players, a board of 2 x 6 (6 pits for each player), one store for each player and 4 stones in each of the 12 pits. The design considerations are actions in which stones have to be distributed in anti-clockwise directions, toggling of players and ensuring the possibility of a player's turn being repeated and determining the terminal state. The numbering of done in anti-clockwise manner which is tricky to implement in code due to differences in how data structures are indexed. Deciding which variables and data structures to keep global is also crucial. Selecting moves for all types of players can be done in a single function by taking player type as an argument. During game play, it must be ensured that it's the correct player's turn since keeping it as a global variable can result in it getting modifies during minimax move selection. Same goes for objects of board and store. In python, recursion depth is limited, so it might need to be increased before calling the minimax function. The design of the board in the output screen is also important and numbering of pits must be taken care of.

The key components in the design are:

- Number of pits (N_PITS): It is a global constant representing the number of pits on each player's side that is set as 6 in the code.
- Number of stones (N_STONES): It is a global constant representing the number of stones in each pit. It is set as 4 in the code.
- The board (board): A standard board of 2 rows, one for each player and number of pits in each row specified as the global constant. In the code, a 2-dimensional Python list of 2 x N_PITS is initialized as the board with value as N_STONES in each cell.
- The stores (store): Each player is assigned a store where the stones dropped would count towards that players final score. A list of 2 integer values, one for each player is initialized as the store.
- The players: Since it is a two player game, two integer values have been designated for the players. 0 is chosen for the first player and 1 is chosen for the second player.
- The current player (PLAYER): A global integer is maintained throughout the execution of the code that stores the current player number.
- The winner (WINNER): A global integer designating the winning player, it is updated every time the game satisfies the terminal condition.
- Player type: A command line argument for representing the type of each player – random, human, minimax and alphabeta

The key functions of the game are:

- Gameplay(player1, player2, board, store): Used for simulating the entire game by selecting moves from the players by repeatedly selecting moves, making moves, checking for terminal and toggling player until terminal condition is true.
- Print_board(board, store): For printing the current state of the board.
- Select_move(player_type, board, store): For selecting the moves according to player type – random, human, minimax or alphabeta.
- Minimax(player, maxPlayer, board, store, depth): For computing the depth limited minimax score of a player given a game state.
- Alphabeta(player, maxPlayer, alpha, beta, board, store, depth): For computing the depth limited alphabeta score of a player given a game state.
- Utility(player, store): For determining the utility at the terminal game state for a player.
- Heuristic(player, board, store): For determining the expected utility at an intermediate state for a player.
- Move(pit, board, store): For making a move by a player and updating the state of the game.
- Toggle_player(): For changing turns between players.
- Game_ends(): For evaluating whether the game has reached the terminal state and determining the result.

State space representation of the game is as follows:

- **State:** N_PITS, N_STONES, board, store, PLAYER
- **Actions:** select_move(), minimax(), alphabeta()
- **Transition model:** move(), toggle_player()
- **Initial state:** board = [[4, 4, 4, 4, 4, 4], [4, 4, 4, 4, 4, 4]], store = [0, 0], PLAYER = 0
- **Terminal state:** [[0, 0, 0, 0, 0, 0], [-, -, -, -, -, -]] or [[-, -, -, -, -, -], [0, 0, 0, 0, 0, 0]]

I used depth limited approach for minimax and alphabeta to decrease the time taken to select a move and hence reduce the running time of the game. The utility function of the result for a player (in case the game ended before given depth) is the difference between the number of stones in the player's store and the number of stones in the opponent's store. The heuristic after the given depth is difference between the sum of the number of stones in the store and the number of stones in all the pits on the player's side for both the players.

$$Heuristic\ (P1) = \left(Store(P1) + \sum_{i=1}^{6} Board(P1, i) \right) - \left(Store(P2) + \sum_{i-1}^{6} Board(P2, i) \right)$$

Member Contributions

This is project was done alone:

Member: Ibrahim

Contribution: Everything that's in the project

Experiments

A. Investigating skill of minimax and alphabeta

Minimax vs Random

Serial Number	Player1	Player2	Winner
1	Random	Minimax	Minimax
2	Minimax	Random	Minimax
3	Random	Minimax	Minimax

We can see that minimax defeated random players regardless of whether it was the first player or not.

