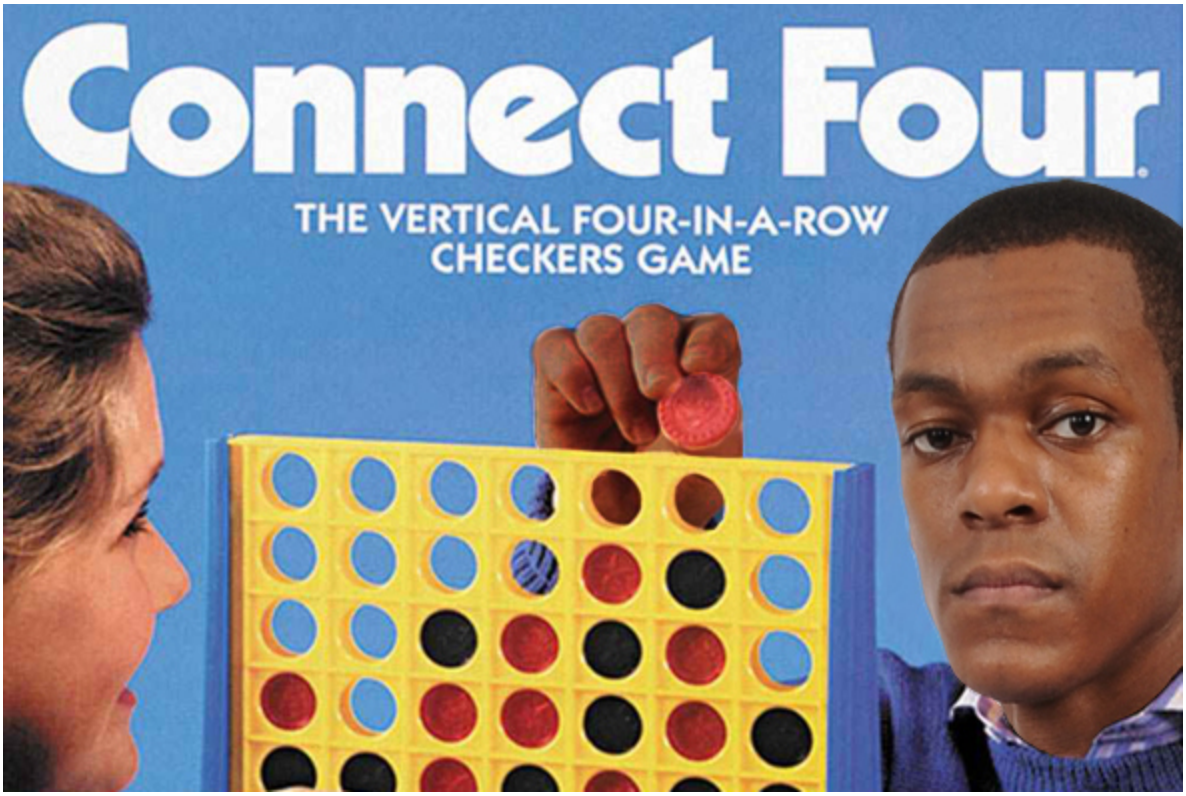
Connect Four Minimax AI Using Alpha-Beta Pruning

Ryan Kerr, Milan Ravenell, Evan Sandhoefner, Matt Tesfalul



http://img.bleacherreport.net/img/images/photos/002/350/927/rajon-rondo-connect-four_crop_north.jpg?w=590&h=394&q=75

Instructions and Video

First things first: download the .zip file and extract its contents. Then, if you have Python installed on your machine, just open up a command prompt and navigate to the newly extracted `code` directory. Once there, enter `python connectFour.py` to run the interactive game, or `python cpuFour.py` to watch two AIs duke it out. These commands will both compile and run the code, so no need to worry about "make". If you don't have Python installed on your machine, see the installation guide.

For more information, check out our video!

