Problem

Write a program to play Connect Four. Read in the game state in JSON format through standard input, determine the best move and write it to the standard output.

Design

Bash Script (main.sh)

This file gets the arguments from the driver program and passes them along to *main.swift* which actually implements the move. (Was unable to directly call *main.swift* from the driver program because standard output would not complete until the program completely finished, resulting in a broken pipe when the driver program tried to get the 2nd move from my program)

GameState Class (main.swift)

Class that holds the information for a particular game state. The class holds its children states, its parent state, the move made to get to this state, etc. Functionality wise, this class has a function (findWinOrBlock) that determines if there is a move that will win the game for the current player, or block a win from the other player. It has a function (generateChildren) that determines all of the possible moves that could be made from this state and sets the children of the state. The last main function in the class is calculateLeafValue, which determines the value of the a leaf node by setting the states value at 100 if the current player won, -100 if they lost, and stays 0 if no winner is determined. The other function is just printState which helped me when building the program.

Main Functions (main.swift)

The main procedure uses five main functions to help implement the AI. The first one is called *currentMove*. This determines which player is making the move, not the players number, but if they were the first or second person to go. The variable returned for this function is used when building the tree to help determine whether it should take the min value or max value of its children. The next function is called *calculateWinningMoves*. This function calculates all the winning moves to be used in the class's *findWinOrBlock* function. The next function is *buildTree.* This is the meat of the entire program and is the implementation to the min-max algorithm. It is first called with the nodeState read in from the driver program, and then it loops through each of the nodeState's children and

recursively calls itself for each child node until it reaches the max depth. Once it reaches the max depth, it calculates that leaf's value. Once all of the children of a node have been looped through, depending on the current move given by *currentMove,* the current node gets the value of either the min or max of its children's values. This is my implementation of the min-max algorithm instead of going through the same tree twice (once to build and once to evaluate), I just build and evaluate simultaneously. The next two function are called *isMovePossible* and *findFirstOpenColumn.* These functions are used to double check whether the move given by *findWinOrBlock* or *buildTree* is a legal move or not. If the move is not possible, then the program finds the first open column and makes that its move.

Standard Functions (main.swift)

These three functions, *getData, sendData, writeError* handle the passing of information to and from the driver program. The implement the standard input, output and error, respectively.

Main Procedure (main.swift)

This is the main procedure of the program that utilizes all of the code listed above. First, the program gets the information from the standard input and sets some variables. Then it calls *currentMove* to determine which move it is on. Once that is complete the procedure continues an calculates the winning moves, followed by creating the first node. After the first node is set up, the program calls *findWinOrBlock* to first determine if there is a move that will win them the game or block the other player from winning the game. If one of these moves is present, it will skip the building of the tree and implementation of the min-max algorithm and just return that move. If there is no win or block move, the program will proceed to build out the tree and determine the best move using min-max. After a move has been determined (via *findWinOrBlock* or the min-max implementation) the main procedure double checks and makes sure the move is valid. If the move is not valid it will find the first possible move in the children and return that as the move. The program then writes to the error file the state and the move to be made, and then finally makes the call to send the driver program the move that was determined.

Source Code

main.sh

```
#!/bin/bash

while true; do

    read state
    a=$(Connect4/Connect4/./main.swift $state)
    echo "$a"

done
```

