Connect Four Artificial Intelligence (AI)

Preamble

This assignment introduces you to a core Artificial Intelligence (AI) concept that utilizes trees. In this assignment, you will implement a tree that can approximate the best possible move given a game board. Please read the whole handout before starting as there are many new concepts you have not seen before.

Learning Objectives

- Achieve proficiency with using trees and recursion
- Learn about the minimax algorithm
- Develop an awareness for how computationally infeasible it may be to enumerate all possible paths of a game
- Learn about the Model-View-Controller software architecture.

Collaboration policy and academic integrity

You may do this assignment with one other person. Both members of the group should get on CMS and do what is required to form a group well before the assignment due date. Both must do something to form the group: one proposes, the other accepts.

People in a group must *work together*. It is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. Take turns "driving", using the keyboard and mouse.

With the exception of your CMS-registered group partner, you may not look at anyone else's code, in any form, or show your code to anyone else (except the course staff), in any form. This includes having a public repository for your code, like github.

Getting help

If you don't know where to start, if you don't understand testing, if you are lost, etc., *see someone immediately* - an instructor, a TA, a consultant. Do not wait. A little in-person help can do wonders. See the course homepage for contact information and the course calendar for updates on office hours

Connect Four: The Game

The purpose of A5 is to create an AI program that can skillfully play Connect Four. Connect Four is a two-player game in which the two players take turns dropping colored chips from the top into a 7-column, 6-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The object of the game is to connect four of one's own chips (of the same color) in a row vertically, horizontally, or diagonally before your opponent does.
(Source: http://en.wikipedia.org/wiki/Connect_Four)

Your job

There are two parts to this assignment. First, you have to familiarize yourself with the classes we give you and learn

about the Model-View Controller software architecture. This includes writing two methods so that the game can be played.

Second, you have to write methods that implement an AI (artificial intelligence) technique called the minimax algorithm.

We start by explaining the architecture of the ConnectFour code base. A software architecture is a high-level understand of how the many components of the program fit together and relate to each other. For ConnectFour, we have adapted an architecture known as Model-View-Controller, which is an extremely common architecture for (human-) interactive programs.

Model The model represents the abstract concept that the interactive program is manipulating. For ConnectFour, the program is manipulating a Connect Four board. This is represented by the Board class, which is simply a grid indicating which locations are occupied and by which player. This grid is stored as field board of class Board. The model tells us things like what moves are still possible to make and by whom, whether the game is over, and who won if anyone. But the model doesn't decide which moves are made or how the board is displayed.

View The model is just a bunch of values. The view decides how to communicate the values to the user. In ConnectFour, the subclasses of the class UI serve as the view. We provide two such subclasses: Console and GUI. Console presents the model textually, as lines of ASCII characters on the console. GUI presents the model graphically, as a grid of colored circles with animations for when moves are made. In our design, class UI itself is not a view, but instead serves as the glue between the model, view, and controller.

Controller The controller is in charge of deciding what changes should be made to the model. For Connect Four, the controller is in charge of deciding which move is played on the board as the game is played. In ConnectFour, the implementations of interface Player serve as the controllers. We provide four such implementations: Random, File, Human, and AI (where AI has to be finished by you). The Random player simply chooses to play a random possible move each time. The File player reads moves from lines of a file. The Human player interacts with the users of the ConnectFour program. The AI player uses minimax to intelligently and automatically choose its moves to play.

Interestingly enough, we do not have an explicit class called Human. Instead there are two anonymous classes for Human, one connecting with a Console view and the other with a GUI view. In modern programs, the controller often has to be coupled with the view. This is illustrated in our design by the fact that the Console and GUI views each implement the Human player differently; Console reads text from the console; GUI listens to buttons displayed as part of its view.

Logging Many software architectures integrate some form of logging. Logging is responsible for outputting a record of the key events that occur during the program's execution. The hope is that, if at some point the output seems faulty, one can backtrack through the logs to find out where exactly things first went wrong. In ConnectFour, classes implementing interface Logger serve as the logging. Class UI, as it runs the game, also makes sure to inform the logger about each critical step as it happens.

We provide two loggers: StateLogger and MegaLogger. StateLogger is designed for the AI player. As the game progresses, StateLogger repeatedly records a detailed representation of the AI's internal minimax tree. This record should be extremely useful to you for debugging your AI implementation.