One important note on the structure of our .zip file: all the code that actually goes into the two final runnable programs is in the `code` folder. There is some additional (old) code, along with a bunch of other odds and ends, in the report folder. None of the code or files there will be called upon for `connectFour.py` or `cpuFour.py`. We thought we'd include it anyway, to give you a sense of things we tried along the way. Here's a brief overview of those files:

- `dist/` contains support files for .exe versions of `cpuFour.py` and `connectFour.py`, produced using py2exe. Unfortunately, Gmail wouldn't let us send a zipped .exe, so we took out the actual executables. They wouldn't have been fully platform-versatile anyway, because we got a warning about DLLs (screencapped in a .png in the dist folder) that we didn't look into resolving. `Setup.py` is also associated with our py2exe experimentation.
- `output/` contains experimental output from pyjs (a utility for basically converting Python code to JavaScript code) from our attempts to host the program online with a GUI. `Index.py` is also associated with our pyjs experimentation.
- `tk-gui.py` is from a brief experimentation with desktop GUIs.
- `evaluate_old.py`, `interface.py`, and `prototype.py` are old versions of `evaluate.py`, `connectFour.py`, and `ab_pruning.py` respectively.
- `tests.py` includes the testing we did to check our algorithms.
- `pseudo_signatures.py` is where it all began.

Overview / Motivation

Celtics legend and premier NBA point guard Rajon Rondo is a Connect Four fiend. He has been known to play with expert skill and speed, often playing against teammates, coaches and small children. We wanted to capture Rondo's competitive spirit and penchant for the game in our final project. From a computer science perspective, we wanted to explore artificial intelligence, which led to us picking the minimax algorithm using alpha-beta pruning. From this, our deadly connect-four-playing computer AI - Rondo - was born.

This report includes our current functionality and in-depth explanation of the algorithms used. We then detail our work process and look at past versions of our work.

Report

Functionality

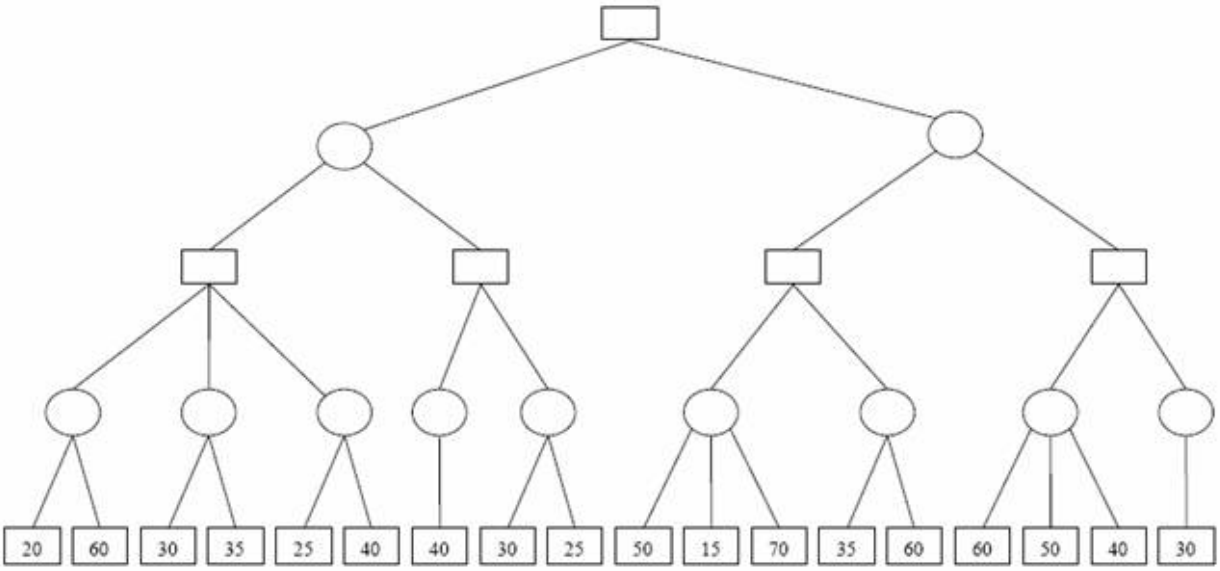
Note: A comparison of our final functionality compared to previous drafts is included below in the "Previous Work" section.

