PHASE 2 Report

Team Infinity

CMPS 270: Software Construction

Dr. Rida Assaf

11/18/2022

Laptop Specs:
TUF Gaming FX705GM, 16 GB RAM, 6 GB Nvidia Geforce 1060, Core i7
Lenovo Slim 7, 16 GB RAM, Core i7 1065g7

Mini-Max Algorithm General View

- The Mini-Max algorithm is a recursive or backtracking method used in game theory and decision-making. It offers the player an ideal move, presuming the opponent is also playing perfectly.
- Mini-Max algorithm uses recursion to search through the game-tree.
- This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game; one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Here, the algorithm is implemented as a 2D matrix, as requested, but logically acts in the same way as the above.

Function Implementation:

- coinToss() function: The user possesses the ability to select between a player or bot opponent, where the beginning move (player vs opponent) is determined randomly by a toss of a coin.

- evaluate(State arr[ROWS][COLS]) function: compare the scores on the board in order to determine where the bot will either block the player or place a winning move. Such algorithm works by checking the values at the entries where the score increases as the adjacent entries of the same value increase. This is checked horizontally, vertically, and diagonally.

- CheckHorizontal(State board[ROWS][COLS], State key): this method will traverse the whole rows and check for the first 3 columns - 3 elements of each array- and will check whether there is 4 in a row of the same coin- 1 or 2- in case one series of 4 found, it will return true and the game will be won.

- CheckVertical(State board[ROWS][COLS], State key): this method will traverse the matrix by columns and will visit the first 3 rows of the matrix and will check whether there is 4 consecutive coins of the same value. in case found, it will return true and hence the game is won.

- CheckDiagonals(State board[ROWS][COLS], State key): in this method, we need to traverse the whole rows of the matrix, however we need to traverse the column in 2 ways. In case of the top 3 rows, we hence need to check for the negatively sloped diagonals, and check for a possible of 4 consecutive coins of same type, so that if it occurred, the game is won hence returns true. after the first 3 rows are checked, we have checked all the possible negatively sloped diagonals in the game, and we need to check the positively sloped diagonals now. we do that on the last 3 rows in the matrix, as they include the whole positively sloped diagonals in the matrix, and in case of 4 consecutive coins of same value, it returns true(1) and the game is won.

- check(): this method will return the value of checkVertical,checkHorizontal and checkDiagonal and in case one them is true(1) it returns the value and the game is won, otherwise it returns false(0) and the game continues.

- countCol(State board[ROWS][COLS], int startRow, int startCol) function: loops over the board and returns the number of consecutive keys in a col (e.g., 2,2,2,2 => 4)

- countRow(State board[ROWS][COLS], int startRow, int startCol) function: loops over the board and returns the number of consecutive keys in a row (e.g., 2,2,2,1 => 3)

- countDiag(State board[ROWS][COLS], int startRow, int startCol) & countDiagb(State board[ROWS][COLS], int startRow, int startCol) functions: loop over the board and return the number of consecutive keys diagonally

- add_token(State board[ROWS][COLS], State key) function: adds the token/key input value into the board starting from the last row (e.g., input col = 4, place at col 4 at the lowest row with 0 at col 4)

- minmax(State arr[ROWS][COLS], int isMaximizing, int depth, int alpha, int beta) function: this is the minmax algorithm implementation. first, we take declare an int score and assign to it the evaluation score of the matrix that is returned from the evaluate method. We check this score if it is -100 or 100, and that is the base case that will break the recursion. We create an array of all the possible columns which is initially 7 since we have 7 choices, and we make them -1. In case the function was called on the maximizing player- the argument entered is 1. we initialize the value best to a very low negative number so that we ensure that we are maximizing it the best we can. We check the legal moves that can be made -the calls that are not full- and change the values of the array accordingly. Then at each valid column, we check for each position that the bot can insert, we calculate its score using recursion by calling the minmax function until arriving to the base case inserted in the function. after the recursive step, we check whether the temp score is bigger than score and we save the best value in the 'best' variable by using the max function since this is the maximizing player. then we return the values to zeros -make the matrix empty- then we save the max number between alpha, and the evaluation score we calculated during the iteration of the loop.in case the beta value was less, we break it immediately. After all the steps are done, we return the value of best minus the depth we get from the arguments.Using the same logic, when the case now is for the minimizing player. we now try to minimize the score, so we set the best value to a very high number and start minimizing it as much as possible.

- findBestmove(State arr[ROWS][COLS]) function: Determines and predicts the player movements up to depth times, where the depth level is chosen depending on the difficulty. The bot will have the best legal position to put the next input into after calling both the legalmoves function and the minmax function.

- win(State board[ROWS][COLS]) function: check for the winner of the game by having either 4 adjacent player moves, of 4 adjacent bot moves (horizontally, vertically, and oblique).

- tieFull(State[ROWS][COLS]) function: loop over the board, in case the board was entirely filled up with entries other than 0 (which stands for empty), then a tie persists between both sides.

- tieTime() function: if the time taken between both sides was not equal, then the winner will be automatically selected as that that took less time throughout the game.

- legalmoves(State board[ROWS][COLS], position moves[COLS]) function: check and assign legal moves to the bot in order to insert input in an optimal criterion horizontally, vertically, or oblique inside the State board matrix.

