Artiom Arutiunov

Professor Smallberg

5/20/16

Project 3 Report

**Description of design of classes**

Class ScaffoldImpl:

I did not add any additional public or private functions to the ScaffoldImpl class. However, I did add additional private member variables: 2 integer variables, a 2-dimensional vector, a struct that contained 2 stacks and an object of the struct. The 2 integer variables were mColumns and mLevels, which corresponded to the dimensions of the playing field of the Connect N game. These variables were initialized via the ScaffoldImpl constructor. The 2-dimensional vector contained a vector of type integer within it. I called this 2-dimensional vector playBoard for clarity. Next, the struct I created was called moveHistory and in it were the 2 integer-containing stacks lastLevel and lastColumn onto which I pushed each corresponding player’s move for recording. Lastly, I the object call moveHistory that I used to actually push the last move’s coordinates.

In my constructor I initialized mColumns, mLevels, and playBoard via the initialization list. In the body of the constructor I created an object of moveHistory using the new keyword. In my numberEmpty() function, I used a nested for loop to visit each element within playingBoard and incremented a variable that I ultimately returned. In my checkerAt function, I first checked to make sure the passed parameters were valid (both > 0 since the user choses a column with a minimum value of 1) and then checked to see whether that corresponding coordinate contained a black or red checker. If there was no red or black checker in the coordinate, I returned the constant value “VACANT.” I also used a nested for loop for my display function, but the outer for loop worked backwards and was decremented while the inner for loop was incremented. In effect the playing board was drawn as the user would see it instead of being upside down. In my makeMove function, I first checked to make sure the column was greater than 0 and that there was at least one vacant spot in that column choice, and then found the lowest empty level in that corresponding column. I then assigned the respective color to that coordinate and pushed the corresponding column and level onto the respective stacks using the recorder object. In my undoMove function, I first checked to make sure that both stacks in moveHistory were not empty (to avoid a compilation error) and then retrieved the column and level from the stacks before popping them.

Class HumanPlayerImpl

Since HumanPlayerImpl is a derived class of the Player class, I only had to implement 2 functions that were unique to HumanPlayerImpl: the chooseMove function and the isInteractive function. I did not create any additional member functions or members that were unique to HumanPlayerImpl. For the chooseMove function, I prompted the user to enter a column number until it was a valid column. I then returned that input value. All I had to do for the isInteractive was return true since it’s the human player’s class.

Class BadPlayerImpl

Since BadPlayerImpl is a derive class of the Player class, I only had to implement 1 function: chooseMove. The chooseMove function consisted of 1 for loop that iterator until there was a column open (at least the top level of it was open). Otherwise, I returned -1.

Class SmartPlayerImpl

Although I was not able to implement the recursive model that was provided in the spec, I managed to create a SmartPlayer intelligent enough to stop most of the opponent’s winning moves. Alongside the chooseMove function, I created and implemented 2 other public functions in the SmartPlayer class: ratingSystem, which is similar to the complete function I use in game to determine the outcome of the game following every move, and the function immediateMove, which incorporates ratingSystem to determine whether an immediate move by the computer should be made to either win or lose the game. I also created some predefined constants called WIN, TIE, and LOSS for clarity.

Class GameImpl

I did not create any additional functions in the GameImpl class that weren’t found in the Game class. However, I did add several private data members: an object of Scaffold called board, an integer variable called connectGoal, an integer variable called lastColumn (initialized to 0), a boolean variable called redTurn (initialized to true), and two pointers of the Player type called Red and Black. The Scaffold object board was what I used to call all of Scaffold’s methods and treated it as the playing board of the game. The variable connectGoal was the numerical value of how many checkers in a row the player’s were trying to get (the “N” in “Connect N”). The boolean variable redTurn was what I used to determine whether it was the red or black player’s turn to play. Lastly, the two pointer of Player was what I used to call the appropriate methods for the player, which obviously differed depending on whether the player was human or not.