MegaLogger is more general purpose than StateLogger. It can log the progress of the board, the moves made by each player, and the states of each AI player. This is done by supplying MegaLogger one or multiple arguments of the form board=FILENAME.txt, move1=FILENAME.txt, move2=FILENAME.txt, state1=FILENAME.txt, and state2=FILENAME.txt. MegaLogger logs moves in the format expected by player File, enabling player File to replay moves made by you or an automated player from an earlier game.

The simplest way to play the game is to execute method Setup.main. It provides arguments to the ConnectFour application. Currently, the arguments start a game with two human players, Ross and David. You can also execute the ConnectFour application directly from the command line using

java ConnectFour UI Player Arg Player Arg [Logger Args…]

Or, you can set up Eclipse Run Configurations to execute ConnectFour and set the arguments.

In the above command line, the UI argument is the view you want to use: Console or GUI. For the first Player argument, give the player you want to use: Random, Human, or AI. The following Arg argument specifies the seed, name, or depth of the player. The second Player and Arg arguments similarly specify the controller for the second player.

The optional Logger argument, and its succeeding Args arguments, specify which logger to use and how it is configured (e.g. which file to record to). If no Logger argument is supplied, then the logger is used. With this setup, you can easily play and experiment with Connect Four in a wide variety of ways, as well as develop a wide variety of tests for your AI implementation.

Now that you have a high-level understanding of the code base, we next explain what you have to do to apply that understanding. The AI part follows, and you would be wise to wait until you have completed the first part to even look at the second part.

Part 1: Getting the game to play —writing two methods

This assignment requires you to fill in only small pieces of code. But to do so correctly requires understanding what those pieces of code are supposed to do. To figure this out, you need to read the specifications, and it will help to read through other parts of the code that interact with your pieces. There's a saying that programming is 10% writing code and 90% reading code. That is why we so strongly emphasize style guides, documentation, and clean solutions. We hope this assignment will help you appreciate the value of readability.

1. Download the A5 zip file from either the pinned Piazza A5 FAQs note or the course website, start a new project named a5 (or something similar) in Eclipse, and import the .java files into it. The simplest way to put all the .java files in is to simply to drag them over directory src. Make sure you copy, not link, to the files. Save the .txt files somewhere; they are used to help you debug.
2. Run Setup with a GUI and two human players. You can't do anything! *womp womp* That's because the model isn't finished. Implement method getPotentialMoves in class Board according to its very thorough specification. Once this is done, you and your friend should be able to play Connect Four. Column buttons should become disabled as the columns fill up.
3. Run Setup with a GUI and two random players. Currently the random players always choose the leftmost column, even if it's full, which crashes the game —not very random. Implement method getMove in class Random to randomly select one of the possible moves. Note that Random has a field called random that generates numbers for you.
4. The A5 zip file contained files random1.txt and random2.txt. These are the result of executing with these parameters:

   Console Random cs2110 Random a5 MegaLogger move1=random1.txt move2=random2.txt

Run with these parameters. Two files will be written in your Eclipse directory for this project. Compare their contents with the two A5 zip file .txt files mentioned above. If they specify different moves, then you have an error in either Board.getPossibleMoves or Random.getMove.