Front End

Our front-end functionality is contained in `connectFour.py` (for the primary interactive game implementation) and `cpuFour.py` (for the AI duel implementation). Comments in the code provide function-level descriptions, so we'll stick to the big picture here. The interactive game takes user inputs for who should go first and for how smart the AI should be (read: how many turns ahead the AI looks when evaluating potential moves). Then it takes a user input for each of the user's turns until the game ends. At each prompt, the user's input is purified to prevent unexpected-input errors. Function overview: `quit_if()` and `printBoard()` are helpers used throughout the game. `Init()` is called once, and then `move()`, `moveAI()`, and `movePlayer()` call each other until the game ends. The AI duel file is similar in structure. The user decides which AI goes first, and how smart each AI should be, and then sits back and enjoys the show.

MiniMax

The minimax algorithm is a standard of game artificial intelligence. It creates a decision tree for computer AI by looking many moves ahead and evaluating the eventual outcomes. Once the entire tree is constructed, the algorithm chooses the best move as detailed below:



http://aidblab.cse.iitm.ac.in/cs638/questionbank_files/image016.jpg

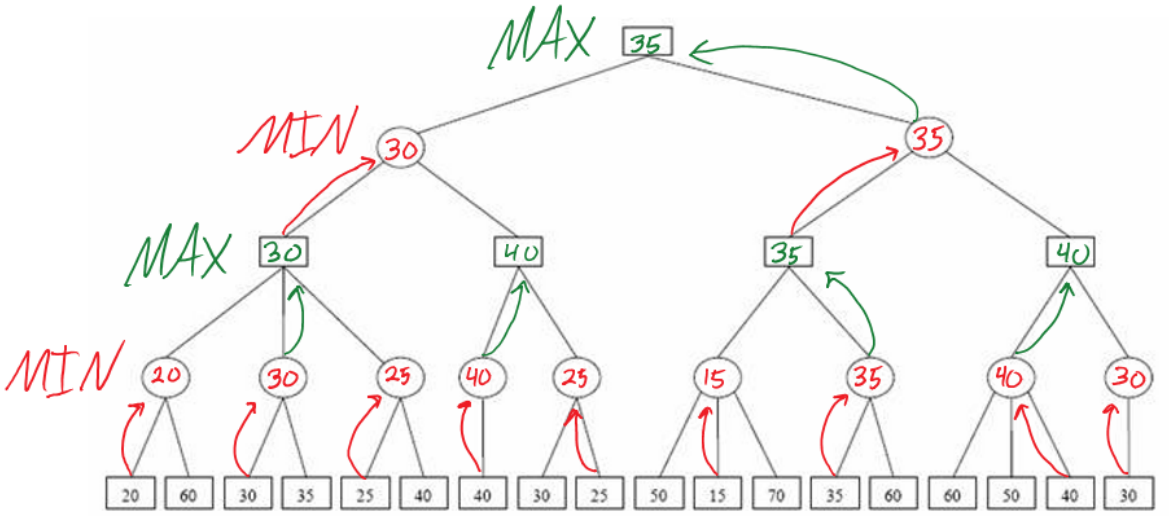
The computer, or “max player” (shown by squares) behaves as follows:

- Look at all possible moves/nodes one level below and choose the highest score possible

The opponent, or “min player” (shown by circles) behaves as follows:

- Look at all possible moves/nodes one level below and choose the lowest score possible
- *Note that “score” here is always assigned from the max player’s perspective. A board with a high score is good for the max player. This is why the min player picks the lowest score - it means it is bad for the max player and good for them.*
- *Also note that the min player is not actually a human making a move - it is the minimax algorithm predicting what the min move would be.*

This tree uses depth-first search and evaluates each decision from the bottom up:



In this decision tree we see that the MAX player's best score would be 35, by choosing the move on the right.

For our implementation we have significantly more possible moves and boards, which is why we implemented alpha-beta pruning (detailed below).

