CPSC 4610/5610 Artificial Intelligence
Homework 1: Search

Introduction to Mancala

Mancala is a two-player game from Africa in which players moves stones around a board, trying to capture as many pieces as possible. In the board above, player 1 owns the bottom row of stones and player 2 owns the top row. There are also two special pits on the board, called Mancalas, in which each player accumulates his or her captured stones (player 1's Mancala is on the right and player 2's Mancala is on the left).

There are a number of variants on Mancala, but here are the rules that we will play by: The board starts out with four pieces in each of the non-Mancala pits. To take a turn, a player chooses on of the pits on his or her side of the board (not the Mancala) and removes all of the stones from that pit. The player then places one stone in each pit, moving counterclockwise around the board, starting with the pit immediately next to the chosen pit, including the player's Mancala but **NOT** the opponent's Mancala, until the player has run out of stones.

If the player's last stone ends in his or her own Mancala, the player gets another turn. If the player's last stone ends in an empty pit on his or her own side, the player "captures" all of the stones in the pit directly across the board from where the last stone was placed (the opponents stones are removed from the pit and placed in the player's Mancala) **as well as the last stone placed** (the one placed in the empty pit). The game ends when one player cannot move on his or her turn (i.e. there are no stone's left on the player's side), at which time the other player captures all of the stones remaining on his or her side of the board.

For more details on Mancala and its variants visit:
`http://en.wikipedia.org/wiki/Mancala`

Python Code

I have included some initial code to get you started. This code includes support for the basic game as well as a simple AI player (it's REALLY simple). The starter code contains the following files:

MancalaBoard.py: A file that contains a class that represents the Mancala gameboard. This class manages the gameboard, knows how to add moves, can return legal moves, can determine when a player has won, etc.

- Player.py: A player class that can be instantiated with several types ("constants" defined at the beginning of the class):
  – HUMAN: a human player (i.e. prompt the user for a move)
  – RANDOM: a player that makes random legal moves
  – MINIMAX: a player that uses the minimax algorithm and the score function to choose its next move, limited by a specified ply depth
  – ABPRUNE: a player that uses alpha-beta pruned search and the score function to choose its next move, limited by a specified ply depth. This player is not yet supported (you will implement it).
  – CUSTOM: the best player you can create. This player is not yet supported (you will implement it).
  Notice that this file also contains a class MancalaGuru which inherits from Player and in which you will also fill out the details.

- MancalaGUI.py: A simple GUI for playing the Mancala game. To invoke the game you call the startGame(p1, p2) function, passing it two player objects.

- TicTacToe.py: A file that contains a class representing a Tic Tac Toe gameboard.

To run a GUI game between two humans:

```
# load the GUI class and associated functions
>>> from MancalaGUI import *
>>> player1 = Player(1, Player.HUMAN)
>>> player2 = Player(2, Player.HUMAN)
>>> startGame(player1, player2)
```

Notice that the minimax algorithm we discussed in class has already been implemented. *You can run the MancalaGUI.py and play against a 5-ply minimax player that has a really simple scoring function. To understand the game, I suggest beating this AI repeatedly.* :-)

Problem 1: Scoring function

The board scoring function in the basic player is too simple to be useful in Mancala?the agent never looks ahead to see the end of the game until it?s too late to do anything about it. Your first task is to write an improved score function in the MancalaPlayer class, a subclass of Player, that scores the quality of the current board. You may wish to consider the number of pieces your player currently has in its Mancala, the number of blank spaces on its side of the board, the total number of pieces on its side of the board, specific board configurations that often lead to large gains, or anything else you can think of. You should experiment with a number of different heuristics, and you should systematically test these heuristics to determine which work best. Note that you can test these heuristics with the MINIMAX player, or you can wait until you?ve completed alpha-beta pruning. In addition to your code, you will submit a short (1-2 paragraphs) writeup about how you chose your final score function. What did you try along the way? What worked well and how did you determine what works well?

Problem 2: $\alpha - \beta$ Pruning

The next part of the assignment is to implement the alpha-beta pruning search algorithm described in class. Look in the code to see where to implement this function (in the Player class).

In your alpha-beta pruning algorithm, you do NOT have to take into account that players get extra turns when they land in their own Mancalas with their last stones. You can assume that a player simply gets one move per turn and ignore the fact that this is not always true. Notice that my provided version of minimax also makes this simplifying assumption. This makes the scoring function slightly inaccurate, but easier to code.

