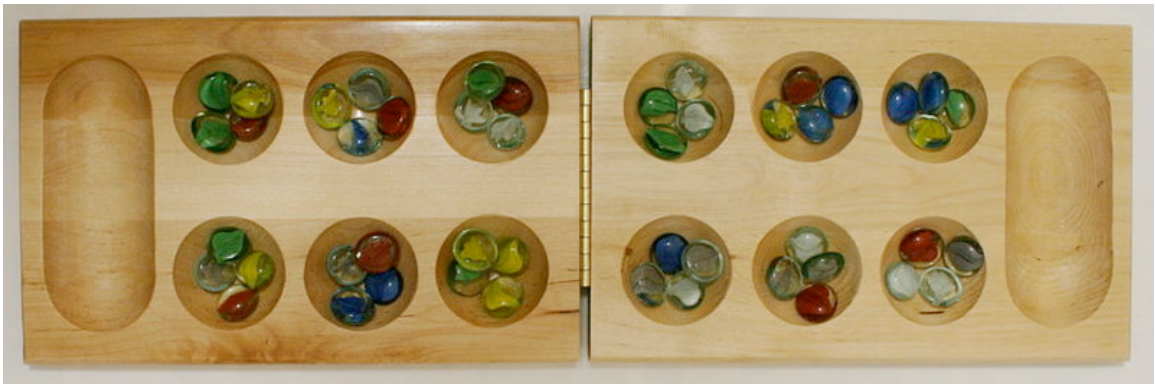
Introduction to Artificial Intelligence – EECS 348
Programming Assignment 2 – Mancala!!

You may also work alone or in pairs. You will submit only ONE assignment for the group. As a reminder, if you work in a group, you may not "divide and conquer." That is, all students must be present and contributing while any work on the project is being done. Per the submission instructions below, you will write the following statement at the beginning of your assignment (and it must be true): "All group members were present and contributing during all work on this project.".

The purpose of this assignment is to program some of the search algorithms and game playing strategies that we have covered in class. In particular, you will implement two aspects of an artificial intelligence game that plays Mancala (and other two-player games): search with alpha-beta pruning, and a game board evaluator. Your goals are to implement alpha-beta pruning correctly, and to create the best AI player you can.

Introduction to Mancala



Mancala is a two-player game from Africa in which players moves stones around a board (shown above), trying to capture as many 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).

On a player's turn, he or she chooses one 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 his or her Mancala but NOT his or her opponents Mancala, until he or she 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)** – *even if there aren't any stones on the opponents side to collect* (note – this interpretation of the rules differs from how they are implemented on

play-mancala.com). The game ends when one player cannot move on his or her turn, at which time the other player captures all of the stones remaining on his or her side of the board.

You can practice playing the game here: http://play-mancala.com/ .

Provided Code

We have provided code to support the basic game play, as well as a simple AI player (it's REALLY simple). It 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:
  - HUMAN: a human player
  - RANDOM: a player that makes random legal moves
  - MINIMAX: a player that uses the minimax algorithm and the board score function to choose its next move, limited by a specified ply depth
  - ABPRUNE: a player that uses alpha-beta pruned search and the board 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).
- 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, similar to what you implemented in Programming Assignment 1

You should download these files and make sure you can run them. To run a GUI game between two humans, load the file MancalaGUI.py (at a python interpreter "execfile("MancalaGUI.py")"). The GUI will not show up until you start the game using the commands below. Then create two players with type HUMAN (the ply is ignored for human players, and can be left as its default value):

```
>>> player1 = Player(1, Player.HUMAN)
>>> player2 = Player(2, Player.HUMAN)
>>> startGame(player1, player2)
```

You should see a window appear (it may appear in the background). You can now play Mancala with a friend.

But what if you don't have a friend, you ask? Well, never fear! The computer will play against you (and you will likely win). To play against the computer you simply need to create a computer player to play against, e.g:

```
>>> startGame(Player(1, Player.HUMAN), Player(2, Player.RANDOM))  # Ply is also ignored for random players
```

or

```
>>> startGame(Player(1, Player.HUMAN), Player(2, Player.MINIMAX, 5))
```

You can also play Tic Tac Toe using the same player object (but no GUI provided with Tic Tac Toe). Load the Tic Tac Toe game, create a new TTTBoard, and then use the "hostGame" function to play the game.

Once you understand how to run the code, make sure you read though the provided code and understand what it does. Notice that the minimax algorithm we discussed in class has already been implemented.

Part 1: A Better Board Score Function

The board score function in the basic player is too simple to be useful in Mancala--the agent will never be able to look ahead to see the end of the game until it's way too late to do anything about it. Your first task is to write an improved board score in the MancalaPlayer class, a subclass of Player. 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 part 2 below (alpha-beta pruning).

Part 2: Alpha-Beta Pruning search

The next part of the assignment is to implement the alpha-beta prune search algorithm described in your textbook and in class. Look in the code to see where to implement this function. You may refer to the pseudocode in the book, but make sure you understand what you are writing.

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.

You probably wish to test your alpha-beta pruning algorithm on something simpler than Mancala, which is why I 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 will take the agent awhile (20 seconds or so), but after that the agent will choose its moves quickly. Contrast this to minimax, where a ply greater than 5 takes an unreasonable amount of time.

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.

Part 3: Creating your best 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 (unless you choose to opt out).

Past years of resourceful students have led me to be more specific about my specification and restrictions for your players:

- Your player must compile without errors.
- Your player must make its moves in 10 seconds or less (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).
- The name of your player should be your netid (the netid of the student submitting the assignment for your group). Rename both the MancalaPlayer 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 would name my player "sho533." So, my file (which would be called "sho533.py") would contain a class called "Player" and a class called "sho533".
- 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 sho533.sho533(1,sho533.Player.CUSTOM)
  In other words, I will not initialize it with a ply - 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 code must compile in 5 seconds or less.
- 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.
- Let me know if you have any further questions!

What to submit: (please read all bullets below before submitting)

- netid.py: (i.e. the renamed "Player.py" file) The file containing all the code you have written, including your score function, your alpha beta pruning algorithm and your custom player.
- Be sure to include all of your team member's names and netid's in comments at the beginning of the file.
- If you worked in a group, in a comment at the beginning of the file, please write this exact statement. "All group members were present and contributing during all work on this project."

- Additionally, your code should be readable, commented and clear. There are many possible ways to approach this problem, which makes code style and comments very important here so that the grader and I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Optional (NOT Extra Credit):
If you find yourself incredibly interested in this assignment and want to do a little extra, read on….
As described above, my 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 "minimaxBonus" and a new alpha-beta pruning function, called "abPruneBonus", that takes into account that a player gets another move on their turn if they land in their own Mancala with their last stone.