main.swift

```
#!/usr/bin/swift

import Foundation


var TOTAL_NODES = 0
var WINNNING_MOVES = Dictionary<Int, [[Int]]>()
var PLAYER = 0

//
//  Connect4
//
//  Created by Tanner Juby on 3/1/17.
//  Copyright © 2017 Juby. All rights reserved.
//

import Foundation

class GameState {

    // MARK: - Class Variables

    var key: Int!

    var player : Int
    var lastMove : Int?
    var bestMove : Int?

    var state : [[Int]] = []
    var width : Int!
    var height : Int!

    var value : Int = 0

    var childrenStates : [GameState] = []
    var childrenValues : [Int] = []

    weak var parentState : GameState?


    // MARK: - Class initializers

    init(key: Int, player: Int, state: [[Int]], width: Int, height: Int) {

        self.key = key

        self.player = player

        self.state = state
        self.width = width
        self.height = height
    }


    // MARK: - Class Functions

    /**
     Block Move

     Blocks a move if needed
     */
    func findWinOrBlock() -> Int {
```

```
        generateChildren()

        for i in 0 ..< WINNNING_MOVES.count {

            let move = WINNNING_MOVES[i]

            let index1 = move?[0]
            let index2 = move?[1]
            let index3 = move?[2]
            let index4 = move?[3]

            for child in childrenStates {

                // Determine Win Move
                if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index3?[0])!]
[(index3?[1])!] == 1 {
                    if index4?[1] != height-1 && child.parentState?.state[(index4?[0])!][((index4?[1])!)+1] != 0 && child.parentState?.state[(index4?
[0])!][(index4?[1])!] != 1 {
                        return index4![0]
                    }

                } else if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index4?[0])!]
[(index4?[1])!] == 1 {
                    if index3?[1] != height-1 && child.parentState?.state[(index3?[0])!][((index3?[1])!)+1] != 0 && child.parentState?.state[(index3?
[0])!][(index3?[1])!] != 1 {
                        return index3![0]
                    }

                } else if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index3?[0])!][(index3?[1])!] == 1 && child.state[(index4?[0])!]
[(index4?[1])!] == 1 {
                    if index2?[1] != height-1 && child.parentState?.state[(index2?[0])!][((index2?[1])!)+1] != 0 && child.parentState?.state[(index2?
[0])!][(index2?[1])!] != 1 {
                        return index2![0]
                    }

                } else if child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index3?[0])!][(index3?[1])!] == 1 && child.state[(index4?[0])!]
[(index4?[1])!] == 1 {
                    if index1?[1] != height-1 && child.parentState?.state[(index1?[0])!][((index1?[1])!)+1] != 0 && child.parentState?.state[(index1?
[0])!][(index1?[1])!] != 1 {
                        return index1![0]
                    }
                }

                if player == 1 {
                    // Determine Block Move for Player 1
                    if child.state[(index1?[0])!][(index1?[1])!] == 2 && child.state[(index2?[0])!][(index2?[1])!] == 2 && child.state[(index3?[0])!]
[(index3?[1])!] == 2 && child.state[(index4?[0])!][(index4?[1])!] == 1 {

                        return index4![0]

                    } else if child.state[(index1?[0])!][(index1?[1])!] == 2 && child.state[(index2?[0])!][(index2?[1])!] == 2 && child.state[(index3?[0])!]
[(index3?[1])!] == 1 && child.state[(index4?[0])!][(index4?[1])!] == 2 {

                        return index3![0]

                    } else if child.state[(index1?[0])!][(index1?[1])!] == 2 && child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index3?[0])!]
[(index3?[1])!] == 2 && child.state[(index4?[0])!][(index4?[1])!] == 2 {

                        return index2![0]

                    } else if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index2?[0])!][(index2?[1])!] == 2 && child.state[(index3?[0])!]
[(index3?[1])!] == 2 && child.state[(index4?[0])!][(index4?[1])!] == 2 {

                        return index1![0]
                    }


                } else {

                    // Determine Win Move
                    if child.parentState?.state[(index1?[0])!][(index1?[1])!] == 2 && child.parentState?.state[(index2?[0])!][(index2?[1])!] == 2 &&
