Artificial Intelligence Player: Connect4

Karl-Frederik Maurrasse

August 10, 2014

1 Project Idea

To reach the goals for this class, I aimed to develop a fully functional, smart artificial intelligence player for the classic game Connect Four. This involves the following steps. First, establish the game logic by coding the rules and the gameplay for the Connect Four game. Secondly, develop a GUI to allow easy user-testing. Thirdly, code the artificial intelligence player itself that is capable of playing the Connect Four game effectively and efficiently. I chose Connect Four to demonstrate the work done using Artificial Intelligence techniques for the following reason. Because the game has very few rules, the logic implementing it could be coded fairly easily, thereby focusing the limited resources at my disposal on testing a plethora of AI techniques and developing an adequate Connect Four player.

2 Timeline

Given the size of this project, a detailed timeline will help guide development and assure completion of a working AI by the end of the semester.

September Game Logic & Design. Build the game from scratch, incorporating the actual rules and allowing human and ai players.

October - Mid November AI development. Experiment with a variety of techniques from the book (greedy local search, minimax) as well as custom techniques that incorporate several Computer Science and specifically Artificial Intelligence ideas.

Mid November - December Statistics and User Testing. Provide statistical comparison between various AIs developed, and determine how well the best AIs fare against human competitors.

3 Code Structure & Implementation

The project was developed from scratch using Python. Please find below the implementation details for all aspects of this project. A very modular design was preferred to allow for code clarity and feature isolation. The "logic" files contain the logic to accomplish a task (e.g one player's turn in a game of Connect Four), while the matching "functions" files contain the helper functions used by the "logic" files to accomplish said task. A very specific coding guideline was developed to make

*

clear dependencies between functions and files (e.g clear functions specs, knowing what functions call what functions,etc).

The game grid is represented as a $6X7$ numpy matrix initialized with 0s. The two players are assigned an identifier (1 or 2). Each coin placed by a player sets the corresponding matrix cell with that player's identifier.

3.1 Game Logic & Design

(gamelogic.py, gamelogicfunctions.py, gui.py)

The gamelogic file guarantees that the gameplay occurs following the rules of the classic game: two players take turns dropping a coin of their corresponding color (red or blue) down a valid grid column of their choosing. The game ends when a player successfully places four (4) coins of their assigned color in a row (horizontally, vertically or diagonally), or when there are no more valid moves to make (resulting in a tie). If a player attempts an invalid move, gamelogic simply requires the player to replay the turn.

The gamelogicfunctions file contains all the functions that check and assure that all the above rules are enforced during gameplay. For example, isRangeOutOfBounds and isMoveValid respectively return a flag notifying whether a given move is not within the bounds of the game grid or whether the chosen column is already full. Similarly, boardContainsWinner checks whether a player has won the game.

Every move received by gamelogic is then painted on the gui using the gui.py functions. Interactiveness is turned on to allow faster gui operations by allowing to simply add elements to the gui as opposed to redrawing it form scratch. The game grid is simply a $6X7$ grid, and every mmove played is represented by a red or blue coin (circle) painted on the appropriate cell of the grid.

3.2 AI Development

(aiplayer.py, aifunctions.py)

The various AIs developed are stored in the aiplayer file. Each artificial player is its own standing function for modularity and ease of use. Certain AIs are combinations of others, and as such call its parts in a particular order given the state of the game.

The file contains a gradient of artificial players, from the trivial random player to the best local move player to a purely defensive one, and much more. Artificial intelligence techniques employed include greedy best local search and look-ahead searching. Several metrics were tested for each.

For best local search, several characteristics were used to determine the "best" state, thus leading to differences in performance.

3.2.1 Board State Analysis

scoreBoard The first algorithm developed consisted of scoring a game board in the following way. First, a copy of the target board is made. The copy is exact except that the cells that already contain a coin are set to -1 so as not to influence the algorithm (unplayed cells are 0). Every unplayed cell in the copy receives a positive integer based on its "importance" in the original board. For instance, a "0" cell at the end of a 2-coin chain will receive a score of 5 while one at the end of a 3-coin chain will receive the logically higher score of 9. The more important cells are those with the higher scores.

boardContainsWinner A key feature of a board is whether it contains a win or a loss. This algorithm loops through a given state of a board and finds a winning chain of coins if one exists.

getSequentialCellsPlus Another feature to consider when analyzing the state of a game is the number of cell chains that could potentially lead to a victory or a loss. While identifying a loss is crucial, we may not be able to prevent it if we observe it too late. This function allows to identify all sequences of a specified length of coins in a given board state. This allows to identify opponent strategies early on and hopefully thwart them.

uselessSlotFilter Although we wish to block the opponent's sequences early on, we also want to create many opportunities for us to win by progressively increasing the number and length of sequences of our coins on the board. This algorithm identifies the cells that would contribute to such an offensive expansion, and rejects those that would not contribute to an offensive expansion. This allows for a very effective offensive strategy by building several win opportunities, hence increasing the probability of a win.

3.2.2 'lookAhead' AIs - Defense

Similarly, several look-ahead steps were considered in order to determine the more efficient version for this game.

The 'lookAhead' family comprises primarily defensive AIs. The AI plays defensively if such an action is flagged as necessary (conditionals vary between AIs). For example, should the opponent be able to win on his/her next turn, the AI's flag will go off, leading the AI to block the loss by playing where the opponent would have had to play.

lookAheadOne,lookAheadTwice, lookAheadThrice This family of players were the first to consider future board states in determining the best move to make. They simulate a certain number of steps (exact number is indicated in their name) and analyze the resulting board state in considering potential good moves to make in the moment. For instance, lookAheadTwice simulates its current move and the opponent's next move in order to pick a good slot now while lookAheadOne only considers the direct consequences of its current move choices (i.e the states to which its current possible moves will lead).

lookAheadOnePlus,lookAheadTwicePlus, lookAheadThricePlus Similar to the above AI trio, these players look into the future when choosing their next move. They differ form the aforementioned merely in the evaluation function employed in the analysis. While the above players call on "isBetterState", these next-level ones rely on "isSafeToPlay" and "getSequentialCellsPlus(3)" - the number of potential wins for each player.

3.2.3 randomOffense AIs - Offense

Offensive strategies appear in the randomOffense family. Contrary to their lookAhead partners, these players focus on improving their chances of winning by increasing their number of potential wins. They therefore employ a different board comparison algorithm by favoring boards that contain more winning opportunities for them. However, these players are not purely offensive. Should they determine that a good offensive move is not currently possible, they will default to "lookAheadTwice"'s move choice.

randomOffense, randomOffenseWithTwicePlus, randomOffenseOnewithTwicePlus

forwardEval

3.3 Testing & Statistics

3.3.1 user testing

3.3.2 automated testing

Here we pin AIs against each other in an attempt to rank their performance vis-a-vis each other. (test.py)