You will likely want to test your alpha-beta pruning algorithm on something simpler than Mancala, which is why we have provided the Tic Tac Toe class. Using alpha-beta pruning, it?s possible for an agent to play a perfect game of Tic Tac Toe (by setting ply=9) in a reasonable amount of time. The first move by the agent may take a few seconds depending on the computer, but after that the agent will choose its moves quickly. Contrast this timing to minimax which will take 5-10 seconds to make it?s first move.

Test your algorithm carefully by working through the utility values for various board configurations and making sure your algorithm is not only choosing the correct move, but pruning the tree appropriately. You must submit along with your code at least one example that illustrates that your algorithm correctly prunes the search tree. For example, consider the board:

```
XX_
_O_
___
```

Your search will first try an O in the top right corner and should find that that move leads ultimately to a tie. Now consider this point in the search tree:

```
XX_
OO_
___
```

where O has tried a move in the left, center spot. On the next level (when X plays in the top right), the algorithm immediately sees that X will win, resulting in a score of 0 for O. Since X is the min player, X will choose this move unless there is a move with an even LOWER value (which there is not). Thus, the best O can do is a score of 0 if it moves in the middle left. It does not need to try the other positions for X because it knows that it is not going to choose to play here (because blocking X in the top row leads to a score of 50, which is better). Thus the algorithm will prune the rest of the search tree at this point after it has tried the top right corner for X.

To write this part up, you must first illustrate that your program behaves correctly in this case by adding print statements to your code (**that you must remove before submitting**), which might then produce the following output:

```
alpha is 50.0, score is 0.0. Aborted on move 2 in minValue on
XX2
OO5
678
```

You should then explain what is going on as above. **You should choose a different example when testing your code.** Come see me if you have questions about this part. Be sure that you understand the above explanation and that you can produce one of your own.

Problem 3: Custom Player

Create a custom player (using any technique you wish) that plays the best game of Mancala possible. This will be the player that you enter into the class tournament. (*You may decide to just use the alpha beta pruning and scoring function you wrote in the previous parts, additional improvements are optional!*)

- Your player must compile without errors within 5 seconds.
- Your player must make its moves in 10 seconds or less on a "reasonable" machine (you don't need to get fancy with timers or anything, but if it runs significantly longer than that, it will be disqualified from the tournament).
- Choose a name for your player. Rename both the MancalaGuru subclass and the Player.py file to exactly match your player's name. (This is so they can be easily identified in the tournament). For example, I might name my player "DavesPlayer". So, my file (which would be called "DavesPlayer.py") would contain a class called Player and a class called DavesPlayer.
- I will not specify a ply for your tournament player. It is your (your player's) responsibility to use the ply that makes it return a move within 10 seconds. What I mean by this is, I?ll instantiate your player in this way for the tournament:

  ```
  DavesPlayer.DavesPlayer(1, DavesPlayer.Player.CUSTOM)
  ```

  In other words, I will not initialize it with a ply and you should either have a default ply, or have the player determine on its own what ply it can get to in each move.
- Your player may NOT use a database.
- Your player may NOT connect remotely to another machine.
- Your player may NOT spawn any other processes or threads. The player must use a single thread.
- Any pre-computed moves can be hard coded, but not written to or loaded from a file or database.

Hints

For alpha-beta pruning, you likely will need equivalent minValue and maxValue functions for your pruning approach, for example minValueAB and maxValueAB.

Notice that minValue, maxValue and minimaxMove return both a score and a state. When you call a function that returns two values, make sure that you either assign the result to two values OR, if you save it as a single value, it will be a tuple and you'll have to select the appropriate value. (Technically, we only need the score in minValue and maxValue, however, it was included here for debugging purposes).

If you want to start Tic Tac Toe from a particular state, comment out self.reset() in the hostGame method. Then, create a new board and two new players. Use makeMove to make the appropriate moves to change the board configuration. When the board state is where you'd like it to be, you can then call hostGame and since the reset call is commented out, it will start from that state.

Writeup

1. How you chose your final score function. What did you try along the way? What worked well and how did you determine what works well?
2. Alpha-Beta pruning correctness example.
3. Commentary on what you did or tried to do with the Custom Player.

Submission

- YourPlayer.py: (i.e. the appropriately renamed "Player.py" file) This file should contain all the code you have written, including your score function, your alpha beta pruning algorithm and your custom player.
- A pdf file containing your writeup.

Optional

If you find yourself incredibly interested in this assignment and want to do a little extra, read on...

As described above, the current minimax implementation does not take into account the fact that a player gets another move if his or her last stone ends in the player?s own Mancala. Write a new minimax function, called “minimaxFull” and a new alpha-beta pruning function, called “abPruneFull”, that takes into account that a player gets another move on their turn if they land in their own Mancala with their last stone.