Assignment 4

B351 / Q351

Due:
Tuesday, Oct 11th @ 11:59PM

1 Summary

- Develop a competitive AI for the game Mancala
- Demonstrate your understanding of the MiniMax search algorithm, alpha-beta pruning, and dynamic programming

We will be using Python 3 so be sure you aren't using older versions. Code that will compile in Python 2.7 may not compile in Python 3. See our installation guide for help removing old versions and installing Python 3.

You may optionally collaborate on this assignment, **but you can only share ideas**. Any shared code will not be tolerated and those involved will be subjected to the university's cheating and plagiarism policy. **If you discuss ideas, each person must write a comment at the top of the assignment file naming the students with whom ideas were shared.**

2 Background

Mancala is a two-player, turn-based strategy game where the objective is to collect the most pieces by the end of the game. To play a game, you need a Mancala board, which is a board made up of two rows of six pockets, also known as pits, and two "stores" located at the ends of these pits.

2.1 Rules

1. Each turn consists of a player picking up all the pieces in a pit and, moving counter-clockwise, deposit a stone in each pocket until the stones run out. When moving your stones, if you pass your store then you deposit one stone inside it as well, but you skip over your opponent's store.
2. If the last piece you place goes into your store, you take another turn.
3. If the last piece you place is in an empty space on your side, and your opponent has stones in the pit exactly opposite of this empty space[1], then you collect all the stones in your opponent's opposite pit as well as your last placed stone.

[1]Note that this rule differs slightly from the rules linked, which do not require your opponent have stones in their pit to make an "empty capture". However, this is consistent with the interactive website linked, and with how the game is most often played.

4. The game is over when all the stones on one side of the board are gone, and when this happens the other player takes all their stones on their side and puts them in their store.

5. To score the game, each player counts up the total number of stones in their store. The player with the most stones wins!

For those of you who have never played Mancala before this might seem like an overload of information. If you are confused about the structure/rules of the game, please visit the following links for more detailed explanations with visuals:
`https://www.scholastic.com/content/dam/teachers/blogs/alycia-zimmerman/migrated-files/mancala_rules.pdf` and

`https://endlessgames.com/wp-content/uploads/Mancala_Instructions.pdf`.

When designing a heuristic, it is vitally important to understand how the game is played. Because of this, we **highly** suggest that you play a couple of games to really understand the underlying rules and strategies. The following link will take you to a website where you can play some practice games against a computer which is a great way for beginners to learn the game:

`https://mancala.playdrift.com/`.

We also hope to share a brief video of ourselves playing the game to illustrate some of the rules.

3 Programming Component

In this assignment's programming component, you will implement an artificial intelligence agent that can competitively play Mancala. To achieve this, you will implement three key algorithms: MiniMax, alpha-beta pruning, and dynamic programming. Additionally, you will develop your own heuristic function to help evaluate maximum depth nodes in your game tree search.

3.1 Game Data Structures

You will utilize the following classes. You should read and understand this section **before** embarking on your assignment.

1. **Board Class**
   The `Board` class is the data structure that holds the Mancala board and the game operations.

The data structures that hold the current board state and the historical states are Python `lists`, but you will primarily be interfacing with them through the `p1_pits`, `p2_pits`, `p1_pot`, `p2_pot`, and `turn` properties. `p1_pits` and `p2_pits` are both `lists` which contain integers indexed from 0-5 representing how many stones a player has in a given pit. The indices for these lists increase counter-clockwise (an illustration of this can be seen at the top of the `__init__()` function.) `p1_pot` and `p2_pot` contain the integer value of how many stones each player has in their store (and has therefore scored) so far. `turn` is an integer, which is 0 when it is player 1's turn and 1 when it is player 2's turn.

Note that the Board class also contains a `list` called `_trace`, which contains a list of moves made so far. Traces are usually used for debugging and replaying what moves led to a board state. You can access a convenient string representation of the trace through the `trace` property.

You will use the following methods, properties, and attributes to interact with the `board`:

(a) `__init__([trace])` - This constructor function can be utilized in two ways:

   i. `no args` - If no arguments are passed, the constructor function generates a new board with 4 stones in each pit.

   ii. `trace` - If the `trace` argument is set to a valid string of moves, then the constructor function generates a new `Board` object based on that trace.

