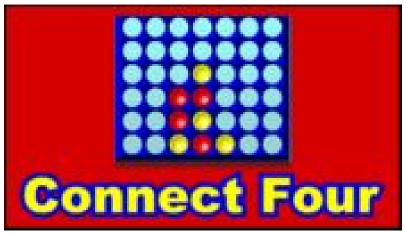


CS 152 Project Report

CONNECT FOUR

'Only Rajnikanth can win a game of Connect 4 with 3 moves'

TEAM MEMBERS
Devansh Shah (150070004), Archit Gupta (150070001), Madhav Goel(150110017)

THE PROBLEM STATEMENT

Our project is a scheme recreation of the famous two player strategy board game Connect Four. In the two-player version of the game each player first chooses a color and then they 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 challenge is to design a user v/s computer AI version of the game, using heuristics to ensure efficient moves of the AI. Also designing a UI for the game.

THE PROGRAM DESIGN

❖ **External Structure:**

There are 4 files

1. *position.rkt* - contains the class which handles the board on which the game is played, and heuristic for determining the move played by AI.
2. *connect4.rkt* - implements the solver, making use of minimax algorithm and alpha-beta pruning.
3. *main.rkt* - handles the actual execution of the game, handling the dynamic display of the board after each move by player and AI.
4. *drawing-routine.rkt* - handles the GUI component of the project wherein it builds and displays the dialog showing the current state of the game.

❖ **Internal Structure:**

<u>**Classes**</u> **:** We have implemented 3 major classes:

1. **Position class** : Implements the state of the board as a 2d-vector.Throughout the game, the moves played by the user/computer are reflected on the board via the functions defined here. The major functions user are:
    1. **play** function modifies the 2d-vector given the player number and column to play.
    2. **heuristic** function calculates the score given state of the board.The score is indicative of which player is currently in a better position. The idea used here is that the number of potential 4 consecutive with voids is calculated for each player and their difference is returned.
    3. **isWinningMove** function determines whether the game has ended by evaluating whether there is a horizontal, vertical or diagonal four consecutive discs of a colour.

2. **Solver class:** The heart of our AI Game. Given a state of the board as input, the best move for the computer is evaluated. Main function used here:

    1. Minmax function that implements the **minmax** algorithm using **alpha-beta pruning** on the game tree. The algo searches until a specified depth. If a terminating state is reached before the depth is attained, it returns that configuration. Else at the end of the specified depth, a heuristic function of the Position class determines a score for that leaf state.

3. **Frame and Canvas Class** : Implements the GUI window using the default frame function defined in Drracket library. For making the canvas we inherited a my-canvas from the standard canvas function of racket to override the on-mouse-click function.

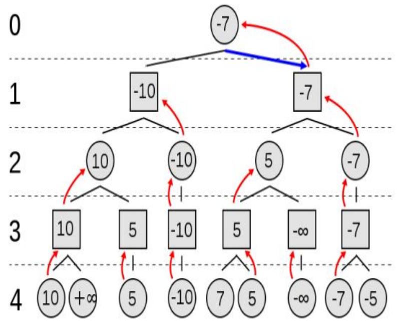
1. **On-event get-left-down** is triggered when a mouse click occurs. Defined for overriding, this function implements a user play and a red coin is dropped in the corresponding column. It also triggers the AI function.
2. **AI-play** function represents the computer's move. It sends the current configuration to the heuristic solver, obtains the best playable position, and then drops a yellow coin there, then prompting the user to take his turn.
3. **Draw-config** function is called by main.rkt on every iteration.It updates the displayed board according to the configuration of red and yellow coins.

Abstractions and higher order functions:

1. **Count target configurations abstraction:** The heuristic function to calculate score uses abstraction to count the number of possible four in a row patterns in our board. This pattern can be vertical/horizontal or along either diagonals. So we wrote a common higher-order function to count such patterns independent of the direction. Also, to calculate whether playing a move results in a winning configuration, we used the same abstraction defined above
2. **Classes and Functions Abstraction:** The position class is an abstraction for all manipulations of the board on which the game is played. Functions like canPlay, isWinningMove represent a higher level implementation of the functionalities, hiding the unnecessary details from user.
3. The solver class is the abstraction which given an object of position class and the player number returns the column in which to play the next move to ensure the most efficient move.
4. **Control Flow Abstraction**: Used define-syntax for executing 'for' and 'while' loop which represents standard patterns in the code execution.