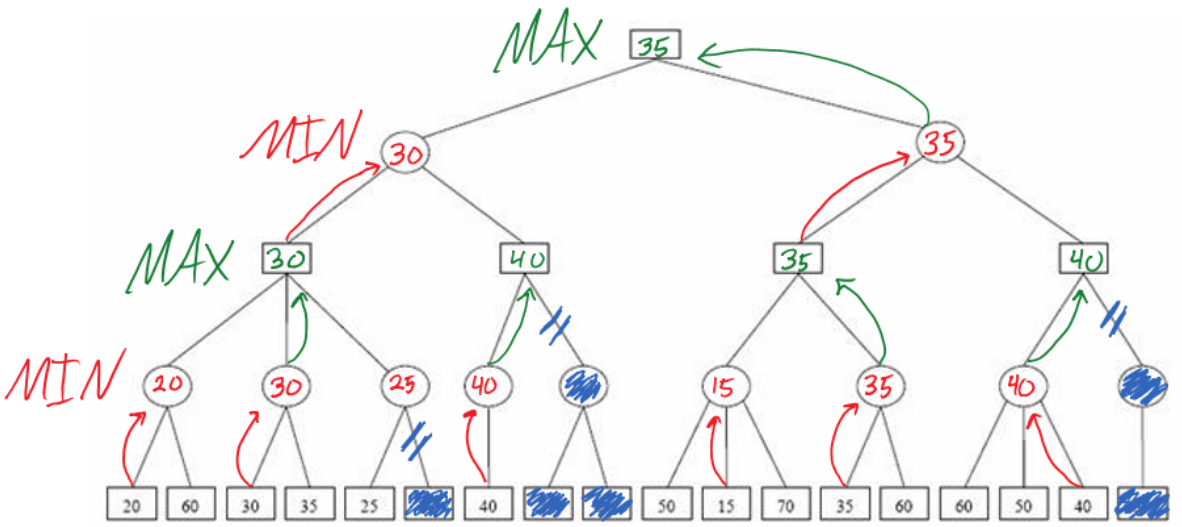
The minimax algorithm is implemented, commented and explained technically in the file `ab_pruning.py` .

Alpha-Beta Pruning

Alpha-beta pruning is a technique applied to the minimax algorithm to improve run time. Creating this decision tree is very computationally demanding - looking 5 moves ahead will have us create 7^5 = 16,807 boards. Alpha-beta pruning makes our algorithm faster and smarter by not evaluating nodes we know will not be chosen.

Each node is assigned an alpha and a beta value.
alpha = maximum lower bound of possible solutions
beta = minimum upper bound of possible solutions

The algorithm keeps track of these values after each node we visit, and knows that when alpha > beta the path will never be chosen.



Here we see that the addition of alpha-beta pruning to the algorithm eliminates many boards from even being created or evaluated.

For an even-more-detailed explanation of the algorithm, read this UCLA tutorial.

Alpha-beta pruning is implemented, commented and explained technically in the file `ab_pruning.py` .

Evaluation and Heuristics

- The AI heuristically evaluates potential future board outcomes using a series of functions which we divide into “board functions” and “threat functions.”
- In `board_functions.py`, we created a few helper functions to scan the board for four in a row of a specific state (red or yellow). These include one for vertical wins, one for horizontal, and one for each of diagonal-up and diagonal-down. We created a `possible_moves` function in order to pass the opening moves to our minimax algorithm to be evaluated. Lastly, we created another handful of functions to conduct small important tasks for the algorithm and testing.
    - `is_terminal` to check board for tie or win
    - `full` to check if board is full
    - `copy_board` to help reproduce the board
    - `game_won` declares the winner if there is one
- Next, the threat functions check for three-in-a-row threats. The algorithm distinguishes between a three-in-a-row threat with an open cell on each side (which is inevitably a “checkmate” situation) and a three-in-a-row threat with only one open side.

- Finally, `evaluate.py` utilizes a combination of these tests to assess the current board. Calling `evaluate` on the board will give it a score based on how good it is for the MAX (AI) player.
- The strategy we used to evaluate the board is as follows:
    - If MAX player won, assign score of infinity
    - if MIN player won, assign score of -infinity
    - Assign higher score to boards with more threatening AI positions using `threat` (how many places are there three-in-a-row?)
    - Assign lower score to boards with more threatening MIN positions using `threat`

The function-specific code is explained in comments in `board_functions.py` and `evaluate.py` .

Testing

- We've included tests.py, which is a file composed of all of the tests that we performed on our functions. In order to test, we created a certain number of boards that satisfied a certain evaluation, then tested our evaluating functions on that board to see if they returned the desired outputs.

Project

How good was your original planning?

Our planning was very effective. We were able to generally follow the direction of our checkpoints give or take a few schedule hiccups here and there. The specs allowed us to foundationally set up an implementation schedule and brainstorm potential needed functions. Which also proved to help out splitting the work efficiently.

How did your milestones go?

