CSCI 4202 – Introduction to Artificial Intelligence
Fall 2021 – Dr. Williams
Program 2 – Game Playing
Due: November 19, 2021

Connect Four

Connect Four is a two-player connection game in which the players first choose a color and then take turns dropping colored discs from the top into a seven-column, six-row vertically suspended grid. The pieces fall straight down, occupying the next available space within the column. The objective of the game is to connect four of one's own discs of the same color next to each other vertically, horizontally, or diagonally before your opponent. Connect Four is a strongly solved game. The first player can always win by playing the right moves.[1]

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.[2]

A description of the JSON format as well as links to parsers for various languages is available on the JSON web site at http://www.json.org.

Specifics

You will write a program to play Connect Four. Your program will read, from standard input, a description of the current game state in JSON format and write, to standard output, its move in JSON format. Your will be provided a driver program and a naïve player program (that makes random, valid moves) that will be used to test your player program. You can also test your player program against itself and other student's player programs.

An acceptable player program must:

1. Read the game state in JSON format from standard input and write the move in JSON format to standard output. Any desired trace or debugging information – as well as error messages – may be printed to standard error, which will be written to a text file.
2. Play a valid game of Connect Four. In particular, it must make valid moves in response to the current game state.
3. Consistently beat the naïve Connect Four player program provided. [The naïve player program may be the first player and – by chance – make optimal moves and win a game. But, this would be exceedingly rare.]

[1] Wikipedia contributors. Connect Four [Internet]. Wikipedia, The Free Encyclopedia; 2016 Sep 3, 00:32 UTC [cited 2016 Sep 3]. Available from: https://en.wikipedia.org/w/index.php?title=Connect_Four&oldid=737467965.
[2] Introducing JSON. Available from: http:www.json.org.

In addition, every student must play their player program against at least one other student player model. Students are encouraged to form tournaments pitting several student player models against each other and ranking them.

You may program your player model in any programming language you like. The JSON web site has links to JSON parsers for many languages – there are often several available for any particular language. Your player program must be callable from the command line.

Driver Program

The driver program is written in Racket and runs under Windows, Apple OS X, and (various) Linux operating systems. You will need to download and install Racket for your system. It is available from the Racket web site http://racket-lang.org/. [I generally use the 64-bit version. But, the 32-bit version is fine for this project.]

The driver program and naïve player program are available from the class Canvas page:

`connect-four-driver.rkt`	Driver program that plays to player programs
`connect-four-naive.rkt`	Naïve player program – makes random, valid moves

Configuring the driver program

The driver program – `connect-four-driver.rkt` – can be directly edited and run from the DrRacket IDE. You don't need to compile it.

There are several global variables at the top of the `connect-four-driver.rkt` file.

```
;;; Define grid dimensions.
(define HEIGHT 6)
(define WIDTH 7)

;;; These will eventually be command line arguments.
(define exe-1 "connect-four-naive.exe")
(define args-1 "")
(define exe-2 "C:\\Program Files\\Racket\\Racket.exe")
(define args-2 "connect-four-naive.rkt")
```

The first two, `HEIGHT` and `WIDTH`, define the size of the grid. The standard connect four grid, 6 × 7, is the default. The smallest size that makes sense is 4 × 4. Your program must be able to accommodate any grid size up to 16 × 16. [The maximum size here is totally arbitrary and you may choose to handle even larger ones.] For a tournament, you may consider a 6 × 8 grid. This may reduce the benefit to the first player.

The next group of four (4) define the two (2) player program command lines. Each is defined by the name of the executable program and any command line arguments. There are two cases: (1) If a player program is compiled as an executable – as is the first case here – you just give the path to that executable. In this case, there are no arguments. (2) If a player model is interpreted – such as Python or, as shown here, Racket – or runs in a virtual machine environment – such as Java, you give the path to the executable for the interpreter or VM and the arguments specific to your player model – typically the source code file (for an interpreter) or object code file (for a VM).