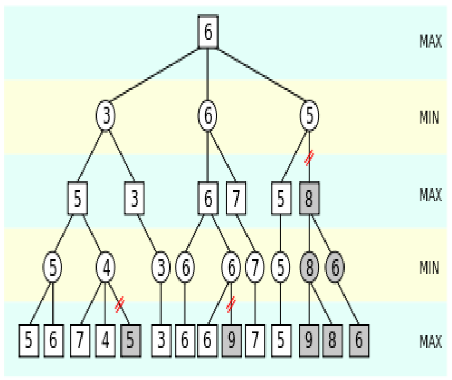
Our AI Algorithm

1) **Minmax**



There are essentially two players playing the game - The computer is the maximizing player and the user a minimizing player. Given a state of the board there are multiple possibilities to play the next move. The objective is to choose the best move that the maximizing player should play. For this all possible next states are considered and that move is selected which gives the maximum score for the maximizing player assuming the worst case where the opposite player also plays his best game.The minimizing player picks up that next move which results in a minimum score for the maximizing player .The game alternates between the two players and evaluates until a specified depth.

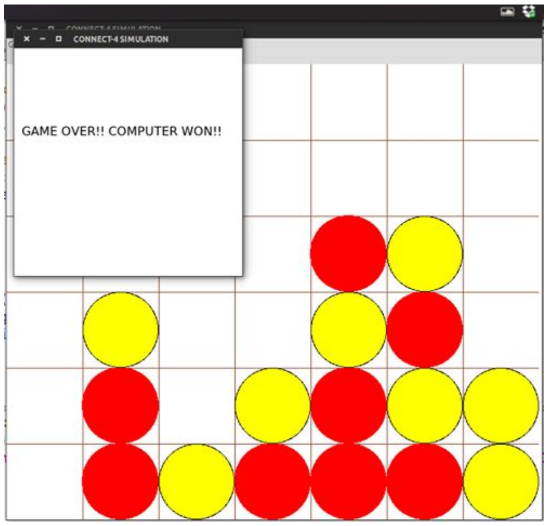
2) **Alpha-beta Pruning**



A naive recursion will blow up the search space in a few depths itself so some states are needed to be pruned or eliminated. The algorithm maintains two values, alpha and beta, which represent the maximum score that the maximizing player is assured of and the minimum score that the minimizing player is assured of respectively. Initially alpha is a large negative value and beta is a large positive value player. It can happen that when choosing a certain branch of a certain node the minimum score that the minimizing player is assured of becomes less than the maximum score that the maximizing player is assured of (beta <= alpha). If this is the case, the parent node should not choose this node, because it will make the score for the parent node worse. Therefore, the other branches of the node do not have to be explored.

SAMPLE INPUT AND OUTPUT

1. To start the game, put all 4 files in the same folder and run the main.rkt file.
2. A 6 x 7 board in displayed, and the game is afoot!
3. By default the user is the first player(red coin) and the AI is the second player(yellow coin).
4. The user can click in any column on the board and a yellow coin will be dropped at the topmost available space in the column.
5. After that, the AI determines the best possible move to ensure its victory. (Who doesn't like to win?)
6. The game continues with user and AI alternating moves, until one of them wins the game or the game is draw.
7. When the game ends, a dialog box opens signalling ''GAME OVER and declaring the winner, leaving behind a sulking loser.



GAME OVER!! COMPUTER WON!!

LIMITATIONS

1. ***Speed vs AI accuracy tradeoff -*** Designing heuristics for the game involves making a tradeoff between speed of the game and the accuracy of the AI algorithm. We tried out several different heuristics for the game, and discovered that an algorithm which gives more accurate answers usually tends to take longer time. We have attached with the project submission a file showing the incremental heuristics we tried out.

INTERESTING FEATURES

1. ***Debounced Mouse Click*** Whenever a user unintentionally double-clicks the mouse or the laptop mouse click is really sensitive, the player move was duplicated, not desired right? So to take care of that we used the concept of states taught in class. We maintain a state variable 'Debouncer', which becomes 0 on a mouse click , but is restored only after the a few iterations of the main loop. This is enough to ensure a single click is registered.

2. ***How to Clone objects?*** In racket if we try copying a class instance to another directly using define statement, it associates the same pointer with the new copy that is created.As a result if we modify the new class instance the same changes get reflected in the original class instance. To avoid this we created a new member function called clone that copies the original class fields into the new class instance and returns the new object instance with the same fields as the original class.

CITATIONS AND REFERENCES

1. http://web.mit.edu/sp.268/www/2010/connectFourSlides.pdf
2. http://roadtolarissa.com/connect-4-ai-how-it-works/
3. http://www.cs.cornell.edu/courses/cs2110/2014sp/assignments/a4/A4ConnectFour.pdf
4. https://docs.racket-lang.org/gui/mouse-event_.html
5. http://blog.gamesolver.org/
6. The Racket Documentation _/_