We were extremely proud with our progress on our milestones. As we developed and polished the minimax algorithm with alpha beta pruning and strengthened our evaluation heuristics, the project's different parts started to tie together nicely. There was no bigger milestone than having our AI beat our friends and us practically every time.

What was your experience with design, interfaces, languages, systems, testing, etc.?

We went with the traditional, yet slick, modest ASCII terminal interface to display the board game. Board consisted of 2-D array which allowed for easy manipulation. The minimax algorithm utilized a tree structure with depth determined by how far user chooses to look in future or the difficulty.

What surprises, pleasant or otherwise, did you encounter on the way?
From a technical perspective we were surprised to find that lists were mutable in python. Assigning list1 = list2 and then modifying list 1 would also modify list2. This was very different from Ocaml.
From an algorithmic standpoint, we were surprised by the fact that the minimax algorithms itself was the not the brunt of the coding work. While it was the main focus conceptually, it was actually very easy to implement using abstraction. The most difficult challenge coding-wise was creating all of the functions related the connect four board and gameplay.

What choices did you make that worked out well or badly?
Choosing to split up the work early and having meetups a few times a week to decompress the material each of us added proved to be extremely helpful and prevented any problems or misunderstandings.

What would you like to do if there were more time?
We would like to implement a GUI if we had more time. We were able to make rondo as complex as we wanted using minimax and alpha-beta pruning, but the game is still not super approachable to outside users.

How would you do things differently next time?
If we were to do this another time we would try different languages to cut down on run time. We would consider using Java or C to allow us to look even further ahead and create a smarter Rondo.

What was each group member's contribution to the project?
Initially, we split the work into 3 categories: the algorithm, functions that evaluate and manipulate the board, and the user interface. Ryan worked on implementing the minimax algorithm, Matthew and Milan worked on the board functions, and Evan created the interface. After we finished the core functionalities, the next steps were to implement alpha-beta pruning, and the functions that evaluate the threatening positions on the board and assigns the heuristic. Because Ryan had the most experience with the algorithm, he and Evan worked on alpha-beta pruning, and Milan and Matthew worked on the evaluate functions, since it was closely related to the board functions.

What is the most important thing you learned from the project?
The most important thing we learned was the aspect of working on a coding project with a team. We gained experience splitting up the project most effectively, having the team members work on their respective project, then bring it all together. We also learned of the importance of continuously making sure our code is clean and readable, not only for the users, but for our team members and future selves. We also gained a

lot of experience using the GitHub out of the context a problem set. Additionally, we gained experience in beginning a project from scratch, from designing how to most effectively design our project, to selecting the programming language that would most efficiently implement our idea.

Functionality we included in our final implementation is highlighted in green
Functionality we left out is highlighted in red

Project Proposal

Ryan Kerr, Milan Ravenell, Evan Sandhoefner, Matt Tesfalul

Overview

Connect four is a classic board game in which two players make sequential moves to try to establish board control. The game is played on a 7x6 board where players drop in pieces to try to get four of their own pieces in a row. We want to create a program that uses artificial intelligence to play connect four versus a human. This program would evaluate the game state before every move and make its own move to maximize the likelihood of winning. The AI would be unbeatable unless the human goes first, puts his/her piece in the center column, and plays perfectly.

We plan to use the Minimax algorithm to create this AI. The minimax algorithm evaluates the board state before every move and makes a counter-move based on the chance of winning the game. The technical summary for the minimax algorithm is outlined below in the "Technical Specifications" section.

From a learning perspective we want to come to understand the minimax algorithm and how it relates to AI in general. We will understand how to implement this algorithm and how to implement a knowledge-based approach for decision making. In all likelihood we will not use OCaml, and will therefore understand the differences between other languages in how they relate to AI and this algorithm in particular.

