Program 2 - Game Playing

Jordan Stein

Introduction / Problem Statement

The purpose of this program is to write an artificial intelligence that simulates games of Connect-Four against multiple different artificial intelligences. Two racket programs are provided. One racket program contains a naive AI that makes random moves and and the other racket program acts as a driver to simulate a game between two different game playing programs.
The objectives of the assignment are:

- Read move data in via standard input from the driver program. Write move data to standard output to the driver program.
- Play a valid game of Connect-Four, in particular make valid moves in response to the current game state.
- Consistently beat the naive program that generates random moves.
- Use the driver program to match your algorithm vs. other students algorithms.

Contents

Player Program Design

In designing the artificial intelligence to play Connect-Four, my approach was to break up the problem into many different sub-problems to achieve the overall goal of making optimal moves. I initially wrote the naive random-move generating AI into my own script before I faced it against the naive AI script in racket, so a few unnecessary functions are added for local debugging. This was to optimize my smart AI before working with the driver program.
The subproblems listed as follows:

- Adding moves to the current game state (For heuristic evaluations)
- Removing moves to the current game state (For heuristic evaluations)
- Drawing the current game state in a grid (Debugging/Visualization purposes)
- Checking win conditions
- Heuristic evaluation function (Calculates/returns the “worth” of placing a potential move)
- Parsing through JSON data sent in by driver program

Adding Moves:
I wrote the addMove(grid, move, player) function that takes the current game state, and adds a desired move to the state. It first checks if the move being played is the first move on that column. If it is, the move will be placed at the end of the list. Otherwise, it iterates through the list to find the next available spot for a move. Error checking is done prior to calling addMove to ensure a move will not be added if the column is full.

Removing moves:
I wrote the removeMove(grid, move, player) function to remove a previous move that was added. This is important because the heuristic function adds a move, evaluates its “worth”, and removes the move to calculate which move was optimal. This function finds the most recent move that was played in the column given, and re-initializes it to 0 to indicate that the move is now empty again.

Drawing the current game state:
The purpose of the drawGrid(grid, height, width, moveCount) function is to draw the current game state prettily for local debugging purposes. This function is not necessary when the driver program is operating the algorithm. The function prints a visualization of the current game state to the user to see how the game is played out move-by-move from the naive AI and the smart AI.

Checking win conditions:
The checkWin(grid, height, width) function traverses through the grid to determine if either player has obtained a win condition. It first checks if any vertical win conditions are met, then horizontal wins, then diagonal wins to the right and left. If a player has won, the function returns True and the winning player.

Heuristic evaluation function:
The function heuristic(grid, move, player) is the main driver of the smart AI. It first temporarily adds a move, then compares the count for every consecutive tile claimed by that player to a local maximum value. It first checks how many consecutive vertical tiles are present, then horizontals, and diagonals in both directions. The counter keeps incrementing for each direction as long as the player's tiles are consecutive. After the overall maximum from each counter is obtained, the temporarily added move is removed and the maximum heuristic value is returned. This function is called for every potential next-move in the current game state.

JSON parsing:
The readJson(data) function breaks up the json data that is read in and stores it into its own variables. This is mostly important for obtaining the new state of the grid after the script I am playing against has made a move.

Main:
I have two versions of the main driver. The first version is for reading/sending the moves calculated to and from the driver program, and the other is for local play.
In the version that utilizes the racket driver program, it first reads a JSON representation of the current game state from standard input, then iterates through each move to find the maximum heuristic value obtained from every possible next-move. The move that has the maximum heuristic value is stored in moveJson[“move”] that will eventually be outputted back to the driver program. If the smart AI does not have a win condition, it check if the opponent is getting close to a win condition before sending the best move to the driver program. To do this, the heuristic value for the opponents possible next moves are calculated. If the heuristic value is 3 or greater, it will update the best move to the opponent's best move to block their win path. Afterwards, the move being passed is translated to JSON and sent to the driver program. The standard output and error is then flushed so it will be ready to read and output the next move given by the driver program.

