Assignment 4

B351 / Q351

Comprehension Questions Due:
Tuesday, October 8th, 2019 @ 11:59PM

Initial Due:
Tuesday, October 15th, 2019 @ 11:59PM

1 Summary

- Develop a competitive AI for the game Connect 4
- 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.

To run your completed programs, you may find that you need to install the `requests` module. You will do this utilizing Pip and using your terminal to execute the command `pip install requests`. If your code still does not run due to missing modules, please come by office hours.

Please submit your completed files to your private Github repository for this class. You may not make further revisions to your files beyond the above deadline without incurring a late penalty.

You may collaborate on this assignment (optional) **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

You may already be familiar with the game Connect 4. If you are not, please see the following links: `https://en.wikipedia.org/wiki/Connect_Four` and `https://www.mathsisfun.com/games/connect4.html`.

Even those who have played the game before may want to peruse `http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf`. While not critical to read before starting, Allis' thesis may help you develop a better heuristic when the time comes. Of particular interest to this assignment are sections 3 and 4 of his paper. Before coming to office hours with questions about heuristic improvement, please read those sections of the paper first.

3 Programming Component

In this assignment's programming component, you will implement an artificial intelligence agent that can competitively play Connect 4. 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 Connect 4 boards and the game operations.

   The underlying data structure is a 2-dimensional Python `list` called `board`. The first dimension contains columns, and the second dimension represents each row. The structure is indexed starting from (0,0) at the bottom left of the game board. Thus, `board[1][4]` would be the fifth cell up in the second column from the left.

   Every cell in `board` contains either a "0" or a "1" which represent player pieces. Player 1 is represented by "0" pieces, and Player 2 is represented by "1" pieces. While this is somewhat confusing, please trust that using 0s and 1s (rather than 1s and 2s) helps to boost the speed of your program.

   The lists that represent the columns in `board` are only as long as the amount of pieces in the column. For example, an empty column will have length 0. Please be considerate of this fact as you traverse the `board` in order to avoid index errors.

   Note that there are two different ways of representing board objects: states and traces. Traces correspond with sequences of moves, and states correspond with board layouts. This means that there might be more than one trace for each state. For example, the traces 1122 and 2211 correspond with the same state. Traces are usually used for debugging, whereas states can be used for all sorts of optimizations.

You will use the following functions to interact with the `board`:

(a) `__init__()` - This constructor function can be utilized in three ways:

   i. `no args` - If no arguments are passed, the constructor function generates a new, empty board.

   ii. `orig` - If the `orig` argument is set to another `Board` object, then the constructor function generates a deep copy of the `orig` board.

   iii. `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) `placeMove(column)` - This "drops" a piece in the indicated column, and records a tuple `(piece, column)` column as `Board.prevMove`. The function automatically determines whose turn it is based on how many pieces have been played. NOTE: This function does not perform any error checking by default. It is the responsibility of the user to ensure that proper column values are passed. See `getAllValidMoves()` for an example of proper usage.

(c) `getChild(column)` - This generates a new child `Board` object, where the parent is the current `Board` instance with a move made in the supplied `column`.

(d) `getChildState(column)` - This returns the state of the child `Board` object.

(e) `getChildTrace(column)` - This returns the trace of the child `Board` object for the given `column`, if and only if tracking is enabled for the current `Board` object.

(f) `getAllValidMoves(order=range(7))` - This generates a list of all the valid moves of the `Board` object. The optional parameter, `order`, allows you to specify the order in which moves are generated. The default is 0-6 ascending.

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

(h) `isFull()` - This function returns true iff the board is full (draw state).

(i) `isEnd()` - This function returns a value representing the current state of the board. It will return `None` iff the game is not over. Otherwise, it will return `-1` for a draw, `0` for a player one win, or `1` for a player two win.

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

   You will need to work with the following functions:

   (a) `findMove(board)` - This will return the optimal move (column to move in) for `board` upon executing a game tree search up to the `maxDepth`.