child.parentState?.state[(index3?[0])!][(index3?[1])!] == 2 {
                        if index4?[1] != height-1 && child.parentState?.state[(index4?[0])!][((index4?[1])!)+1] != 0 &&
child.parentState?.state[(index4?[0])!][(index4?[1])!] != 1 {
                            return index4![0]
                        }

                    } else if child.parentState?.state[(index1?[0])!][(index1?[1])!] == 2 && child.parentState?.state[(index2?[0])!][(index2?[1])!] == 2
&& child.parentState?.state[(index4?[0])!][(index4?[1])!] == 2 {
```

```
                    if index3?[1] != height-1 && child.parentState?.state[(index3?[0])!][((index3?[1])!)+1] != 0 &&
child.parentState?.state[(index3?[0])!][(index3?[1])!] != 1 {
                        return index3![0]
                    }

                } else if child.parentState?.state[(index1?[0])!][(index1?[1])!] == 2 && child.parentState?.state[(index3?[0])!][(index3?[1])!] == 2
&& child.parentState?.state[(index4?[0])!][(index4?[1])!] == 2 {
                    if index2?[1] != height-1 && child.parentState?.state[(index2?[0])!][((index2?[1])!)+1] != 0 &&
child.parentState?.state[(index2?[0])!][(index2?[1])!] != 1 {
                        return index2![0]
                    }

                } else if child.parentState?.state[(index2?[0])!][(index2?[1])!] == 2 && child.parentState?.state[(index3?[0])!][(index3?[1])!] == 2
&& child.parentState?.state[(index4?[0])!][(index4?[1])!] == 2  {
                    if index1?[1] != height-1 && child.parentState?.state[(index1?[0])!][((index1?[1])!)+1] != 0 &&
child.parentState?.state[(index1?[0])!][(index1?[1])!] != 1 {
                        return index1![0]
                    }
                }

                // Determine Block Move For Player 2
                if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index3?[0])!]
[(index3?[1])!] == 1 && child.state[(index4?[0])!][(index4?[1])!] == 2 {

                    return index4![0]

                } else if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index3?[0])!]
[(index3?[1])!] == 2 && child.state[(index4?[0])!][(index4?[1])!] == 1 {

                    return index3![0]

                } else if child.state[(index1?[0])!][(index1?[1])!] == 1 && child.state[(index2?[0])!][(index2?[1])!] == 2 && child.state[(index3?[0])!]
[(index3?[1])!] == 1 && child.state[(index4?[0])!][(index4?[1])!] == 1 {

                    return index2![0]

                } else if child.state[(index1?[0])!][(index1?[1])!] == 2 && child.state[(index2?[0])!][(index2?[1])!] == 1 && child.state[(index3?[0])!]
[(index3?[1])!] == 1 && child.state[(index4?[0])!][(index4?[1])!] == 1 {

                    return index1![0]
                }
            }
        }
    }

    return -1
}

/**
 Generate Possible Moves

 Generates possible moves for the current state
 */
func generateChildren() {

    childrenStates = []

    for i in 0 ..< width {
        if state[i][0] == 0 {

            var madeMove = false

            for j in 1 ..< height {
                if !madeMove {
                    if state[i][j] != 0 {

                        var newState = state
                        var newGameState : GameState

                        if player == 1 {
                            newState[i][j-1] = 1
                            newGameState = GameState(key: TOTAL_NODES+1, player: 2, state: newState, width: width, height: height)
                            newGameState.parentState = self
                            newGameState.lastMove = i
                        } else {
                            newState[i][j-1] = 2
                            newGameState = GameState(key: TOTAL_NODES+1, player: 1, state: newState, width: width, height: height)
                            newGameState.parentState = self
                            newGameState.lastMove = i
                        }
                        madeMove = true
```

```
                childrenStates.append(newGameState)

            } else if j == height-1 {

                var newState = state
                var newGameState : GameState

                if player == 1 {
                    newState[i][j] = 1
                    newGameState = GameState(key: TOTAL_NODES+1, player: 2, state: newState, width: width, height: height)
                    newGameState.parentState = self