The second version of the main driver is used locally in the same python script. This was used for initial building/debugging of the smart AI. It operates similarly to the version that works with the racket driver program, however instead of reading input from the driver it locally generates random moves for the naive AI and plays it. When ran, it shows a game being played step by step or simulates up to 100000 games and displays the win-rate for each AI depending on user input.

Player Program Implementation

The following pages show the source code for my smart AI that competes against other AI’s using the racket driver program.

```
# Jordan Stein
# connectfour.py
# Python 2.7

# This program simulates a game of connect four using the provided driver program in racket.'
# Information from the grid is read in from the driver, and outputted back to the driver.
# My AI script will compete against another AI script as indicated in the driver program.
# Each smart move is calculated using a heuristic function to calculate optimal moves.

import sys
import json
import copy
import random


# Adds desired move to the grid
def addMove(grid, move, player):
    if grid[move][len(grid[move])-1] == 0: # if its the first move, store it
        grid[move][len(grid[move])-1] = player
        return
    else:
        for x in range(0,len(grid[move])-1): # else look for next available spot
            if (grid[move][x+1] != 0):
                grid[move][x] = player
                return


# removes desired move to the grid (for smart AI heuristic calculation)
def removeMove(grid, move, player):
    for x in range(0, len(grid[move])):
        if (grid[move][x] == player): # find most recent move, make it empty
            grid[move][x] = 0
            return


# generates a random number from 0 to height
def randomMove(height):
    return random.randint(0,height)


# prints the grid to the output prettily
def drawGrid(grid, height, width, moveCount):
    print "--- Move",moveCount,"---"
    for x in range(0, height):
        row = ""
        for y in range(0, width):
            row += str(grid[y][x]) + " "

        print row.replace("0"," ")
```

```python
# traverses through the grid to determin if any win conditions are met
def checkWin(grid, height, width):
    #check vertical win
    for x in range(0, width):
        for y in range(0, height-3):
            if grid[x][y] == 1 and grid[x][y+1] == 1 and grid[x][y+2] == 1 and grid[x][y+3] == 1:
                return True, 1
            if grid[x][y] == 2 and grid[x][y+1] == 2 and grid[x][y+2] == 2 and grid[x][y+3] == 2:
                return True, 2

    #check horizontal win
    for x in range(0,width-3):
        for y in range(0, height):
            if grid[x][y] == 1 and grid[x+1][y] == 1 and grid[x+2][y] == 1 and grid[x+3][y] == 1:
                return True, 1
            if grid[x][y] == 2 and grid[x+1][y] == 2 and grid[x+2][y] == 2 and grid[x+3][y] == 2:
                return True, 2

    #check diagonal win
    for x in range(0, width-3):
        for y in range(0, height-3):
            # diagonal right
            if grid[x][y] == 1 and grid[x+1][y+1] == 1 and grid[x+2][y+2] == 1 and grid[x+3][y+3] == 1:
                return True, 1
            if grid[x][y] == 2 and grid[x+1][y+1] == 2 and grid[x+2][y+2] == 2 and grid[x+3][y+3] == 2:
                return True, 2
            # diagonal left
            if grid[x][y] == 1 and grid[x-1][y-1] == 1 and grid[x-2][y-2] == 1 and grid[x-3][y-3] == 1:
                return True, 1
            if grid[x][y] == 2 and grid[x-1][y-1] == 2 and grid[x-2][y-2] == 2 and grid[x-3][y-3] == 2:
                return True, 2
    return False, 0
```

```
# heuristic function.
# calculates/returns the "worth" of placing a potential move on the grid (smart AI)
def heuristic(grid, move, player):
    addMove(grid, move, player) # temporarily add move

    maximum = 0
    counter = 0

    #check vertical max
    for x in range(0, width):
        counter = 0
        for y in range(0, height-3):
            counter = 0
            if grid[x][y] == player:
                counter += 1
                if grid[x][y+1] == player:
                    counter += 1
                    if grid[x][y+2] == player:
                        counter += 1
                        if grid[x][y+3] == player:
                            counter += 1
            if counter > maximum:
                maximum = counter

    counter = 0
    # check horizontal max right scan
    for x in range(0, width-3):
        counter = 0
        for y in range(0, height):
            counter = 0
            if grid[x][y] == player:
                counter += 1
                if grid[x+1][y] == player:
                    counter += 1
                    if grid[x+2][y] == player:
                        counter += 1
                        if grid[x+3][y] == player:
                            counter += 1
            if counter > maximum:
                maximum = counter

    counter = 0
    # check horizontal max left scan
    for x in xrange(width-4, 0, -1):
        counter = 0
        for y in range(0, height):
            counter = 0
            if grid[x][y] == player:
                counter += 1
                if grid[x+1][y] == player:
                    counter += 1
                    if grid[x+2][y] == player:
                        counter += 1
                        if grid[x+3][y] == player:
                            counter += 1
            if counter > maximum:
```

```
                maximum = counter

    counter = 0
    # check diagonal max
    for x in range(0, width-3):
        counter = 0
        for y in range(0, height-3):
            counter = 0
            # diagonal right max
            if grid[x][y] == player:
                counter += 1
                if grid[x+1][y+1] == player:
                    counter += 1
                    if grid[x+2][y+2] == player:
                        counter += 1
                        if grid[x+3][y+3] == player:
                            counter += 1
            if counter > maximum:
                maximum = counter

            counter = 0

            # diagonal left max
            if grid[x][y] == player:
                counter += 1
                if grid[x-1][y-1] == player:
                    counter += 1
                    if grid[x-2][y-2] == player:
                        counter += 1
                        if grid[x-3][y-3] == player:
                            counter += 1
            if counter > maximum:
                maximum = counter

    removeMove(grid, move, player) # remove the added move after heuristic calculations
    return maximum

# parses through the json data passed in by driver program.
def readJson(data):
    global grid, height, player, width
    grid = data["grid"]
    height = data["height"]
    player = data["player"]
    width = data["width"]
```

```
# main driver starts here

moveJson = {
    "move": 0
}
data = json.loads(raw_input())
grid = data["grid"]
height = data["height"]
player = data["player"]
width = data["width"]

playerME = 0
playerOpponent = 0

# we are the opposite of whichever player is initially passed.
if player == 1:
    playerMe = 1
    playerOpponent = 2
else:
    playerMe = 2
    playerOpponent = 1

win = False
while(not win):

    hMax = 0 # heuristic max
    for x in range(0, width):
        if grid[x][0] == 0: # if the move is valid
            hVal = heuristic(grid, x, playerMe)
            if hMax < hVal: # once we found the maximum heuristic value
                hMax = hVal # save it as the best move

                if (hMax == 0): # if it is the first move
                    moveJson["move"] = 3 # play in the middle
                moveJson["move"] = x

    # if we do not have a win condition, lets check if the opponent is close to one
    blocking = False
    if hMax != 4:
        for x in range(0, width):
            if grid[x][0] == 0: # if the move is valid
                opponentHeuristicVal = heuristic(grid,x,playerOpponent)
                if opponentHeuristicVal >= 3:
                    #bestMove = x  # if opponent is close to a win condition, block their path
                    moveJson["move"] = x
                    blocking = True
                if opponentHeuristicVal == 4: # if opponent has a win condition, crush their dreams
                    break

    if playerMe: # if we are first and it is the first move
        if not blocking:
            firstMove = True
            for x in range(0, width):
                if grid[x][0] != 0:
                    firstMove = False

            if firstMove:
                moveJson["move"] = 3  # play in the middle


    sys.stderr.write(str(grid))
    sys.stderr.write("\n")
```

```
jsonMove = json.dumps(moveJson)

print (jsonMove)
sys.stdout.flush()
sys.stderr.flush()

win, winner = checkWin(grid, height, width)
if win:
    sys.exit()

readJson(json.loads(raw_input()))
```

Expected Results

It is expected that the smart AI will beat the naive AI the large majority of the time because the naive AI is making random moves whereas the smart AI is making optimal moves while blocking the naive AI from winning. However, due to probability it is possible for the naive AI to win under rare circumstances.
When playing the smart AI against itself, I expect whoever plays first to win.
I do not know what to expect when playing my smart AI against other players smart AIs. I should be able to win against some and lose against others. I am most excited to see these results because it shows whoever has the most optimized AI.