(b) `makeMove(pit)` - This function "picks up" all of the stones in the given pit, and places the stones according to Mancala rules. It also records the move in `_trace`, and saves a copy of the previous board in `board_history` for later recovery. The function automatically picks up stones from the side of whoever's turn it is. NOTE: This function does not perform any error checking by default. It is the responsibility of the user to ensure that proper non-empty pit is passed. See `getAllValidMoves()` for an example of proper usage.

(c) `undoMove()` - This will undo the last move made on a board, returning the board to the most recent state in the history, and updating the history, trace, and `game_over` variables appropriately.

(d) `getAllValidMoves([preorder])` - This generates a series of all the valid moves of the `Board` object. You may optionally pass a prior ordering that will influence the order in which moves are considered

and returned.

(e) `print()` - This prints a graphical representation of the current board state to your terminal.

(f) `board` - `board` is a 15-item integer array containing (in order) the first player's six pits, the first player's store, the second player's six pits, the second player's store, and the next player to play. All of this data is more easily accessed through different properties.

(g) `p1_pot`, `p1_pits`, `p2_pot`, `p2_pits` - These properties allow you to access information about the current state of the board. `p1_pot` and `p2_pot` are integers; `p1_pits` and `p2_pits` are six-item integer arrays, arranged counterclockwise. (For instance, `p1_pits[0]` is opposite to `p2_pits[5]`.) These properties are often used in your `heuristic` functions.

(h) `turn` - This property is 0 when it is currently player 1 (max)'s turn and 1 when it is currently player 2 (min)'s turn. This property is used to determine whether to minimize or maximize in your `minimax` functions.

(i) `game_over` - This attribute is `True` when the game is over and `False` otherwise.

(j) `winner` - This property (not function) contains a value representing the terminal state of the game. If the game is over, it will contain `-1` for a draw, `0` for a player one win, or `1` for a player two win. If the game is not over, it will evaluate to `None`.

(k) `trace` - This property (not function) contains a convenient string representing the sequence of moves that led to the current state of the board. This can be used to easily recreate another similar board, and is used in our debugging and grading tools.

(l) `state` - This property (not function) contains an integer representing the current state of the board. Note that two different traces might lead to the same state. `state` is used in the dynamic programming portion of this assignment.

2. **Player Class**

In this class, you will implement your search algorithm. This class is utilized by the `Game` class to simulate a game using your AI.

The base player class must define constants `P2_WIN_SCORE`, `TIE_SCORE`, and `P1_WIN_SCORE` to represent scores for the different end states of the game.

You will work with the following common to all Players:

(a) `findMove(trace)` - This will return the optimal move for `trace` upon executing a game tree search up to the Player's `max_depth`. It will call an appropriate function according to the particular algorithm the player uses. Some Player subclasses do very unusual things here!

(b) `heuristic(board)` - This function returns a value representing the "goodness" of the `board` for each player. Good positions for Player 1 will return high values. Good positions for Player 2 will return low (or highly negative) values. As explained later, it is your responsibility to implement this function.

We provide the following subclasses of `BasePlayer`:

(a) `RandomPlayer` - this subclass randomly selects one of the valid moves to execute each turn.

(b) `ManualPlayer` - this subclass lets you play directly against your AI. Very fun!

(c) `RemotePlayer` - this subclass lets you play against the same instructor AI you are tested against on the Gameplay criterion. (You are tested with the instructor AI at a depth of five.) This subclass depends on the `requests` library.

You will implement your search algorithms in the subclasses of `BasePlayer` following the instructions in the next section.

Additionally, you can create subclasses of your various Players with different heuristic functions in order to test your heuristics against each other locally. We have provided an example of how to accomplish this.

3. **Game Class**
This class is found within the `a4.py` file. This contains the interface for testing your search algorithms.

You will work with the following functions:

(a) `runGame()` - This will simulate a game using the AI agents that you create using your `Board` and `Player` classes. In particular, it will test if your search strategy was implemented properly and can be used to test heuristics against each other.