                    newGameState.lastMove = i
                } else {
                    newState[i][j] = 2
                    newGameState = GameState(key: TOTAL_NODES+1, player: 1, state: newState, width: width, height: height)
                    newGameState.parentState = self
                    newGameState.lastMove = i
                }
                madeMove = true
                childrenStates.append(newGameState)

            }
        }
    }
}

/**
 Calculate Leaf Value

 Calculates the value of a node
 */
func calculateLeafValue() {

    for i in 0 ..< WINNNING_MOVES.count {

        let move = WINNNING_MOVES[i]

        let index1 = move?[0]
        let index2 = move?[1]
        let index3 = move?[2]
        let index4 = move?[3]

        if state[(index1?[0])!][(index1?[1])!] == 1 && state[(index2?[0])!][(index2?[1])!] == 1 && state[(index3?[0])!][(index3?[1])!] == 1 &&
state[(index4?[0])!][(index4?[1])!] == 1 {

            if player == .max {
                self.value = 100
            } else {
                self.value = -100
            }
        }

        if state[(index1?[0])!][(index1?[1])!] == 2 && state[(index2?[0])!][(index2?[1])!] == 2 && state[(index3?[0])!][(index3?[1])!] == 2 &&
state[(index4?[0])!][(index4?[1])!] == 2 {

            if player == .min {
                self.value = 100
            } else {
                self.value = -100
            }
        }
    }
}


/**
 Print State

 Prints out the board
*/
func printState() {
    var printItem = ""
    for i in 0 ..< height {
        for j in 0 ..< width {
            printItem += "\(state[j][i]) "
        }
        printItem += "\n"
    }
```

```
        print(printItem)
    }
}

/**
 Determine current move

 Determines if the current player was the first to go or the second
 */
func currentMove(state: [[Int]], height: Int, width: Int) -> Int {
    var totalMoves = 0

    for i in 0 ..< height {
        for j in 0 ..< width {
            if state[j][i] != 0 {
                totalMoves += 1
            }
        }
    }

    if totalMoves % 2 == 0 {
        return 1
    } else {
        return 2
    }
}

/**
 Calculate Winning Moves

 Determines all of the possible wins
 */
func calculateWinningMoves(width: Int, height: Int) {

    var moveIndex = 0


    // Calculate Horizontal Wins
    for i in 0 ..< height {
        for j in 0 ..< width-3 {
            let slots = [ [j, i], [j+1, i], [j+2, i], [j+3, i] ]
            WINNNING_MOVES[moveIndex] = slots
            moveIndex += 1
        }
    }

    // Calculate Verticle Wins
    for i in 0 ..< height-3 {
        for j in 0 ..< width {
            let slots = [ [j, i], [j, i+1], [j, i+2], [j, i+3] ]
            WINNNING_MOVES[moveIndex] = slots
            moveIndex += 1
        }
    }

    // Calculate Right Down Wins
    for i in 0 ..< height-3 {
        for j in 0 ..< width-3 {
            let slots = [ [j, i], [j+1, i+1], [j+2, i+2], [j+3, i+3] ]
            WINNNING_MOVES[moveIndex] = slots
            moveIndex += 1
        }
    }

    // Calculate Left Down Wins
    for i in 3 ..< height {
        for j in 0 ..< width-3  {
            let slots = [ [j, i], [j+1, i-1], [j+2, i-2], [j+3, i-3] ]
            WINNNING_MOVES[moveIndex] = slots
            moveIndex += 1
        }
    }
}


/**
 Build Tree

 Builds a tree based off of the root's children. Once a node's childen are all set, it sets that node's value depending on which player is making the move.
 This is the implementation of the min/max algorithm because it sets up the node's value as it the tree is being wrapped back up.
```

```
 */
func buildTree(root: GameState, currentDepth: Int, maxDepth: Int, move: Int) -> GameState {

    if currentDepth == maxDepth {
        // Reached max depth
        root.calculateLeafValue()

        return root
    } else {
        root.generateChildren()

        var moves : [GameState] = []

        for child in root.childrenStates {
            if move == 1 {
                let newMove = buildTree(root: child, currentDepth: currentDepth+1, maxDepth: maxDepth, move: 2)
                moves.append(newMove)
                root.childrenValues.append(newMove.value)
            } else {
                let newMove = buildTree(root: child, currentDepth: currentDepth+1, maxDepth: maxDepth, move: 1)