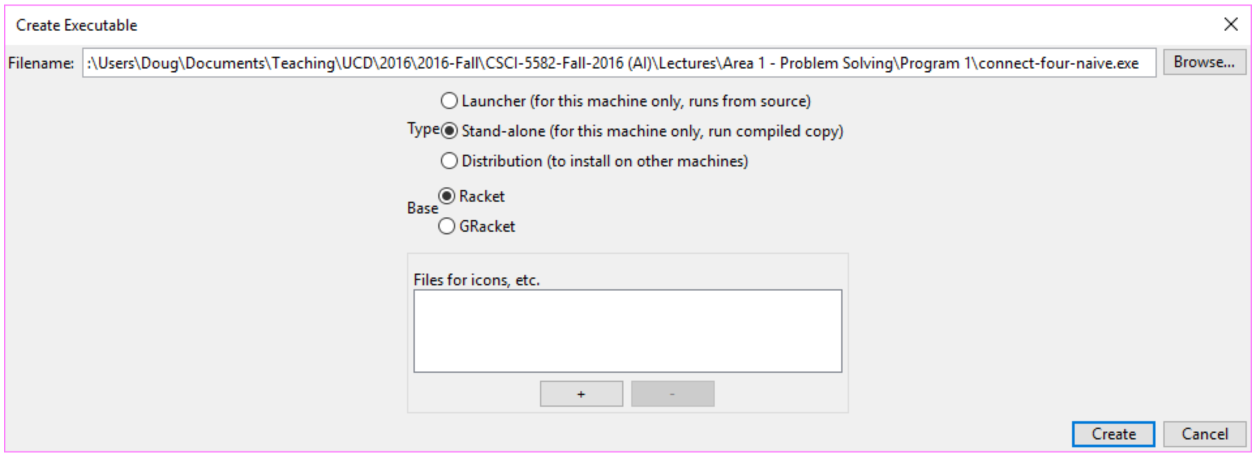
Note that this example is for a Windows environment. For an Apple OS X or Linux environment, there will not be a .exe extension for executable files.

It is important that the executable name be either a complete relative path from the current working directory or a complete absolute path. Particularly, the command isn't executed inside a command shell and it won't search the PATH environment variable for the executable name. [Your mileage may vary depending on operating system, etc.]

Compiling the naïve player program

The naïve player program can be either compiled – as in the first case above – or interpreted – as in the second case above. Generally, I would recommend compiling a player model. This is the only possibility for compiled languages like C++. It is optional for some interpreted languages, like Racket, that provide both compiled and interpreted options. Java is (generally) a combination of the two. You compile your source code to byte code using the javac command and execute the resulting byte code using the JVM using the java command.

The easiest way to compile the naïve player program is to open the Rack source code file `connect-four-naive.rkt` in DrRacket. Then select "Racket > Create Execuable …" from the menu bar. This brings up the following dialog box.


Create Executable

Filename: :\Users\Doug\Documents\Teaching\UCD\2016\2016-Fall\CSCI-5582-Fall-2016 (AI)\Lectures\Area 1 - Problem Solving\Program 1\connect-four-naive.exe

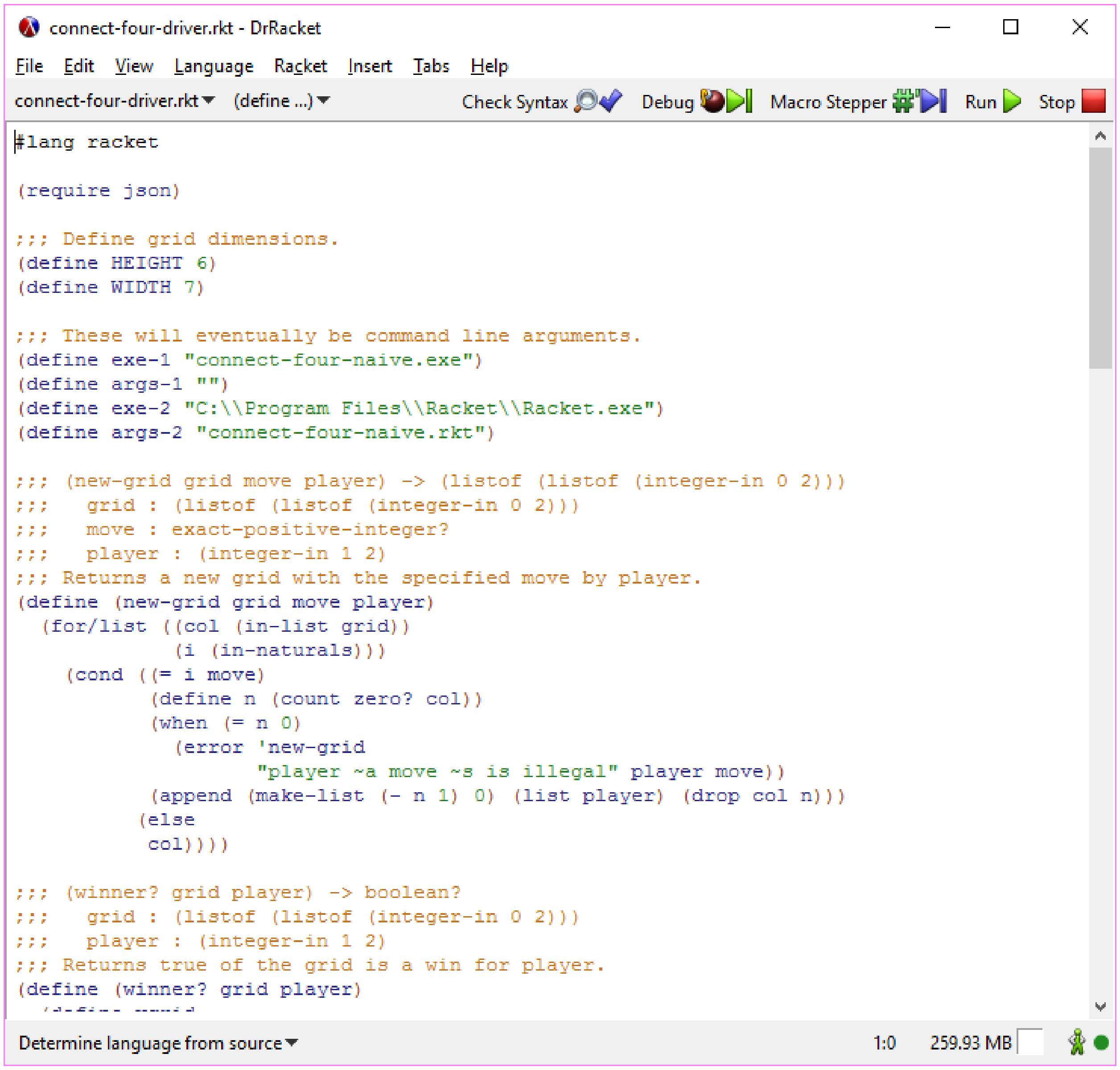
Type: ○ Launcher (for this machine only, runs from source) [✓ ● Stand-alone (for this machine only, run compiled copy)] ○ Distribution (to install on other machines)