Alphabeta vs Random

Serial Number	Player1	Player2	Winner
1	Alphabeta	Random	Alphabeta
2	Alphabeta	Random	Alphabeta
3	Random	Alphabeta	Random
4	Random	Alphabeta	Alphabeta
5	Random	Alphabeta	Alphabeta
6	Random	Alphabeta	Alphabeta
7	Alphabeta	Random	Alphabeta
8	Alphabeta	Random	Alphabeta
9	Alphabeta	Random	Alphabeta
10	Random	Alphabeta	Alphabeta

From the above table, it is clear that this implementation of depth limited heuristic based alphabeta is optimal or at least very close to optimal. It won 90% of times against a random player. I observed the moves for the one time it lost. In that case, random was the first player and randomly chose a move considered to be the strongest first move (pit 3). Since this game has a first mover advantage, it's fair to assume that alphabeta is optimal.

Minimax vs Alphabeta

Serial Number	Player1	Player2	Winner
1	Alphabeta	Minimax	Alphabeta
2	Alphabeta	Minimax	Alphabeta
3	Minimax	Alphabeta	Minimax

The above results show that the first player wins (probably due to first mover bias) and thus, there is no significant difference in the results. Both are equally optimal. It's surprising that there are no draws. That maybe because the probability of a draw in a game like this is very low.

B. Computational effectiveness of alphabeta over minimax

Minimax vs Random (after reducing the depth from 10 to 9)

Serial Number	Player1	Player2	Winner	Time (seconds)
1	Minimax	Random	Minimax	694.311666
2	Random	Minimax	Minimax	1006.142332

Alphabeta vs Random

Serial Number	Player1	Player2	Winner	Time (seconds)
1	Alphabeta	Random	Alphabeta	28.047025
2	Alphabeta	Random	Alphabeta	25.123754
3	Alphabeta	Random	Alphabeta	29.854388
4	Random	Alphabeta	Alphabeta	42.761208
5	Random	Alphabeta	Alphabeta	22.098403
6	Random	Alphabeta	Alphabeta	41.089955

On an average, using Alphabeta yields a speedup of 30x (even accounting for decrease in minimax depth) without any other improvement and gives essentially the same result.

C. Number of states expanded and effective branching factor

Minimax:

Number of states expanded for choosing one move = **82,860,431**

The depth is taken to be 10 after the initial enumeration, so total depth = 11

If we assume *b* to be the branching factor, $b^{11} = 82860431$

From here, we can compute effective branching factor, ***b* = 5.246**

That means all possibilities of moves at every step have been expanded, except for the ones where number of stones in a pit is zero. It's very close to the maximum number of 6.

Alphabeta:
Number of states expanded for choosing one move = **402,781**
The depth is taken to be 10 after the initial enumeration, so total depth = 11
If we assume *b* to be the branching factor, $b^{11} = 402781$
From here, we can compute effective branching factor, ***b*** **= 3.232**
This is a massive improvement over minimax and hence results in a big decrease in execution time.

D. First mover advantage

This game has a significant first mover advantage. As seen in the minimax vs alphabeta case in the A part of the experiment, both are optimal players but first player has won most of the time. In alphabeta vs random in part B, there is a lone case of first mover random defeating alphabeta because it coincidentally chose optimal moves in the starting. In particular, the first playing moving pit 3 and then pit 6 drastically increases its chances of winning.

Discussion

There is no guarantee that the results obtained above represent an absolute optimal performance. For starters, we are not enumerating the whole game tree but instead searching to a limited depth in a trade-off for time. But yeah, it is at least almost optimal. The only way to confirm full optimality is have these algorithms play against world champions. The question of perfect optimality depends on the heuristic we have chosen. Maybe a better, closer heuristic can be devised. But, for all intents and purposes, the performance of these algorithms is commendable and reducing the depth is worth the enormous savings in time and getting essentially the same performance. When assigned as first player, these algorithms always choose pit 3 as first move and that is actually considered the best first move. This observation shows that they are in fact making optimal moves.