Edmund Zhi

805328871

Project 3 Report

**Description of Class Designs:**

* Scaffold: Scaffold is a dynamically allocated 2D array based on how the user wants to construct it’s columns and levels. Users can access how many levels and columns it has, the number of empty spaces, what checker is at a certain place, and it can display the scaffold. It uses stacks to determine whether or not a move can be made at a certain location, and if it can, the user can place a checker of a specified color in a certain column. It uses the same stack to keep track of moves to allow an undo move function to be implemented.
* Player: Player is an abstract base class. HumanPlayer is a derivative of this class. When it wants to make a move on a scaffold, it asks the user for an input as to which column it gain place. If the column is valid, a move is made. BadPlayer is also a derivative that always places a checker in the leftmost column. SmartPlayer incorporates the minimax algorithm to determine the best possible move on the scaffold. This function was combined with a hasWon function to determine if a game is won, lost, or tied, as well as an alarm clock to time the function out if the function runs for too long.
* Game: Game creates a game with a dynamically allocated scaffold and pointers to two players. When play is ran, the two players of the game alternate taking turns making moves on a scaffold until the game is over. Playing the game utilizes public functions such as completed, which determines if the game is done or not, and takeTurn, which takes a turn on the board.

**Description of SmartPlayerImpl::choosemove():**

First, we go through the columns of the scaffold and mark down each possible level where a move can be made. For each column, and for each of the next available level in said column, we make a move in each column at each of our recorded levels. If it causes us to win the game, we record this rating based on the depth into our possibilities tree that we have traveled. Winning is checked based on a helper function called hasWon, which takes our piece, and searches vertically, horizontally, and diagonally to see if this added piece is part of a connected N segment. If we do not win, we check for a tie. If we have not won and have not tied, we make a move for the other player and then make another move for ourself, continuing this process until we have found a result where the game is completed, where we then return a rating based on depth. The most positive rating that is returned is the best possible situation for us, one where we win the fastest or lose the slowest. However, if this searching takes too much time, we time out our program and return the best possible move we have found in the moment.

**Pseudocode for nontrivial algorithms:**

ScaffoldImpl::makeMove():

If the column is out of bounds or color input is incorrect

Return false

For each level in the selected column

If the level is the first vacant level

Add checker to this position in the scaffold

Record this position into a stack

Return true

Return false

ScaffoldImpl::undoMove():

If the grid is empty, return 0;

Check the top of the row and column stack for the most recent checker that was placed

Set the grid at that position to be vacant

Pop the row and column stack

Return the column that had the piece removed

SmartPlayerImpl::choosemove():

For each lowest available level in each column:

Make a move

If the move causes you to win:

Record the rating of this column based on depth of recursion

Otherwise:

If the move results in a tie game:

Record the rating as zero

Otherwise:

Remember this move and raise the lowest available level in the column

Continue making moves, alternating between players, until we have reached an ending, and remember the rating of that ending

Lower the lowest available level in the column

Undo all the moves just made to reach the ending

If the rating of the ending is the best possible one:

Set this number as the rating

Remember the column of this best possible move

Return the best possible move

GameImpl::takeTurn()

If the game is done

Return false

If the current player is Player 1

Announce the player’s turn

Make move

Set current player to Player 2

Else If current player is player 2

Announce player’s turn

Make move

Set current player to Player 1

Return true

GameImpl::play()

Display the game

While N in a row has not been connected:

Take a turn

Display the grid

Ask user to press enter to continue

Display the final grid

Announce the winner

**Bugs, Inefficiencies, or Notable Problems**