                moves.append(newMove)
                root.childrenValues.append(newMove.value)
            }

        }

        root.childrenStates = moves

        if move == 1 {
            let maxIndex = root.childrenValues.index(of: .max)
            if maxIndex == nil {
                root.value = root.childrenValues[0]
                root.bestMove = root.childrenStates[0].lastMove
            } else {
                root.value = root.childrenValues.max()!
                root.bestMove = root.childrenStates[maxIndex!].lastMove
            }
        } else {
            let minIndex = root.childrenValues.index(of: .min)
            if minIndex == nil {
                root.value = root.childrenValues[0]
                root.bestMove = root.childrenStates[0].lastMove
            } else {
                root.value = root.childrenValues.min()!
                root.bestMove = root.childrenStates[minIndex!].lastMove
            }
        }

        return root
    }
}


/**
 Is Move Possible

 Double checks that the move about to be made is legal
 */
func isMovePossible(move: Int, state: [[Int]]) -> Bool {

    let column = state[move]

    if column[0] == 0 {
        return true
    }

    return false
}

/**
 Find first open column

 Finds the first open column in the state
 */
func findFirstOpenColumn(state: [[Int]]) -> Int {

    for i in 0 ..< state.count {
        let column = state[i]
        if column[0] == 0 {
            return i
        }
```

```
    }

    return -1
}


/**
 Get Data

 Import the Data from the driver program
 */
func getData() -> Dictionary<String, Any> {

    for argument in CommandLine.arguments {

        if let data = argument.data(using: .utf8) {
            do {
                return try JSONSerialization.jsonObject(with: data, options: []) as! [String: Any]
            } catch {
                print(error.localizedDescription)
            }
        }
    }
    return [:]
}

/**
 Send Data

 Write the data to stdout
 */
func sendData(_ move: Dictionary<String, Any>) {
    let stdout = FileHandle.standardOutput

    do {
        let jsonData = try JSONSerialization.data(withJSONObject: move, options: [])
        stdout.write(jsonData)
    } catch {
        let string = "ERROR: Could not process dictionary"
        let data = string.data(using: String.Encoding(rawValue: String.Encoding.utf8.rawValue))
        stdout.write(data!)
    }
}

/**
 Write Error

 Write to the trace file
 */
func writeError(_ err: Dictionary<String, Any>) {
    let stderr = FileHandle.standardError

    do {
        let jsonData = try JSONSerialization.data(withJSONObject: err, options: [])
        stderr.write(jsonData)
    } catch {
        let string = "ERROR: Could not process dictionary"
        let data = string.data(using: String.Encoding(rawValue: String.Encoding.utf8.rawValue))
        stderr.write(data!)
    }
}


/**

 The Main Procedure

 */


/**
 Load the data from the standard input and set the main procedure's base values
 */
let inData = getData()
let state : [[Int]] = inData["grid"] as! [[Int]]
let height = inData["height"] as! Int
let width = inData["width"] as! Int
PLAYER = inData["player"] as! Int

let onMove = currentMove(state: state, height: height, width: width)
```

```
/**
 Set up the program for implementtion of min-max
 */
calculateWinningMoves(width: width, height: height)
var nodeState = GameState(key: TOTAL_NODES, player: PLAYER, state: state, width: width, height: height)

/**
 Set up the children state
 */
nodeState.generateChildren()

/**
 Check to see if there is a block or win move
 */
var move = nodeState.findWinOrBlock()

/**
 If there is no block or win move, implement the min-max
 */
if move == -1 {
    var tree = buildTree(root: nodeState, currentDepth: 0, maxDepth: 3, move: onMove)

    move = tree.bestMove!
}

/**
 Double check if the move generated is possible. If not, set the move as the first open column
 */
if !isMovePossible(move: move, state: state) {
    move = findFirstOpenColumn(state: state)

    if move == -1 {
        move = width-1
    }
}

/**
 Set up the
 */
var moveData = Dictionary<String, Any>()
moveData["move"] = move

/**
 Write the information to the err file
*/
writeError(inData)
writeError(moveData)

/**
 Send the move back to the driver program
 */
sendData(moveData)
```