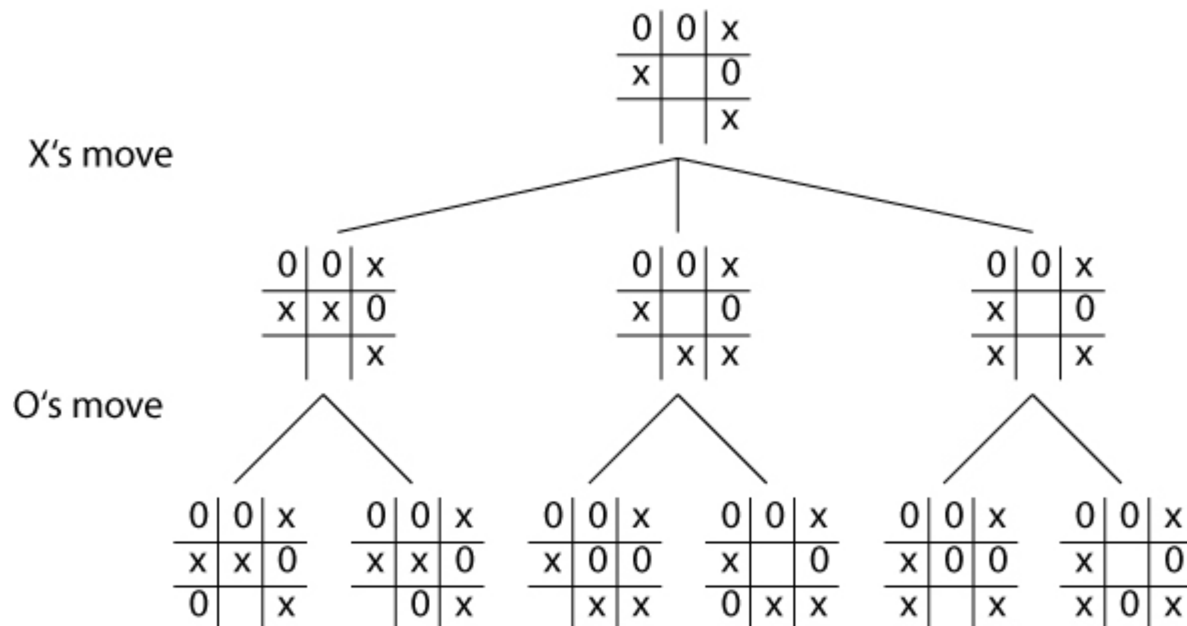
Part 2: Doing the AI stuff

We now talk about implementing an AI (artificial intelligence technique and discuss implementing it in class AI.

Minimax Algorithm

Suppose your program wants to make a move in Connect 4. Before it makes the move, it will look ahead at the possible sequences of moves it and the opponent could make and determine which first move is best, based on all those sequences of moves. The key data structure behind this is a tree in which the nodes are possible game states. Each state consists of the current board and which player's turn it is. From this state, the AI can generate all possible future states that can result after that player takes his turn.

Below is a partial game tree for *tic tac toe*. In the root of the tree, Player X has three possible places to play, so there are three states children of the root state. From each of those states, the states are then generated for Player O's potential moves. This can continue until a state is reached that ends the game or results in a tie game. In the case of Connect Four, this happens when the board consists of one player having four discs connected, or when all the columns are completely filled.



It is usually computationally infeasible to generate all states of a game tree because the state space grows exponentially with the number of turns. Therefore, one generally constructs the game tree up to some fixed depth, expanding the relevant subtrees further as the game is played.

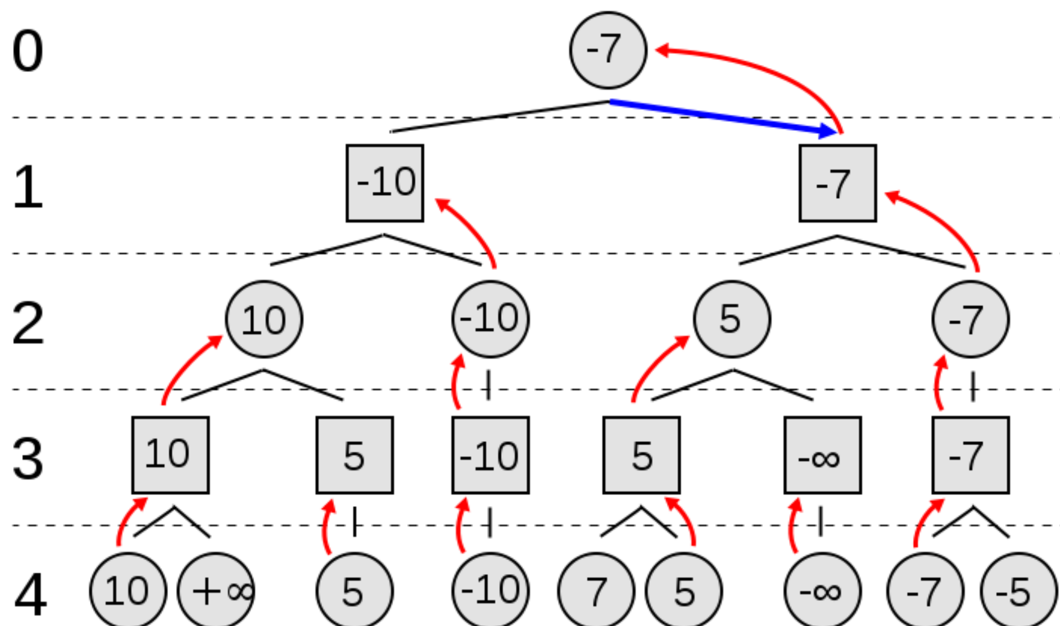
After the game state tree has been constructed, values need to be associated with each state in order to know which move is the best to make. The leaves of the tree are evaluated first: Each leaf calls an evaluation function to evaluate the state's board. The evaluation function gives a score to the state. The higher the score, the better the board is for you. If the state has four discs connected in your favor, the score is positive infinity (or the maximum integer value.) If it is a losing board, the score is negative infinity (or the minimum integer value.)

