Ministry of Defense

9th of December, 2024

PROJECT MANCALA

In remembrance of the flame that scarred me..

Mathis Fituwi

Initial Mancala Implementation

Staying back on track, this is the write-up to my Mancala game w/ MinMax and AlphaBeta. Let's talk about my implementation on Mancala. Playing this game one too many times against my girlfriend, not having certain key features like the ‘capture’ mechanic or randomized pit made the implementation much easier. To start, I created my Mancala game such as

Pseudocode:

```
class Mancala
    Mancala() → constructor for variables
    display_board() → shows game state
    valid_move() → grabs valid move of some player p
    random_move_gen_p2 () →  a random move generator for player p2
    winning_eval () → checks if game is over, shows final score
    print_final_game_results() → prints final result after game over
    get_winner() → get method I just needed to use rq
    play() → plays the game
    executePlay() → like the helper function to play()
```

Since this game was given to us in HW7 as a semi-template, I’ll skip things like the constructor and display_board method were already given to us whilst the rest needed to be completed on our own (I will come back to some things that were added to the Mancala constructor). The two I want to skip are valid_move() and random_move_gen_p2(), I feel as though the implementation is pretty self explanatory. Now getting into the actual juice, most of the implementation was in play() and executePlay(). We would start, at least in the example below, passing in moves to mancala.play()

```
    while not mancala.is_terminal():
        mancala.display_board()
        # Both players random
        move = random_player(mancala, mancala)

        print(f"Player {mancala.current_player} moves: {move}")
        mancala.play(move)
```

There in play we check a few things, one if the game is over or not. After, we check who the current player is as that move is associated with them, then we check if that move is valid for that said player, and finally we execute the actual play, below will be a small snippet of the code for one player, but the same logic applies to 2 as well.

```
if not self.is_terminal():
            if self.current_player == 1:
                if pit is None or self.valid_move(pit) == False:
                    return f"Invalid Move by Player {self.current_player}"

                self.executePlay(pit)
                self.moves.append((1, pit))
```

And I forgot to mention the moves list where we append stuff into it. This is done to **cap out** the amount of moves/set some number of plies as the project required.

As for how executePlay() occurs, the basic rundown is now we grab the pit index in respect to who the player is, save the stones and reset to it being an empty pit, and now iterating through the board while stones still remain. As we go through the board we all one to all the pits beside the opposite player's mancala pit. The code will be below:

```
if self.current_player == 1:
            index = pit - 1
        if self.current_player == 2:
            index = self.p2_pits_index[0] + pit - 1


        # Save and clear current index stones
        stones = self.board[index]
        self.board[index] = 0

        next_index = index
        while stones > 0:
            # Iterate index
            next_index = (next_index + 1) % len(self.board)
            # Skip opposite Mancala
            if (self.current_player == 1 and next_index ==
self.p2_mancala_index) or (self.current_player == 2 and next_index ==
self.p1_mancala_index):
                continue
            # Reduce stone and increase 1 on whatever pit
            self.board[next_index] += 1
            stones -= 1
```

And that is generally it for the most part with the initial implementation.

Mancala Expanded / MinMax / AlphaBeta

Here is where the star of the show appears. I had to take this game and use AI algorithms as intended. We were tasked with implementing first a random player (which was essentially done) and both MinMax/AlphaBeta. However, at the time I have made Mancala really ridge to this type of change. So I had to do a few tweaking. Below is the updated structure of my game:

Pseudocode:

```
class Game:
        is_terminal(state) → check if state is terminal
        actions(state) → available actions at said state
        result(state, move) → results of doing a move in some said state
        utility(state, player) → returning utility in respect to the current player of
game when state is terminal

class Mancala extends out of Game
        Mancala() → constructor for variables
        display_board() → shows game state
        valid_move() → grabs valid move of some player p
        random_move_gen_p2 () →  a random move generator for player p2
Change →        is_terminal () → checks if game is over, shows final score
        print_final_game_results() → prints final result after game over
        get_winner() → get method I just needed to use rq
        play() → plays the game
        executePlay() → like the helper function to play()

        'Game Methods'
        is_terminal(state) → override
        actions(state) → override
        result(state, move) → override
        utility(state, player) → override
```

So having this Game superclass where Mancala inherits from made it easier for me to understand how I can get all the necessary functions of the game without impacting the actual game. And from the last assignment I mean winning_eval() is basically checking terminal states so I changed the name for consistency. To quickly go over the super class methods that Mancala has used, is_terminal again just the same as winning_eval(). The actions() method looks at the current player at some state, where state is a copy of the current Mancala game stored in another Mancala object, and in its actions result the available actions in respect to who is the current player at that current state. The result() method is grabbing a copy of that game and progressing the game like executePlay() (now thinking about it I probably should've decoupled the executePlay() method result and that is basically the same). Finally the utility() spits out a utility number, that is how favorable this state is, in respect to the current player. And I decided here that the best thing you can do in Mancala is to make moves where you will keep as many stones on your side as possible. Therefore, this function just simply added the current player stones on their side/mancala pit and subtracted it to their opposing player. The code snippet will be below:

```
player1_stones = state.board[state.p1_mancala_index] + \
            sum(state.board[i] for i in range(state.p1_pits_index[0],
state.p1_pits_index[1] + 1))

        player2_stones = state.board[state.p2_mancala_index] + \
            sum(state.board[i] for i in range(state.p2_pits_index[0],
state.p2_pits_index[1] + 1))

        # Return utility from the perspective of the specified player
        return player1_stones - player2_stones if player == 1 else
player2_stones - player1_stones
```