Expected Results

It is expected that my program will win every time, or at least close to every time against the naive program. Against others, it should be able to at least block a win if there is only one winning move in a state, and it should be able to win if they have a winning move. My program is expected to win every time it goes first. Currently, the program is set up to only go 5 deep in the min-max tree, so until the game gets to a point where a player is within a move or two of winning, my program should always be picking the first column available as a move. Once the game gets to a point where one player is within a move or two of winning, then my AI should actually make strategic moves.

Results

Round	Player 1	Player 2	Winner	Total Moves
1	Naive	Tanner Juby AI	Tanner Juby AI	16
2	Naive	Tanner Juby AI	Tanner Juby AI	7
3	Naive	Tanner Juby AI	Tanner Juby AI	16
4	Naive	Tanner Juby AI	Tanner Juby AI	12
5	Naive	Tanner Juby AI	Tanner Juby AI	7
6	Naive	Tanner Juby AI	Tanner Juby AI	7
7	Naive	Tanner Juby AI	Tanner Juby AI	7
8	Naive	Tanner Juby AI	Tanner Juby AI	8
9	Naive	Tanner Juby AI	Naive	13
10	Naive	Tanner Juby AI	Tanner Juby AI	8
11	Naive	Tanner Juby AI	Tanner Juby AI	16
12	Naive	Tanner Juby AI	Tanner Juby AI	21
13	Naive	Tanner Juby AI	Tanner Juby AI	8
14	Naive	Tanner Juby AI	Tanner Juby AI	8
15	Naive	Tanner Juby AI	Tanner Juby AI	9
1	Tanner Juby AI	Tanner Juby AI	*Player 1*	37
2	Tanner Juby AI	Tanner Juby AI	*Player 1*	37
3	Tanner Juby AI	Tanner Juby AI	*Player 2*	37
4	Tanner Juby AI	Tanner Juby AI	*Player 1*	37
5	Tanner Juby AI	Tanner Juby AI	*Player 2*	37
1	Jonathan Pham AI	Tanner Juby AI	Tanner Juby AI	27
2	Jonathan Pham AI	Tanner Juby AI	Jonathan Pham AI	17
3	Jonathan Pham AI	Tanner Juby AI	Jonathan Pham AI	17
4	Jonathan Pham AI	Tanner Juby AI	Tanner Juby AI	27
5	Jonathan Pham AI	Tanner Juby AI	Tanner Juby AI	27

Conclusions

My AI program does what I would expect it to. It beats the naive program most of the time pretty quickly. It lost once, and after that one loss, I found an error in my find W*inOrBlock* code that caused my program to not block the win. The next time around, my program made the block that the naive program originally won on. My program makes the correct blocks and win moves when available. If given the chance, my min-max implementation seems to work properly, but its a little hard to tell because I usually beat the naive program in the first 10 moves, and the state of the game hardly ever gets to a point where the naive program is within two moves of winning, so most of the time the tree's node values are 0. This made it hard to evaluate the min-max implementation.

My AI against itself always yielded expected results. Whichever player went first, won the game in 37 total moves. The game was always a long game and the board was almost filled. The game the exact same every time it played against itself, it just depended on which player was chosen to go first.

Against the other student's (Jonathan Pham) program I ran mine against, we were very equal. When my AI was chosen to go first, my AI won the game every time. When Jon's AI was chosen to go first, his AI won the game every time. Between our games, they resulted in the exact same outcomes depending on who went first. When my AI went first, it took 27 moves to win and the final states of each of those games were the same. When Jon's AI went first, it took 17 moves to win and the final states of each of those games were the same as well.

If I were to refine this AI, I would make the scores given to leaf nodes a bit more detailed, such as giving a ranked value to the node: instead of 100 for a win and -100 for a loss, I would include +- 25 for each pair of tiles that are next to each other, +- 50 for each trio a player has on the board, etc. This would give a more in depth evaluation of the current game state and the min-max algorithm and the utilization of it would be more effective.