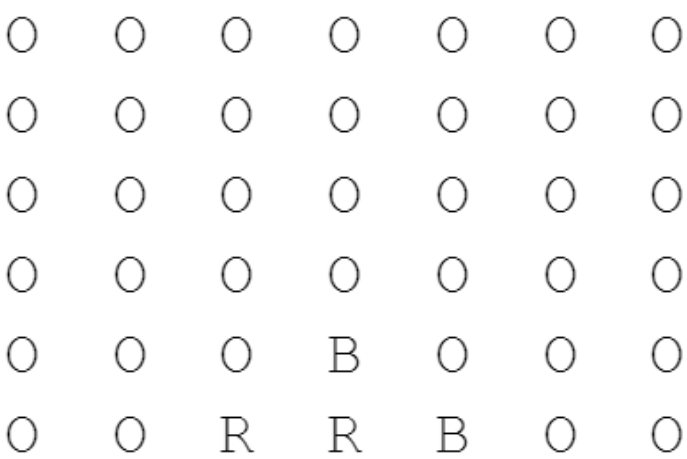
From an end-product perspective we will create a program that:

- Allows for human vs. computer interaction to play connect four
- Implements the artificial intelligence using the minimax algorithm
- Represents the connect four game board in a human-readable way

Features

- Core features:
  - The absolute bottom line in terms of functionality is a command-line program that takes in a textual representation of a Connect Four board state and prints the optimal next move. We may accomplish grid-cell identification by assigning arbitrary letters to rows and numbers to columns, so a given cell on the board could be designated, for example, "A3".
  - One step beyond that is a program, still at the command line, that allows the user to play through an entire game of Connect Four. This would

entail a series of prompts. First, ask the user if they would like to go first or second. Then, for each of the user's turns, ask for a grid location for their move. For each of the AI's turns, compute the optimal move and proceed. Print an ASCII representation of the board after each move. For example:



In the representation above, R and B stand for the classic red and black discs. Moving on: after each turn, check for a winner and print "Game over: _____ wins" if applicable. After game over, ask if the user would like another round.

- Cool extensions:
  - We expect the first cool extension to be substantially easier than the ones below. In a nutshell, we would like to implement easy, medium, and hard modes that the user could choose from before starting a game. This is a little counterintuitive, because our pride and joy will be the perfect-move algorithm, and we're basically watering it down for this feature. But it would certainly enhance the user experience, because the fully optimal AI is going to be very difficult to beat if the user goes first, and impossible to beat if the user goes second.
  - The second cool extension shifts gears from bread-and-butter functionality to aesthetics. We would like to implement a GUI that allows the user to see the board in a more realistic representation, rather than ASCII. The user could then click on a grid cell to execute a move, rather than operating through the cumbersome "A3" identification process.
  - Now we're really getting into best-case scenario territory: if all of the above works out perfectly and quickly, we will attempt to host the program online. This is opening up a whole new can of worms, and the specific technical way that we host the program will depend on which language we end up implementing it in. For example, if we implement in Python, we'll check out Django. Moving Java code from desktop to the

web seems trickier, but we are looking into JavaWS and applet classes. At the other extreme, if we implement in JavaScript, web deployment will be a no-brainer.

Technical Specifications

We plan to split our project into three components:

1. Board: When first coding our project, we will code our board into a grid, designating position as a coordinate that we can access in the terminal. This user will type "Move: A1" when he wants to place a piece on the board. This will update the current board, which will be inputted into the program in order for the computer to determine its best move. When the AI computer makes a move, it will take into account the current board and run the minimax/heuristic algorithm to update the board with the best move. Every time the board is updated it checks the grid for any four in row. If there is it will detect the color and declare the winner. Otherwise the user will be prompted to move again. Possible functions:
    a. Function that prompts the user to select a position on the board for his next move.
    b. Function that calls the update board component and adds the user move to the existing board.
    c. Function that calls the strategy component to get the AI computers move.

2. Strategy/Algorithm: The second component allows the computer to determine the best move possible given a grid/board. The possible options will be split up into a minimax data structure/tree, and assign each of the possible outcomes a 1 if the computer will win, 0 if there is a draw, and -1 if the user will win. Then, the program will take the path with highest probability or number of favorable outcomes. Possible functions:
    a. Function that builds up a tree of the possible paths the computer can choose and the outcomes of those paths.
    b. Function that takes in the tree of possible outcomes, ranks the outcomes and paths based on favorability, and selects the most favorable next move.
    c. Function that calls the update board component.