You may find it beneficial to change the implementation and parameters of one or many of the above functions/classes in order to increase the efficiency of your code. **Edits are not only accepted, but they are expected**

(except where explicitly banned).

That being said, please note that your changes **must** be able to interface with the functions in the `a4.py` file (`Game` class) and test cases. Failure to achieve this will preclude you from receiving any bonus, and will likely result in significant loss of points. You must define each of the homework functions to the specification outlined, except in your bonus player and board classes.

3.2 Objectives

Your goal is to complete the following tasks. It is in your best interest to complete them in the order that they are presented.

3.2.1 BasePlayer.heuristic()

You must complete the `heuristic()` function in the `BasePlayer` class to return a game state value that reflects the "goodness" of the game. Remember, the better the game state for Player 1, the higher the score; the better the game state for Player 2, the lower the score (can and probably should be negative). Take into account whether it is valuable to control more or fewer stones, how late into the game it is, and anything else you find relevant.

Additionally, you must define `P2_WIN_SCORE`, `TIE_SCORE`, and `P1_WIN_SCORE` so that all heuristic values fall between the two players' win scores.

As this function will be called at almost **every** leaf node in your MiniMax search, you will want your function to be as efficient as possible.

Note: While you may test your heuristic in different subclasses, the final heuristic that you'd like to be evaluated on must be in the main `BasePlayer` class.

3.2.2 MiniMax Search

Your goal in this section is to develop a generic MiniMax search algorithm without any optimizations (i.e., alpha-beta). For a reference to this algorithm, you may revisit the lecture slides or see the following Wiki page: `https://en.wikipedia.org/wiki/Minimax`.

In your `PlayerMM` class you will find the `minimax()` function where you must implement the search algorithm.

When creating your search algorithm, please consider the following:

1. The function should ultimately return the pit that represents the best move for the player.

2. Your algorithm should correctly identify end positions and assign them the correct value constant without doing any unnecessary processing. Please note that this can be trickier than it looks, so be sure that you think carefully about your value assignments and use the constants you defined for all Player classes.

3. Your algorithm should correctly stop at maximum depth and return the appropriate heuristic value.

4. You should keep track of whose turn it is and where you are relative to the maximum depth.

5. You should always explore all possible children (except when you have reached maximum depth).

If you have successfully completed this section, and `Player.heuristic()`, you should be able to run `runGame` in the `Game` class and achieve meaningful results (i.e., it will play a Mancala game to completion). **Do not proceed until this works properly.**

3.2.3 Alpha-Beta Pruning

Your goal in this section is to implement alpha-beta pruning in the MiniMax search algorithm that you developed in the previous section. For a reference to this pruning technique, you may revisit the lecture slides or see the following Wiki page: `https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning`.

For this, you will implement the `alphaBeta()` function of the `PlayerAB` class, and you should continue to ensure that it returns the best column to move in for the player.

When implementing alpha-beta pruning, please consider the following:

1. You should update both `alpha` and `beta` whenever appropriate.

2. You should identify the correct pruning conditions and that you place `break` statements in the correct locations.

When implemented correctly, your new MiniMax algorithm should run significantly faster than the one you initially developed. That being said, it should emit exactly the same answer (for the same `depth`).

3.2.4 Dynamic Programming

Your goal in this section is to implement dynamic programming in the MiniMax search algorithm that you developed in previous sections. For a reference to dynamic programming, you may revisit the lecture slides.

For this, you will implement the `heuristic()` function of the `PlayerDP` class. Keep in mind that this is overriding the `heuristic()` function of `BasePlayer`. To access that function, you may call `BasePlayer.heuristic(self, board)` or `super().heuristic(board)`. Your implementation must not differ from the heuristic value returned by `BasePlayer`

When implementing dynamic programming, please consider the following:

1. Remember that you cannot store objects as keys in hash maps because they are mutable. Instead you will need to use a hashable representation of your `Board` objects. Fortunately, this is already provided to you in `Board.state`.
2. You must use `self.resolved` to store the lookup table for your dynamic programming. We will be using this to test and grade this section of the assignment.