- display(State board[ROWS][COLS]) function: print the main matrix which now contains all user and bot inputs

Testing Strategies:

- Avoid increasing the depth to a large value in order to avoid performance issues & lag
- Input at the middle of the board to make it easier to win the game
- Mix between inputting horizontally, vertically, and diagonally to test the bot's input system
- Race to place multiple inputs adjacent to each other to test the bot's blocking mechanics
- Race to block the bot's latest input in order to test the bot's optimal move AI

Test Cases:

TEST CASES:

Horizontal & Vertical:

[1,1,1,1] starting at indices 0,1,2,3 for all rows: check
[2,2,2,2] starting at indices 0,1,2,3 for all rows: check

```
[1]   [2]
[1] , [2]   starting at row 0,1,2 for all columns: check
[1]   [2]
[1]   [2]
```

Diagonal:

```
[1]           starting at indices 0,1,2,3 for rows 1,2,3
  [1]         it will check the first 3 rows and 4 columns accordingly
    [1]       for all the lower diagonals possible in the game
      [1]



      [1]     starting at indices 0,1,2,3 for rows 4,5,6
    [1]       it will check the last 3 rows and first 4 columns
  [1]         accordingly for all the upper diagonals possible in the game
[1]
```

same goes to the value [2]

Easy

TEST CASE: Attempt to win horizontally [1,1,1,1,2,..] =>win, [1,2,1,2,1,..] => nothing
TEST CASE: Attempt to win vertically

```
[1]   [2]
[1] , [2]   starting at row 0,1,2 for all columns: win
[1]   [2]
[1]   [2]
```

TEST CASE: Attempt to win diagonally

```
[1]           starting at indeies 0,1,2,3 for rows 1,2,3
  [1]         it will check the first 3 rows and 4 columns accordingly
    [1]       for all the lower diagonals possible in the game
      [1]     => win
      [1]     starting at indices 0,1,2,3 for rows 4,5,6
    [1]       it will check the last 3 rows and first 4 columns
  [1]         accordingly for all the upper diagonals possible in the game
[1]
```

TEST CASE: Attempt to block horizontally [1,1,1,2,...]
TEST CASE: Attempt to block vertically
[1]

```
[1]
[1]
[2]
[...]
```
TEST CASE: Attempt to block diagonally
```
[2]         starting at indices 0,1,2,3 for rows 1,2,3
  [1]       it will check the first 3 rows and 4 columns accordingly
    [1]     for all the lower diagonals possible in the game
      [1]   => block
      [2]   starting at indices 0,1,2,3 for rows 4,5,6
    [1]     it will check the last 3 rows and first 4 columns
  [1]       accordingly for all the upper diagonals possible in the game
[1]
```
Moves to capitulate

Medium

TEST CASE: Attempt to win horizontally [1,1,1,1,2,..] =>win, [1,2,1,2,1,..] => nothing

TEST CASE: Attempt to win vertically
```
[1]  [2]
[1] , [2]  starting at row 0,1,2 for all columns: win
[1]  [2]
[1]  [2]
```
TEST CASE: Attempt to win diagonally
```
[1]         starting at indices 0,1,2,3 for rows 1,2,3
  [1]       it will check the first 3 rows and 4 columns accordingly
    [1]     for all the lower diagonals possible in the game
      [1]   => win
      [1]   starting at indices 0,1,2,3 for rows 4,5,6
    [1]     it will check the last 3 rows and first 4 columns
  [1]       accordingly for all the upper diagonals possible in the game
[1]
```

TEST CASE: Attempt to block horizontally [1,1,1,2,...]

TEST CASE: Attempt to block vertically
```
[1]
[1]
[1]
[2]
[...]
```
TEST CASE: Attempt to block diagonally
```
[2]         starting at indices 0,1,2,3 for rows 1,2,3
  [1]       it will check the first 3 rows and 4 columns accordingly
    [1]     for all the lower diagonals possible in the game
      [1]   => block
      [2]   starting at indices 0,1,2,3 for rows 4,5,6
    [1]     it will check the last 3 rows and first 4 columns
```

```
 [1]      accordingly for all the upper diagonals possible in the game
[1]
```
Moves to capitulate

Hard

TEST CASE: Attempt to win horizontally [1,1,1,1,2,..] =>win, [1,2,1,2,1,..] => nothing

TEST CASE: Attempt to win vertically
```
[1]  [2]
[1] , [2]  starting at row 0,1,2 for all columns: win
[1]  [2]
[1]  [2]
```
TEST CASE: Attempt to win diagonally
```
[1]       starting at indices 0,1,2,3 for rows 1,2,3
 [1]      it will check the first 3 rows and 4 columns accordingly
  [1]     for all the lower diagonals possible in the game
   [1]    => win
   [1]    starting at indices 0,1,2,3 for rows 4,5,6
  [1]     it will check the last 3 rows and first 4 columns
 [1]      accordingly for all the upper diagonals possible in the game
[1]
```

TEST CASE: Attempt to block horizontally [1,1,1,2,...]

TEST CASE: Attempt to block vertically
```
[1]
[1]
[1]
[2]
[...]
```
TEST CASE: Attempt to block diagonally
```
[2]       starting at indices 0,1,2,3 for rows 1,2,3
 [1]      it will check the first 3 rows and 4 columns accordingly
  [1]     for all the lower diagonals possible in the game
   [1]    => block
   [2]    starting at indices 0,1,2,3 for rows 4,5,6
  [1]     it will check the last 3 rows and first 4 columns
 [1]      accordingly for all the upper diagonals possible in the game
[1]
```
Moves to capitulate

Out of bounds & invalid:

Handle invalid input when inserting into a column at out of bounds indices:
Prompt the user to re-enter a valid column index when the user enters:
- a negative number (e.g. -1)
- a number greater than the number of columns(6)

- a non-integer value (e.g. "a")

Handle invalid(NaN) input for placing balls: check