Results

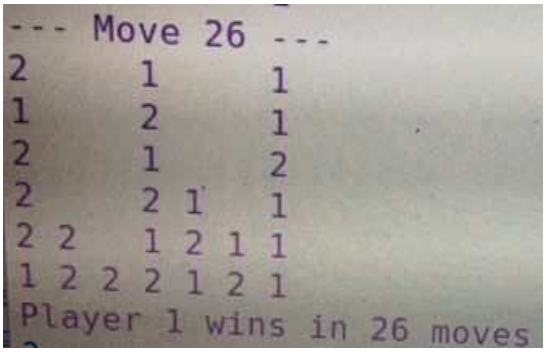
In the local script, the user may input the amount of games they wish to simulate against the naive AI. The naive AI always plays first in the local script.
Below is the results for simulating 100,000 games of the smart AI vs the local naive AI:

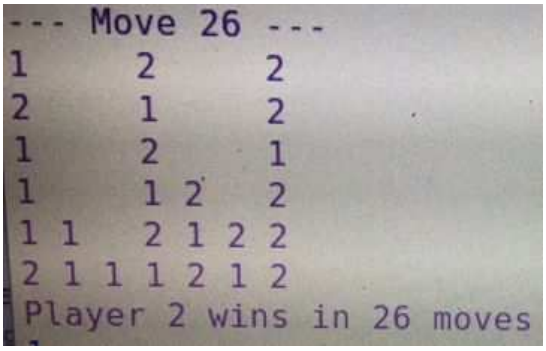
```
Please input your option (1/2):
1 Simulate one game of connect four.
2 Simulate many games, view results.
2
How many games would you like to simulate: 100000
Player 1 (naive AI) win count: 4050
Player 2 (smart AI) win count: 95950
smart AI winrate =  95.95 %
```

Below is the results of my smart AI (Player 2) vs. the racket naive AI using the driver program:

```
A total of 20 games were played.
A total of 0 games were draws.
Player 1 won 0 games total.
Player 2 won 20 games total.
Player 1 won 0 games when they moved first.
Player 2 won 10 games when they moved first.
Player 2 won every game.
```

Below is the results of my smart AI vs. itself using the racket driver program:
When playing my smart AI against itself, whoever plays second always wins in 26 moves. The game is the exact same in both situations. The ending result is interesting as it builds up on both sides, and eventually wins on a left-diagonal play.





I tested my smart AI against other students smart AIs using the driver program. The two contestants are Lewi and Jonathan.
When playing against other players scripts, the same two games are always played because the AIs are programmed to play the same moves when they are given the same exact states. Because of this, simulating multiple games is meaningless because the same exact games are constantly being played over and over again depending on who the driver program has chosen to play first.

Against Lewi:
My smart AI wins when I play first and loses when I play second against Lewi's smart AI.
When my smart AI plays first, I win in 27 moves.
When Lewi's smart AI plays first, I lose in 21 moves.

Against Jonathan:
My smart AI wins against Jonathans smart in in both cases.
When my smart AI plays first, I win in 37 moves.
When Jonathan's smart AI plays first, I win in 40 moves.

Conclusion

I have successfully implemented a smart AI that can play Connect-Four while making optimal moves. All of the main objectives are met with my smart AI. The smart AI can read in and output moves from the racket driver program, play a game of Connect-Four by making valid moves, consistently beat both the local and racket driver naive AI with a 96% win-rate in the local program, a 100% win rate vs the naive racket program, and match against other students smart AIs.