Getting into the MinMax and AlphaBeta implementation, I reckoned it would be easier to have it such that it's just a simple function that takes in states and explores it outside of the Game. So:

```
def minimax_player(game, state):
    """ MiniMax player using alpha-beta search """
    return minimax_search(game, state)[1]

def alphabeta_player(game, state):
    """ MiniMax player using alpha-beta search """
    return alphabeta_search(game, state)[1]
```

And this was made possible thanks to us having Mancala extend out of the Game superclass. Diving into the MinMax function, it follows the same procedure as I learned in lecture. It follows such as starting at some player turn, that player will go through a set of actions and try to yield its max path by assuming that the player that comes next chooses the worst action for them. Same way going down one step of this recursion game tree, the following player will attempt to minimize the previous player's utility by looking through their available actions exploring to find the worst action. However, now that the first player comes back it will attempt to maximize where it is at. This back and forth of minimizing and maximizing is where MinMax comes from. Where each recursive iteration does the **copy** of the game state move forward. We want it to be a copy and not the actual game because it is just analysing its next best move.

```
def minimax_search(game, state):
    """ MinMax Search """

    player = state.current_player
    infinity = np.inf

    def max_value(state, alpha, beta):
        # If game is over return the utilty, breaks recursion
        if game.is_terminal(state):
            return game.utility(state, player), None

        v, move = -infinity, None

        # Looking at all actions at said state
        for a in game.actions(state):
            # For each actions find it's min val recursively
            v2, _ = min_value(game.result(state, a), alpha, beta)
            # Set new alpha
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)

        return v, move

    def min_value(state, alpha, beta):
        # If game is over return the utilty, breaks recursion
        if game.is_terminal(state):
            return game.utility(state, player), None

        v, move = +infinity, None
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a), alpha, beta)

            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)

        return v, move

    # Start off w/ finding max outcome for P1
    return max_value(state, -infinity, +infinity)
```

Ignoring the alpha and beta here I actually implemented AlphaBeta first and realized I needed MinMax so I just took out the pruning stage. This implementation does the same thing as any MinMax would.

Now to talk about AlphaBeta, I knew how the implementation worked BUT I was mistaken on one thing. Even though I knew how to do it, in my head the pruning stage, the stage where we choose not to explore every action but rather ignore, still yields the same result. But actually the simple case it that given that it is the turn of the Min who's **beta**, the variable that keeps track of what minimizes Max's utility, the instance such that once it is Max's turn and Max finds a point where utility is greater than the current beta, we immediately prune the rest. The logic behind this, in this case, is that there is no need to explore any further actions when we already know that path won't minimize Max's turn. Thus we 'prune', otherwise ignore, the rest of actions. And this same idea applies to Max's turn who's **alpha** keeps track of highest yield paths utility and pruning anything that falls under. The implementation follows as below:

```
def alphabeta_search(game, state):
    """ Alpha-Beta Pruning Search """

    player = state.current_player
    infinity = np.inf

    def max_value(state, alpha, beta):
        # If game is over return the utility, breaks recursion
        if game.is_terminal(state):
            return game.utility(state, player), None

        v, move = -infinity, None

        # Looking at all actions at said state
        for a in game.actions(state):
            # For each actions find it's min val recursively
            v2, _ = min_value(game.result(state, a), alpha, beta)
            # Set new alpha
            if v2 > v:
                v, move = v2, a
                alpha = max(alpha, v)
            # Pruning
            if v >= beta:
                return v, move
        return v, move
    def min_value(state, alpha, beta):
        # If game is over return the utility, breaks recursion
        if game.is_terminal(state):
            return game.utility(state, player), None

        v, move = +infinity, None
        for a in game.actions(state):
            v2, _ = max_value(game.result(state, a), alpha, beta)

            if v2 < v:
                v, move = v2, a
                beta = min(beta, v)
            # Pruning
            if v <= alpha:
                return v, move
        return v, move

    # Start off w/ finding max outcome for P1
    return max_value(state, -infinity, +infinity)
```

With the same structure as MinMax but this key part

```
if v <= alpha:
                return v, move
```

```
if v >= beta:
                return v, move
```

That will do the immediate return effectively pruning the game tree.

Results

I talked a little about the results on the project notebook but it wouldn't hurt to say here plus some. The results of testing with Random vs Random, MinMax vs Random, and AlphaBeta vs Random is pretty neat. When I was performing 100 iterations of Random vs Random, surprisingly enough, the first Random, Random 1, would win more often than what random would usually entail. And with some research it turns out it's actually favorable to go first in the game because you get immediate board control. Now I do not know for sure if this is correlated by any means but a nice food for thought I had. With MinMax vs Random, I mean MinMax wiped. Even with different plies I mean the results were generally the same. However, it took long. Doing 100 iterations it averaged ~1 second per game. Which makes sense as MinMax explores the whole game tree. However, talking about AlphaBeta, that took literally ~0.01 seconds per game. It was so much faster that I thought I did something wrong for a split second. With the results being very much the same. One thing I would like to talk about is the plies.

Through my testing I couldn't seem to find a significant difference between them. However I think the case may be how I result in the game/conclude the winner. As the game follows, if the game ends via just naturally one side has no more stones or is interrupted (plies) the remaining stones are added to their respective side. Because I want to assume that MinMax and AlphaBeta, optimally, play **really** defensively such that they are frugal in how they get stones to the other side of the opponents, regardless of our plies because of how the game is structured they still win. Though something that I will think about is that at one point, much later into the game probably, do both algorithms go through that 'attacking' phase. If I happen to find out the current plies to have such that I terminate the game around that time would we see the surprising result of them losing rather more (unless I incorporate the plies feature into the utility method in some way).

With that, I conclude my project. Thanks for your time reading my write-up!