Base: [✓ ● Racket] ○ GRacket

Make sure the "Stand-alone (for this machine only, run compiled copy)" and "Racket" options are selected. Then click the "Create" button to create the executable. It will take several seconds to complete. You should then have a `connect-four-naive.exe` file in the same directory as the source code file. [Note that for Apple OS X and Linux, there will not be a .exe extension on the executable.]

Running the driver program

At this point, you should be able to load the driver program and play to naïve player programs against each other. First, open the `connect-four-driver.rkt` file in DrRacket.


connect-four-driver.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
connect-four-driver.rkt (define ...)
Check Syntax
Debug
Macro Stepper
Run
Stop
Determine language from source
1:0
259.93 MB

```
#lang racket

(require json)

;;; Define grid dimensions.
(define HEIGHT 6)
(define WIDTH 7)

;;; These will eventually be command line arguments.
(define exe-1 "connect-four-naive.exe")
(define args-1 "")
(define exe-2 "C:\\Program Files\\Racket\\Racket.exe")
(define args-2 "connect-four-naive.rkt")

;;; (new-grid grid move player) -> (listof (listof (integer-in 0 2)))
;;;   grid : (listof (listof (integer-in 0 2)))
;;;   move : exact-positive-integer?
;;;   player : (integer-in 1 2)
;;; Returns a new grid with the specified move by player.
(define (new-grid grid move player)
  (for/list ((col (in-list grid))
             (i (in-naturals)))
    (cond ((= i move)
           (define n (count zero? col))
           (when (= n 0)
             (error 'new-grid
                    "player ~a move ~s is illegal" player move))
           (append (make-list (- n 1) 0) (list player) (drop col n)))
          (else
           col))))

;;; (winner? grid player) -> boolean?
;;;   grid : (listof (listof (integer-in 0 2)))
;;;   player : (integer-in 1 2)
;;; Returns true of the grid is a win for player.
(define (winner? grid player)
```

Then, click on the Run button. This will start the two player programs and alternate between them sending them the current game state and receiving their move. This continues until either one player wins or there are no move moves, in which case the game is a draw. Note that the driver randomly selects one of the players to move first.