As with alpha-beta pruning, your new Alpha-Beta algorithm with dynamic programming will likely run significantly faster (depending on how expensive your heuristic is) than the one you initially developed. That being said, it should emit exactly the same answer (for the same `depth`).

4 Tests

Very simple tests have been given for you in `player_test.py` for the functions you implemented. Running the main of these files will run the tests. Be sure to use these to make sure your implementations are working generally as requested. These test cases are NOT all encompassing. Passing these does not mean your implementation is completely correct. In order to save submissions, it is advised to write additional unit tests for more complex puzzles as well as to cover cases that these tests do not. In `player_test.py` we provide some ideas as to what else you should test on your own. This isn't required, but it is highly recommended, as learning to debug on your own is an important skill.

Once all of your functions have been completed, you can also test with the main function in `a4.py`. Additionally, you can and should utilize traces as an extra debugging tool.

5 Grading

Problems are weighted according to the percentage assigned in the problem title. For each problem, points will be given based on the criteria met in the corresponding table on the next page and whether your solutions pass the test cases on the online grader.

Finally, remember that this is an individual assignment.

5.1 BasePlayer (5%)

Criteria	**Points**
`P2_WIN_SCORE`, `TIE_SCORE`, `P1_WIN_SCORE` all real-valued (and not infinity)	1
`P2_WIN_SCORE < TIE_SCORE < P1_WIN_SCORE`	1
Heuristic is real-valued on all boards	1
Heuristic always falls between `P2_WIN_SCORE` and `P1_WIN_SCORE`	2
Total	5

5.2 minimax (logic) (15%)

Criteria	**Points**
(Game over) Returns a null move and the appropriate score constant for board's terminal state	2
(Non-end state & `depth = 0`) Returns a null move and the heuristic value for board	2
(Not base case) Calls `board.getAllValidMoves()` exactly once, optionally specifying an order on the pits	1
(Not base case) Calls `board.makeMove(pit)` and `board.undoMove(pit)` exactly once for each valid move	2
(Not base case) Recursively calls minimax on board after trying each move, with depth - 1, to score each valid move	3
(Not base case) Returns the maximum or minimum-scored move and its score, based on board.turn	5
Total	15

5.3 minimax (correctness) (10%)

Criteria	**Points**
The arrangement of nodes in the call tree is correct	3
Scores for each node are correct	7
Total	10

5.4 alphaBeta (logic) (20%)

Criteria	Points
(Game over) Returns a null move and the appropriate score constant for board's end state	2
(Non-end state & `depth = 0`) Returns a null move and the heuristic value for board	2
(Not base case) Calls `board.getAllValidMoves()` exactly once	1
(Not base case) Calls `board.makeMove(pit)` exactly once for each valid move	2
(Not base case) Recursively calls `alphaBeta` after trying each move, with depth - 1, to score each valid move	1
(Not base case) Returns the first encountered maximum or minimum-scored move and its score, based on board.turn	2
Updates alpha or beta value (as appropriate) and passes values down to recursive calls	5
If a cutoff occurs, does not make any further recursive calls	3
If a cutoff occurs, returns the value that caused the cutoff and a null move	2
Total	20

5.5 alphaBeta (correctness) (15%)

Criteria	Points
The arrangement of nodes is correct	3
Scores for each node are correct	7
Alpha/beta values passed to each node are correct	5
Total	15

5.6 PlayerDP (10%)

Criteria	Points
Checks `self.resolved` for the board's state and returns it without recalculation if found `self.resolved`	5
Otherwise, calls `BasePlayer.heuristic` to calculate a heuristic value for this state. (Does not calculate it in this function.)	3
Stores calculated values in `self.resolved`, indexed by the board's state	2
Total	10

5.7 Competency and Gameplay (25)

Criteria	**Points**
Chooses moves well at a depth of 1	2
Chooses moves well at a depth of 3	3
Chooses moves well at a depth of 5	5
Beats a random player on a blank board as both P1 and P2	5
Beats the instructor AI more than 50% of the time on our suite of boards	10
Total	20