The last thing to do is evaluate the parent nodes in the tree. The game state tree creator (you) is trying to win the game. You want to maximize your score, while your enemy is trying to minimize your score.

This knowledge will help determine the value for the parent states. If the current state is your color (i.e. it is your turn in that state), use the *maximum* value from the list of children as the current state's value. If the current state is not your color, use the *minimum* value from the list of children.

Now you see why it is called the Minimax algorithm.

For example in the illustration below, the circles represent your color and squares the opponent. The tree has already been constructed and the evaluation function was run at level 4. It is the opponent's turn at level 3, so the minimum value is always selected from the children. While at level 2, it is your turn, so you always select the maximum value from level 2's children. Ultimately, the best possible known move bubbles up to the top where at level 0, the maximum value state is selected as the next move.



What to do for part 2

You are ready to start working with classes AI and State.

Look at AI first. Like Random, it implements Player. But unlike Random, it attempts to play the game intelligently. But unfortunately it doesn't work yet. Class AI itself is implemented correctly, but it relies on another class, State, which is incomplete. Your job is to finish implementing State.

How is State used? Look at method AI.getMove. If the AI needs to provide a move for a given board, and no state has been calculated yet, it does 3 steps: make the state field the current state of the game, expand that into a game tree of the appropriate depth, and then run the minimax algorithm on that tree. If a state has already been calculated, then it was calculated by method AI.observeMove, which does the same 3 steps whenever the opponent makes a move (and it is the AI's turn). So the last thing AI.getMove does is get the preferred move from the calculated state. Because that state's player is the AI, its preferred move will be the minimax algorithm's estimate of the best move for the AI.

This process uses three methods of class State: expandUpTo, computeMinimax, and getPreferredMove. But none of these method are implemented. The second part of this assignment is to implement these three methods. Do not make any other changes to the code. Below is a brief description of these methods;

the code we provide has more information.

- expandUpTo(int depth) – expand this state into a game tree of depth depth
- computeMinimax() – compute the minimax value of this state's game tree
- getPreferredMove() – return the move that would be preferred by this state's player

For getPreferredMove, there might be several moves that are equally preferred. In this case, getPreferredMove should return the leftmost such move in order to provide deterministic behavior. To this end, we made the children field a *sorted* map from possible moves to the game trees result from those moves. Due to the map being sorted, leftmost moves will appear earlier as one iterates through the map. This will help keep your code for getPreferredMove simpler, but it will restrict what you can assign to children in expandUpTo.

More on Testing

Here are a few ways to test your project:

1. Visualizing the Game Tree
   a. Use StateLogger or MetaLogger to log the minimaxed game trees constructed by your State code. Check that the trees are expanded to the correct depth with the correct players, moves, and boards. Check that the minimax value is computed correctly for each state from the values of its children. Check that the AI makes the move corresponding to the leftmost preferred move of the state.
2. Comparing Moves
   a. We provide files a1.txt, a2.txt, a3.txt, a4.txt, a5.txt, and a6.txt. These are the results of
      - "java ConnectFour Console AI 1 AI 2 MegaLogger move1=a1.txt move2=ai2.txt"
      - "java ConnectFour Console AI 3 AI 4 MegaLogger move1=a3.txt move2=ai4.txt"
      - "java ConnectFour Console AI 5 AI 6 MegaLogger move1=a5.txt move2=ai6.txt"

      Compare with these moves when you believe you have the correct State implementation.

What not to change

Don't change any code besides the body of the five methods assigned in parts one and two of this assignment. All five methods you must implement are marked with a comment that says TODO. Note the specifications for these (and all) methods because your grade will be based partially on how well your code implements the specification.

Submit your files Board.java, Random.java, and State.java on the CMS by the due date. This means that any changes you make to any other files will not be incorporated into our evaluation.