Here is one possible game output:

Welcome to DrRacket, version 6.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.

```
--- Move 1 ---
```

```
    1
--- Move 2 ---




    2
    1
--- Move 3 ---




    2
    1   1
--- Move 4 ---




    2
  2 1   1
--- Move 5 ---



    1
    2
  2 1   1
--- Move 6 ---



    1
    2
  2 1   1 2
--- Move 7 ---



    1
    2
  2 1   1 2 1
--- Move 8 ---



    1
    2     2
  2 1   1 2 1
--- Move 9 ---
```

```
    1
    2     2
  2 1 1 1 2 1
--- Move 10 ---


    2
    1
    2     2
  2 1 1 1 2 1
--- Move 11 ---


    2
    1
    2     2 1
  2 1 1 1 2 1
--- Move 12 ---

    2
    2
    1
    2     2 1
  2 1 1 1 2 1
--- Move 13 ---
    1
    2
    2
    1
    2     2 1
  2 1 1 1 2 1
--- Move 14 ---
    1
    2
    2
    1
  2 2     2 1
  2 1 1 1 2 1
--- Move 15 ---
    1
    2
    2
    1     1
  2 2     2 1
  2 1 1 1 2 1
--- Move 16 ---
    1
    2
    2
  2 1     1
  2 2     2 1
  2 1 1 1 2 1
```

```
--- Move 17 ---
    1
    2
    2
  2 1     1
  2 2   1 2 1
  2 1 1 1 2 1
--- Move 18 ---
    1
    2
    2
  2 1     1
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 19 ---
    1
    2
    2     1
  2 1     1
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 20 ---
    1
    2
    2     1
  2 1     1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 21 ---
    1
    2
    2     1
  2 1 1   1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 22 ---
    1
    2     2
    2     1
  2 1 1   1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 23 ---
    1
    2     2
    2 1   1
  2 1 1   1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 24 ---
    1
    2     2
    2 1   1
  2 1 1 2 1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 25 ---
    1     1
    2     2
    2 1   1
  2 1 1 2 1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 26 ---
    1     1
    2     2
    2 1 2 1
  2 1 1 2 1 2
  2 2 2 1 2 1
  2 1 1 1 2 1
--- Move 27 ---
    1     1
    2     2
    2 1 2 1
  2 1 1 2 1 2
  2 2 2 1 2 1
1 2 1 1 1 2 1
--- Move 28 ---
    1     1
    2     2
  2 2 1 2 1
  2 1 1 2 1 2
  2 2 2 1 2 1
1 2 1 1 1 2 1
Player 2 wins.
>
```

In this case, player 2 wins after 28 moves.

Since the play is completely random, each game is different.

The naïve player model

Here is the Racket source code for the naïve player program.

```
#lang racket

(require json)

;;; (valid-moves precept) -> (listof exact-nonnegative-integer?)
;;;   percept : jsexpr?
;;; Returns a list of the valid moves - that is columns that aren't full.
(define (valid-moves precept)
  (match precept
    ((hash-table
       ('player player) ('height height) ('width width) ('grid grid))
      (for/fold ((moves '()))
                ((col (in-list grid))
                 (i (in-naturals)))
        (cond ((= 0 (first col))
               (append moves (list i)))
              (else
               moves))))))

;;; (main) -> void?
(define (main)
  (displayln "Connect Four" (current-error-port))
  ;; Process the precepts.
  (for ((json (in-lines)))
    (displayln json (current-error-port))
    (with-handlers ((exn:fail?
                     (lambda (e) (displayln "null"))))
      (define precept (string->jsexpr json))
      ;; Find the valid moves.
      (define moves (valid-moves precept))
      ;; Choose one at random.
      (define move (list-ref moves (random (length moves))))
      ;; Return the action.
      (define action
        (hasheq 'move move))
      (define action-json (jsexpr->string action))
      (displayln action-json (current-error-port))
      (displayln action-json)
      (flush-output))))

;;; Start the program.
(main)
```

This program repeatedly reads the game state in JSON format from standard input, parses the game state into an internal form – a hash table for Racket, determines the valid moves from the game state, selects one of the valid moves at random, creates a hash table with the selected move, converts the hash table into JSON format, and writes the JSON to both standard error (for debugging) and standard output.

IMPORTANT: It is important to remember to flush the output buffer when writing the JSON to standard output. Otherwise, for efficiency, the output will (likely) be placed in a buffer and won't actually be send until the buffer is full. This will cause a deadlock condition between the driver program and the player model. Flushing standard output – however that is done in your language – is important to immediately send the data to the driver.

JSON formatted game state and move

Game state

The game state consists of the which player is to move – which will always be 1 for Player 1 and 2 for Player 2, the height and width of the grid – which will be the same for every game state for a given game, and the grid – which is game grid in column-major order. Note that the fields may be provided in any order. [Generally, the order will depend on the specific JSON implementation.