   (b) `myHeuristic(board)` - This function returns a value representing the "goodness" of the `board` for each player. Good positions for Player 1 ("0" pieces) will return high values. Good positions for Player 2 ("1" pieces) will return low values. As explained later, it is your responsibility to implement this function.

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

   Additionally, you can create subclasses of Player 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) `simulateLocalGame()` - This will simulate a local 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.

(b) `playAgainstInstructor(difficulty)` - This is very similar to the above function. However, this function is used to play against various AIs created by the course instructors. It will pit `player1` of your `Game` class against the instructor AI. You can choose the difficulty of the instructor AI by changing the `difficulty` parameter. Further instructions can be found in `A4.py`.

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.

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 Board.isEnd()

You must complete the `isEnd()` function in the `Board` class to determine if the game state is a end state or not. See the function contract defined above and in the `board.py` for the potential return values.

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

3.2.2 BasePlayer.myHeuristic()

You must complete the `myHeuristic()` 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 ("0" pieces), the higher the score; the better the game state for Player 2 ("1" pieces), the lower the score (can and probably should be negative).

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 `Player` class.

3.2.3 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 column 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.
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, `Board.isEnd()`,
and `Player.myHeuristic()`, you should be able to run `simulateLocalGame` in the `Game` class and achieve meaningful results (i.e., it will play a Connect 4 game to completion). **Do not proceed until this works properly.**

3.2.4 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`).

3.2.5 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 `myHeuristic()`function of the `PlayerDP` class. Keep in mind that this is overriding the `myHeuristic()` function of `BasePlayer`. To access that function, you may call `BasePlayer.myHeuristic(self, 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 an integer representation of your `Board` objects. Fortunately, this is already provided to you in `Board.state`.
2. You must use `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 should run significantly faster 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, with additional tests for `isEnd()` in `board.py`. 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 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**
`self.P2_WIN_SCORE`, `self.TIE_SCORE`, `self.P1_WIN_SCORE` all real-valued	1
`self.P2_WIN_SCORE < self.TIE_SCORE < self.P1_WIN_SCORE`	1
Heuristic is real-valued on all boards	1
Heuristic always falls between `self.P2_WIN_SCORE` and `self.P1_WIN_SCORE`	2
Total	5

5.2 isEnd (5%)

Criteria	**Points**
Returns None for non-goal boards	1
Returns -1 if the game is a draw	1
Returns 0 if player 1 wins	1
Returns 1 if player 2 wins	1
Is correct in all cases	1
Total	5

5.3 minimax (logic) (15%)

Criteria	**Points**
(Non-end state & `depth > 0`) Calls `board.getAllValidMoves()` exactly once, possibly specifying an order on the columns	1
(Non-end state & `depth > 0`) Calls `board.getChild(col)` exactly once for each valid move to generate children	2
(Non-end & `depth > 0`) Recursively calls minimax on each child board, with depth - 1, to score each valid move	3
(Non-end & `depth > 0`) Returns the maximum or minimum-scored move and its score, based on board.turn	5
(End state) 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
Total	15

5.4 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

5.5 alphaBeta (logic) (20%)

Criteria	**Points**
(Non-end state & `depth > 0`) Calls `board.getAllValidMoves()` exactly once, possibly specifying an order on the columns.	1
(Non-end state & `depth > 0`) Calls `board.getChild(col)` exactly once for each valid move to generate children	2
(Non-end state & `depth > 0`) Recursively calls minimax_ab on each child board, with depth - 1, to score each valid move	1
(Non-end state & `depth > 0`) Returns the first encountered maximum or minimum-scored move and its score, based on board.turn	2
(End state) 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
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, either returns the value that caused the cutoff or a null value	2
Total	20

5.6 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

5.7 PlayerDP (10%)

Criteria	**Points**
Saves heuristic values for boards in `self.resolved`	5
Returns saved heuristic values for boards that have already been resolved, without recalculating them	5
Total	10

5.8 Competency (20%)

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 late-game boards	5
Total	20

6 Bonus

6.1 Battle Tournament

Participants compete in a round-robin tournament that test their optimized implementations of a Connect 4 AI.

