# Assignment 4
## Solving the Mancala game with adversarial search

Mancala is a sowing game, also known as Awalé, of African origin. Mancala is played on a board that has a certain number of small pits, usually 6 on each side, and a large pit, called a store, at each end. Each of the small pits contains a certain number of seeds, typically 4. A visualization of the Mancala board is illustrated in Figure 1.

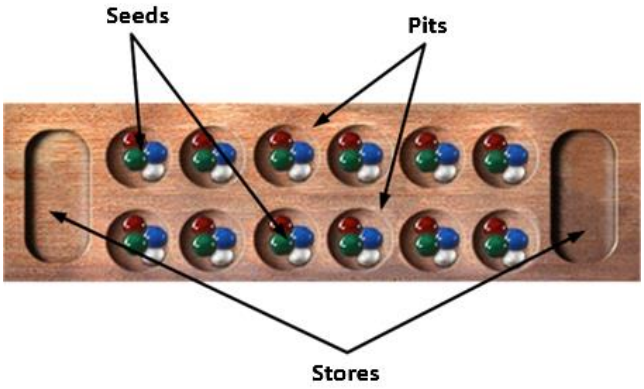


*Figure 1 Mancala Board*

1. **Game Setup:** The organization of the Mancala game is illustrated in Figure 2.
   - The players sit facing each other with the Mancala board between them. The six pits on each player's side belong to them;
   - Each player places 4 seeds in each of their 6 pits;
   - Each player's store is located to their right. The stores are initially empty.

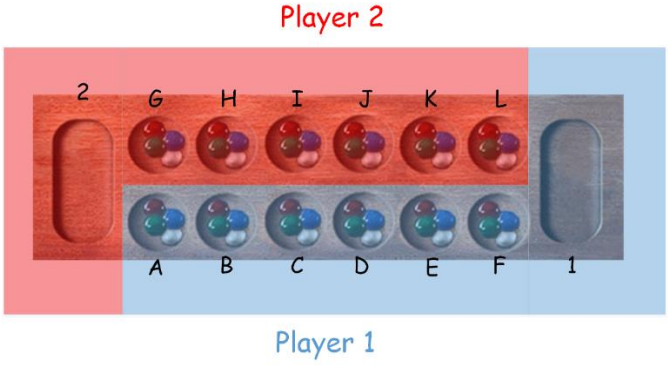


*Figure 2 Setup of the Mancala Game*

2. **Objective of the Game:** The goal of the game is to capture more seeds than your opponent.
3. **Game Rules:** The rules of this game are very simple:
   - The first player selects a pit on their side of the board and collects all the seeds from it;
   - Moving counterclockwise, the player drops one seed into each pit until they have no more seeds in hand;
   - The player can place a seed in any pit on the board (including their own store), except in the opponent's store;
   - If the last seed placed lands in an empty pit on the player's side, that seed and all the seeds in the opposite pit (belonging to the opponent) go to the player and are placed in their store;
   - When a player has no more seeds in all the pits on their side, the game ends, and the opposing player can collect all remaining seeds and place them in their own store;
   - The player with the most seeds in their store wins.

- Propose a formulation for the Mancala game and implement it in Python.
- Solve the game using the Minimax Alpha Beta pruning algorithm.

1. **Modeling a State:** To model the state, you need to create a class **MancalaBoard** where you define the following attributes and functions:
   - The game is represented as a board, which can be modeled using a dictionary. The keys represent the indices of the 12 pits and 2 stores, and the values represent the number of seeds in each. The pits of Player 1 are labeled with letters from A to F, and the pits of Player 2 are labeled with letters from G to L. The stores for players 1 and 2 are represented by corresponding numbers.
   - Optionally, add two tuples: one to store the indices (letters) of Player 1's pits and another for Player 2's pits.
   - Optionally, add two dictionaries: one to store the opposite pit for each pit, and another to store the next pit in sequence.
   - Define the function **possibleMoves(player)** that returns the possible moves (i.e., the indices of the pits belonging to the player that contain seeds).
   - Define the function **doMove(player, pit)** that executes a move.

2. **Modeling the Game:** To model this game, you need to create the following two classes:

**a) The Game Class**

This class represents a node in the search tree. Within this class, you will define:

- An attribute **state** to represent the game state (i.e., an instance of the MancalaBoard class).
- An attribute **playerSide** which is a dictionary that stores the player side chosen by the human and the computer (player1 or player2).
- The function **gameOver()**, which checks if the game has ended (i.e., all the pits of one player are empty). This function will also collect all remaining seeds in the opponent's pits and place them in their store.
- The function **findWinner()**, which returns the winner of the game along with their score.
- The function **evaluate()**, which estimates the value of a node *n* in the search tree using the Minimax algorithm with Alpha-Beta Pruning. This function assigns a value to the node based on the following equation (Equation 1):

$$value(n) = nbSeedsStore(playerSide[COMPUTER]) - nbSeedsStore(playerSide[HUMAN]) \quad (1)$$

**b) The Play Class**

In this class, you will define:

- **The humanTurn function:** This function allows the user to take their turn.
- **The computerTurn function:** This function enables the computer to take its turn. The computer's move is determined using a game search algorithm.
- **The MinimaxAlphaBetaPruning search function:**
    - In simple games like Fan-Tan and Tic-Tac-Toe, the computer can fully develop the search tree and find the move that leads to victory. However, games like Chess and Mancala have too many moves and variations, making it unrealistic for a computer to explore all possible options. This is why it is necessary to specify a maximum tree depth in search algorithms and to use the best heuristic as an evaluation function for a node.
    - Typically, **COMPUTER = MAX** = **1** and **HUMAN = MIN = -1**.
    - The pseudo-code for the **MinimaxAlphaBetaPruning** search function is provided as follows:

```
# game est une instance de la classe Game et player = MAX or MIN
def MinimaxAlphaBetaPruning(game, player, depth, alpha, beta):
  if game.gameOver() or depth == 1:
    bestValue = game.evaluate()
    return bestValue, None
  if player == MAX:
     bestValue = -inf
     for pit in game.state.possibleMoves(game.playerSide[player]):
          child_game = copy(game)
          child_game.state.doMove(game.playerSide[player], pit)
          value, _ = MinimaxAlphaBetaPruning(child_game, -player, depth-1, alpha, beta)
          if value > bestValue:
              bestValue = value
              bestPit = pit
          if bestValue >= beta:
              break
          if bestValue > alpha:
              alpha = bestValue
  else:
     bestValue = +inf
     for pit in game.state.possibleMoves(game.playerSide[player]):
          child_game = copy(game)
          child_game.state.doMove(game.playerSide[player], pit)
          value, _ = MinimaxAlphaBetaPruning(child_game, -player, depth-1, alpha, beta)
          if value < bestValue:
              bestValue = value
              bestPit = pit
          if bestValue <= alpha:
              break
          if bestValue < beta:
              beta = bestValue
  return bestValue, bestPit
```