For example, the game state after the 18th move in the above game (i.e., the one passed to Player 1 for the 19th move:

```
--- Move 18 ---
    1
    2
    2
  2 1     1
  2 2 2 1 2 1
  2 1 1 1 2 1
```

Was represented as follows:

```
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,0,0,2,1],[0,0,0,0,1,1
],[0,0,0,1,2,2],[0,0,0,0,1,1]],"height":6,"player":1,"width":7}
```

Move

The move consists of a single field: move – which is the column in which to move.

For example, the 19th move in the above game – to drop its disk in column 5 – was represented as follows:

```
{"move":5}
```

Standard error trace

The driver program creates a text file for each player program and directly and output by one of the player programs to the correct file. The files are `connect-four-stderr-1.txt` for Player 1 and `connect-four-stderr-2.txt` for Player 2.

Here is the output from the naïve player program for Player 1 from the game above.

```
Connect Four
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,0,0
],[0,0,0,0,0,0],[0,0,0,0,0,0]],"height":6,"player":1,"width":7}
{"move":2}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,0],[0,0,0,0,2,1],[0,0,0,0,0,0],[0,0,0,0,0,0
],[0,0,0,0,0,0],[0,0,0,0,0,0]],"height":6,"player":1,"width":7}
{"move":4}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,2],[0,0,0,0,2,1],[0,0,0,0,0,0],[0,0,0,0,0,1
],[0,0,0,0,0,0],[0,0,0,0,0,0]],"height":6,"player":1,"width":7}
{"move":2}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,2],[0,0,0,1,2,1],[0,0,0,0,0,0],[0,0,0,0,0,1
],[0,0,0,0,0,2],[0,0,0,0,0,0]],"height":6,"player":1,"width":7}
{"move":6}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,2],[0,0,0,1,2,1],[0,0,0,0,0,0],[0,0,0,0,0,1
],[0,0,0,0,2,2],[0,0,0,0,0,1]],"height":6,"player":1,"width":7}
{"move":3}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,2],[0,0,2,1,2,1],[0,0,0,0,0,1],[0,0,0,0,0,1
],[0,0,0,0,2,2],[0,0,0,0,0,1]],"height":6,"player":1,"width":7}
{"move":6}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,0,2],[0,2,2,1,2,1],[0,0,0,0,0,1],[0,0,0,0,0,1
],[0,0,0,0,2,2],[0,0,0,0,1,1]],"height":6,"player":1,"width":7}
```

```
{"move":2}
{"grid":[[0,0,0,0,0,0],[0,0,0,0,2,2],[1,2,2,1,2,1],[0,0,0,0,0,1],[0,0,0,0,0,1
],[0,0,0,0,2,2],[0,0,0,0,1,1]],"height":6,"player":1,"width":7}
{"move":5}
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,0,0,0,1],[0,0,0,0,0,1
],[0,0,0,1,2,2],[0,0,0,0,1,1]],"height":6,"player":1,"width":7}
{"move":4}
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,0,0,2,1],[0,0,0,0,1,1
],[0,0,0,1,2,2],[0,0,0,0,1,1]],"height":6,"player":1,"width":7}
{"move":5}
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,0,0,2,1],[0,0,0,0,1,1
],[0,0,1,1,2,2],[0,0,0,2,1,1]],"height":6,"player":1,"width":7}
{"move":3}
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,0,1,2,1],[0,0,0,0,1,1
],[0,2,1,1,2,2],[0,0,0,2,1,1]],"height":6,"player":1,"width":7}
{"move":3}
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,1,1,2,1],[0,0,0,2,1,1
],[0,2,1,1,2,2],[0,0,0,2,1,1]],"height":6,"player":1,"width":7}
{"move":5}
{"grid":[[0,0,0,0,0,0],[0,0,0,2,2,2],[1,2,2,1,2,1],[0,0,1,1,2,1],[0,0,2,2,1,1
],[1,2,1,1,2,2],[0,0,0,2,1,1]],"height":6,"player":1,"width":7}
{"move":0}
```

Note that the player field is always 1, the height and width fields are always 6 and 7, and the grid field represents the current game grid by column.

Note that during debugging you can write whatever output you need to standard error and it will show up here.

It is good to always include some initial output – like the “Connect Four” in the naïve player program. If the text file doesn’t even contain that initial output, it is likely that the command line arguments – like `exe-1` and `args-1` – are incorrect and your program isn’t even loading.

Report

Each student is to turn in a report – in pdf format – detailing their player program and results. Each report should include:

- Problem statement – the problem as stated in this document
- Player program design – the algorithms (pseudocode), data structures, etc.
- Player program implementation – the source code (with comments)
- Expected results
- Results – your player program versus the naïve player program; versus itself (does the first player always win, do they always tie, or some other outcome); against other student’s player programs
- Conclusions