6.2 Rules

Round-robin tournament (every participant plays ever other participant) on the same set of 5 boards with 2 games for each board. Starting player alternates each game, showcasing your AI's performance as maximizing player or minimizing player. Some boards are weighted towards a specific player winning, which adds an overall balance.

Each participant should be able to pass game play tests without timing out in order to avoid time-out losses.

The top-ranked participant overall will receive at least 25 points of extra credit on this assignment, and the second-ranked participant will receive 15 points. With potential increases dependent on number of competitors and effort put into AI. Overall, credit will be allocated based on the impressiveness of your performance.

6.2.1 Other Restrictions

- All code must be in standard Python 3.
- Submitted code must be your own work.
- Must cite all sources for ideas and algorithms.
- Cannot change behavior of any Python libraries.

- Your code must be entirely contained within your board.py and player.py files

- You may not use blank, or catch-all, `except` calls in your code. Each `except` statements must catch a specific error. (e.g., `except IndexError:` is allowed).

6.2.2 Submissions

You may enroll your code in the tournament at any point until:
11:59PM Friday October 18th

You may enroll your code as many times as you want. For instance, you may enroll your code to see how it performs against your opponents'. If it loses to a majority of your opponents, you may make changes that you think will improve the performance and competitiveness of your AI.

The process to enroll your code is similar to how you run compilation tests or submit on the grading tool, in that it uses your GitHub repository in order to fetch your `a4` files. Each commit you make to your a4 assignment will re-enroll you and re-run your matches.

The url you will use to access the tournament is:
`https://b351.sice.indiana.edu/battles`

Please note that this url is not live at the moment! An announcement will be sent out whenever it is up, and the url itself is subject to change.

6.2.3 Suggestions for Optimizations

To aid in the above, you may want to implement and document optimizations that will allow you AI to compete more effectively. We have provided some ideas below:

1. **Iterative Deepening** – Finish the iterative deepening template provided in your `search.py` file in order to execute best-first ordering of child nodes to optimize your alpha-beta pruning. The effective implementation of this algorithm can increase your search depth by almost 50%! See `https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search` as a reference.

2. **Threat Tracking** – Currently, you likely must traverse your entire board in your `isTerminal` and `heuristic` functions. This can cause significant overhead! Is there any way that you can reduce the amount of computation required by explicitly tracking threats (3-in-a-rows with an open slot)?

3. **Redundancy Reduction** – Are there any points in your algorithm where you do the exact same computation more than once? Can you eliminate this redundancy? A good place to start might be to look at where your `hash` function is used.

4. **Delayed Computation** – When striving for optimum performance, it is often a good idea to delay computation until you are absolutely sure that you will need it. Are there any points in your code where you can further delay computation?

5. **Fail-soft vs. Fail-hard Alpha-Beta** – Which type should you use? Knowing this, how can you take advantage of your heuristic and terminal position values to streamline your computation?

6. **BE AFRAID TO DIE** – Often, agents implementing the minimax algorithm will "give up" when they are in a losing situation and go along with the loss, not delaying it in any way. If your opponent does not yet see their winning configuration, your agent might be able to take steps to avoid it. How could you change your code to prefer slower deaths (and quicker kills)?

7. **Profiling** – One of the best ways to approach optimization is to examine your code for performance bottlenecks and work on optimizing those sections of your codebase. Python has several great built-in tools for profiling your code's performance. You should start here: `https://docs.python.org/3/library/profile.html`.

Note that to implement many of these optimizations you will **need** to change the structure of some of the starter code that we give you. While this is encouraged, be sure to save regularly in order to unwind any mistakes that you might make in your experimentation. It would be a good idea to save a working copy of your files in a separate location so that you always have something to revert to if things go terribly wrong.