3. Updating the board: This component will take in the moves from both the user and the computer and updating the board on the grid interface and on the back end. Possible functions:

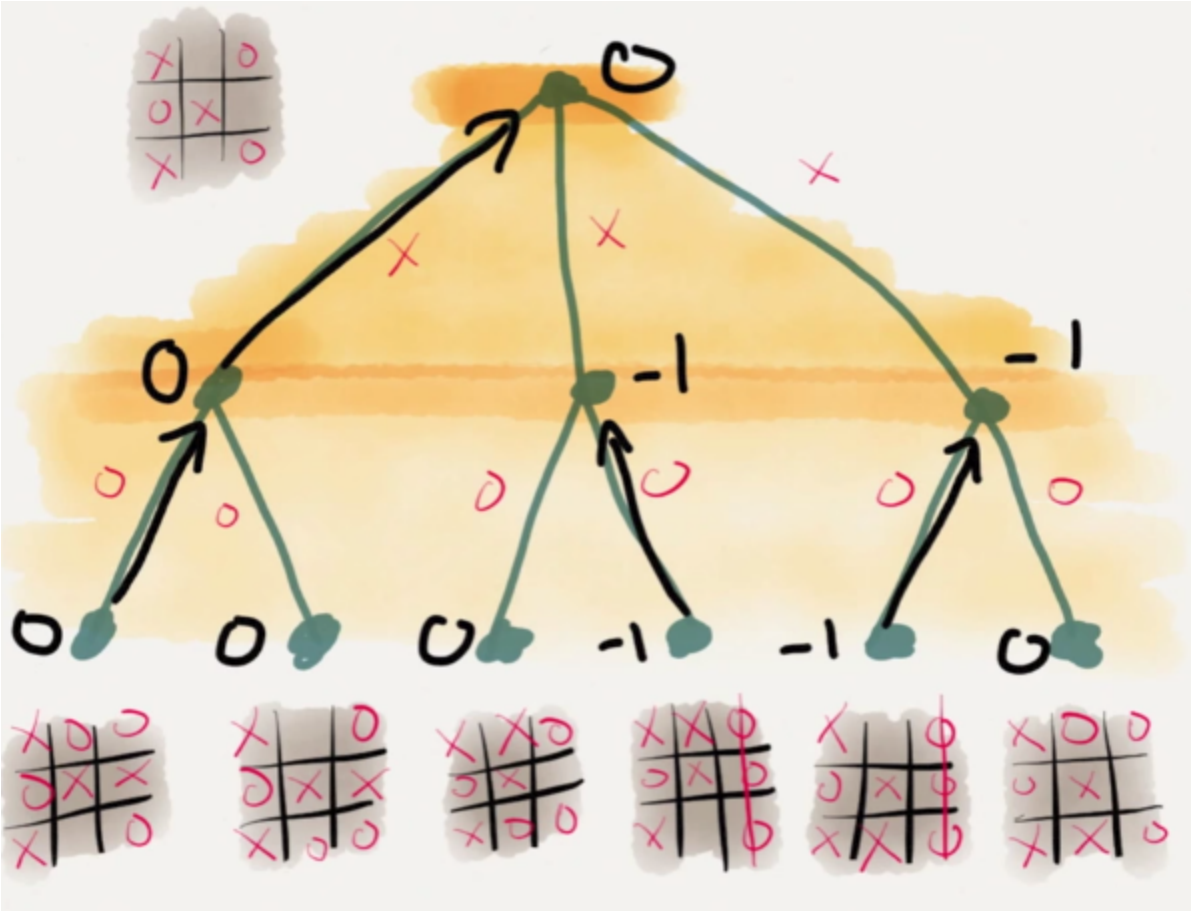
a. Function that takes in a move (either computers or users) and updates the existing board.

b. Function that checks if there are four pieces of the same color in a row horizontally, vertically, and diagonally.

Helper functions:

· Function that prints out the board in the terminal for the user to know the existing positions that have been taken.

· Functions to test the effectiveness of the move and the tree of possible outcomes



Next Steps

Priority 1: (4/12 deadline) Set up coding environment

- We will use a code.seas repository for now, and transfer everything to a github repository once the project is completed.

Priority 2: (4/15) Decide which language/languages we will use.

- Likely Java, will use quora, google and stackoverflow to explore this.
- Explore options for coding frameworks for project.

Priority 3: (after language decided) Assign coding responsibilities

- Assign each person to a specific coding responsibility for those that are mentioned in the technical specification.

