Assignment 4

B351 / Q351

Programming Due: February 27th, 2018 @ 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

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 Due date and time without incurring a late penalty.

This assignment can be completed with **one** partner (or none if you prefer.) If you choose to work with a partner, you **must** include your partner's name in your files. Both you and your partner must submit your own files to your own repositories. Your files will be run against Moss to check for plagiarism within the class outside of your partnership.

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.

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.

   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. `hash` - If the `hash` argument is set to a valid integer hash (see `hash()`), then the constructor function generates a new `Board` object based on that hash.

   (b) `makeMove(column)` - This "drops" a piece in the indicated column, and records a tuple `(piece, column)` column as `Board.lastMove`. 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 `children()` for

an example of proper usage.

(c) `children()` - This generates a list of all the valid children of the `Board` object. The returned list contains tuples. The first index of each tuple holds the move (column moved in) that generates the child `Board` object stored in the second index.

(d) `hash()` - This returns a unique integer representation of the current board state. While it is not important for you to understand how this is done, you will need to use the generated hash values for the dynamic programming section of your assignment.

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

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

(g) `isTerminal()` - This function returns a value representing the current state of the board. It will return -1 iff the game is not over. Otherwise, it will return 0 for a draw, 1 for a player one win, or 2 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.

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 `depthLimit`.

(b) `heuristic(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) `simulateRemoteGame(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.isTerminal() (15%)

You must complete the `isTerminal()` function in the `Board` class to determine if the game state is terminal 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 Player.heuristic() (10%)

You must complete the `heuristic()` function in the `Player` 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).

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 (25%)

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 `findMove()` 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 terminal positions and assign them the correct value 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 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.isTerminal()`, and `Player.heuristic()`, 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 (25%)

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 `findMove()` 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 (0%)

~~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 `findMove()` function of the `PlayerABDP` 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 place your lookup in a position to avoid the most work possible.~~
2. ~~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 `hash()`.~~
3. ~~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`).~~

3.2.6 Game Play (25%)

A portion of your grade will be assigned based on how well your AI plays the game versus different difficulties of the instructor implementation. **We will release the exact scoring chart once the server with the instructor AI is online and you are able to connect.**

The easiest way to achieve full points in this section is to implement a good heuristic function. For ideas on how to proceed, please do examine the text linked to in Section 2. This text provides a comprehensive overview of the game theory for Connect 4. While you do not need to know all of the theory to implement your `heuristic()` function, the text contains many useful insights.

You may want to approach your heuristic development iteratively. Develop a heuristic that does an okay job, and then circle back to refine it later once your search algorithms are developed and you can test heuristics against each other.

4 Bonus

If you are up for a challenge, you may attempt to complete this series of bonus problems for a **maximum bonus of 40%**.

4.1 Objectives

The goal of this bonus assignment is to continue to optimize your search algorithm to become even more competitive. You will then have the opportunity to compete against the AIs developed by your classmates.

4.1.1 Competitive Play (20%)

Later this semester, we will open a server that allows you to battle your AI against your peers' AIs. Through a tournament style system, your AI will receive a ranking in the class and you will be awarded bonus points based on that ranking (up to 20%).

Your work must be entirely yours (and your partner's). Any plagiarism will result in an **automatic failing grade** on the entire assignment.

Finally, we require that you include in your GitHub repository a brief writeup on how you developed your heuristic and what it ultimately accomplishes.

4.1.2 Optimizations (20%)

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 `isWinning` 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. **Dynamic Depth** – Would altering the depth of the search based on the game state allow you to achieve performance gains, particularly toward the end of any given game? How might you go about implementing this in your `Game` class?

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.

As you complete your optimizations, you must document them in an A4 .pdf or .txt file. In order to receive credit you must detail (1) what you did and (2) why you did it. You will submit this file along with your project to your GitHub

repository.

The amount of bonus assigned in this section will depend entirely on your instructor's discretion and how impressive your optimizations are relative to your peers.