Resources:

conspiracy-number search and Allis algorithm:
http://web.mit.edu/Sp.268/www/2010/connectFourSlides.pdf

2048 Minimax algorithm:
https://www.youtube.com/watch?v=-1utA60TNaM

Minimax applied on connect four:
https://johannes89.wordpress.com/2014/02/09/teaching-a-computer-to-play-connect-four-using-the-minimax-algorithm/

Functionality Checkpoint

Ryan Kerr, Milan Ravenell, Evan Sandhoefner, Matt Tesfalul
CS 51 Final Project Specification

Progress:

Over the past week, we began to implement the functions that interact with our game board, and setup up the core functionality and user interface. We created the board as a 2d matrix of strings with 6 rows and 7 columns. Each spot on the board is either an asterisk, which signifies an empty spot, an "R" which signifies a red piece, or a "Y" which signifies a yellow piece. In order to interact with our game board, we implemented 4 functions, "is_terminal," "possible_moves," "go_next," and "game_won." Is_terminal takes in a board, and a string of either "R" or "Y," depending on who's pieces we want to check for 4 in a row. In order to implement this function, we created helper functions that check for a specific winning case: horizontal, vertical, and diagonal. Is_terminal also checks to see if the game board is completely full with no 4 in a row, at which point the game is called a tie. Possible_moves takes in a game board, and returns an int list of the columns that are not full. Go_next takes in a game board, an int, and a string of either "R" or "Y." It will insert the piece into the lowest possible spot in the column that is specified by the int. We tested our code by generating specific game boards, and testing our functions on those boards. So far, all of our tests have passed.

For the user interface, we implemented 5 key functions: init(), printBoard(), move(), moveAI(), and movePlayer(). Init() prints an introduction to the game with instructions, asks the user if (s)he wants to go first (re-prompting for invalid inputs), prints the empty starting board, assigns to the global bool playersTurn based on whether or not the user wanted to go first, and calls move(). Move() checks whether there is a game-over (win or tie). If so, move() prints a game-over message and exits the program. If not, move() checks the value of playersTurn, negates playersTurn, and calls either movePlayer() or moveAI(). MovePlayer() asks the user for a column (re-prompting if the input isn't valid), alters the board in memory, prints the new board, and calls move(). MoveAI() prints a "thinking" message while the program sleeps for a second, then reassigns the first available cell (starting at the bottom-left, moving up then right) to "R", prints the new board, and calls move().* PrintBoard() simply prints an ASCII representation of the current board to the terminal window.

*The next major hurdle here is to make moveAI() use the go_next(_, _, _) function to make smart moves rather than dumb ones.

The abstract flow for the minimax algorithm is implemented fully, but the implementation is still not complete because of errors in our board functions. We also have a function that is used to evaluate how good a board is (because we cannot go all the way to all possible boards), and it currently assigns low scores to

boards where the opponent has three pieces in a row. Neither minimax nor evaluate are included in the current version of connectFour.py because there is lots of debugging to be done.

Problems:

Our problems so far have stemmed from learning the functionality of python and determining how to best represent our board. We began using a class for the board and abstracting the board functions to a mutable board variable, but then decided to take out the class functionality, and simply pass in a board for the arguments of our board functions. This is mainly because we do not fully understand how class implementations work in python, so we are willing to go back to using classes if it is indeed beneficial.

Another major problem is dealing with mutability in python. Lists in python are mutable, and using list1 = list2 does not create a separate copy of list2, like it would in Ocaml. This has caused problems in many of our functions.

Teamwork:

We split the work with Evan working on user interface and tying all the pieces together, Ryan working on the AI, while Milan and Matthew worked on the board functions and creation of the board. We initially had trouble making sure all of our functions interacted with each other as planned, but we made sure to follow the function signatures that were specified and have had fewer problems since.

Schedule:

We plan over the next couple of days to tie the functionality of each of these parts together. After getting the minimax function to work with board implementation, we plan on adding alpha beta pruning to expedite the process.

4/26 - ASCII version of game working, using simple minimax
- Board Functions working entirely (Milan & Matt)
- Minimax and evaluate debugged (Ryan & Evan)

4/30 - Increase minimax complexity (Ryan)
- Alpha-beta pruning (Milan & Matt)
- GUI representation (Evan)

4/31 - Screencast & Submission (all)