In my constructor I initialized all the member variables not initialized in the header file via the initialization list. Next, for the completed function I first determined what color I was looking for, then determined what was the level the last played checker was dropped to, then I checked the 7 different cases a player could win by using a for loop and incrementing a counter I would reset after each independent winning method check. If none of the 7 winning methods incremented the checker enough to win, I assigned TIE\_GAME to winner and returned false. For the takeTurn function, I first checked whether it was the red player’s turn, then recorded the player’s column choice, then called the Scaffold class’ makeMove function, then assigned the opposite Boolean value to redTurn, and ended by returning true. The same case applied for the black player except I assigned redTurn the value of true. For the play function, I checked whether both players were interactive, and if they weren’t I ran a while loop checking the completed function to be false and that the board wasn’t out of empty spots. I then called board’s display function and displayed the grid after every turn. If both players were not interactive (meaning both were computers) I ran a similar while loop but prompted the user to press enter every 5 moves. After either one of these while cycles was completed, I determined what color the winner variable corresponded to and printed out an appropriate win or tie message. Lastly, for GameImpl’s checkerAt method I simply called board’s checkerAt function (they behave exactly the same).

**Description of SmartPlayer::chooseMove**

**Psuedocode for non-trivial algorithms**

Psudeocode for ScaffoldImpl::checkerAt(int column, int level) const

If the column and level are greater than 0

If the play board (vector) at that point is RED

return RED

Else if the play board (vector) at that point is BLACK

return BLACK

return VACANT

Psuedocode for ScaffoldImpl::display() const

For the amount of levels until we reach 0

Print out left-hand “|”

For the amount of levels until we reach 1 less than total amount of columns

If the play board (vector) at that point is RED

Print out “R”

Else if the play board (vector) at that point is BLACK

Print out “B”

Else if the play board at that point is VACANT

Print out “ “

Print out right-hand “|”

Print out an endline

For the amount of columns until we reached 1 less than total amount of columns

Print out “+-“

Print out one more “+”

Psuedocode for ScaffoldImpl::makeMove(int column, int color)

If the column is greater than 0 and the coordinate at that point is VACANT

Determine which level the checker was dropped to in that column

At that coordinate in the playing board (vector) assign the appropriate color

Push that column value onto the lastColumn stack via recorder object

Push that level value onto the lastLevel stack via recorder object

Return true

Return false

Psuedocode for HumanPlayerImpl::chooseMove(const Scaffold& s, int N, int color)

Create an integer V

Print out message for user to enter column

Do loop

Take user’s input

While the input is <= 0 or is > than the amount of columns or the column is already full

Return V;

Psuedocode for BadPlayerImpl::chooseMove(const Scaffold& s, int N, int color)

For loop until one we are one less than amount of columns

If the checker at that point is vacant

Return column + 1 (accounting for user input as 1 greater than vector coordinate)

Return -1

Psuedocode for SmartPlayerImpl: SEE ABOVE

Psuedocode for GameImpl::completed (int& winner) const

Create variable color

Determine what most recently played checker’s player’s color is

Assign RED or BLACK to color variable

Determine the actual level of where checker was dropped in respective column

Assign this value to a variable called lastLevel

Create integer variable counter = 1

// checking RIGHT DIAGONAL CASE (up & down)

For loop with integer i = 1; while <= N (connect Goal) and the lastColumn + i is <= the amount of columns and the lastLevel + i is <= the amount of levels; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

For loop with integer i = 1; while i <= connectGoal and the amount of columns – i is greater than 0 and the amount of levels – i is greater than 0; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

// checking LEFT DIAGONAL CASE (up & down)

Reset counter variable to 1

For loop with integer i = 1; i <= connectGoal and the amount of columns – i is greater than 0 and the lastLevel + i is <= the amount of levels; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

For loop with integer i = 1; i <= connectGoal and the amount of columns + i is <= the amount of columns and the amount of levels - i is > 0; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

// checking the DOWNWARDS CASE

Reset counter to equal 1

For loop with integer i = 1; i is <= connectGoal and the amount of levels – i is > 0; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

// checking the HORIZONTAL CASE (left and right)

For loop with i = 1; i is <= the connetGoal and lastColumn – i is > 0; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

For loop with i = 1; i is <= the connetGoal and lastColumn + i is <= the amount of columns; increment i

If the checker at that point is the target color

Increment the counter

Else

Break out of the for loop

If the counter is equal to the connectGoal

Assign color to the winner integer

Return true

Assign TIE\_GAME to winner if none of the winning combinations increment enough

Return false

Psuedocode for GameImpl::takeTurn()

If it is red’s turn to play

Call Red player’s choseMove function and assign resulting column to variable lastColumn

Call Scaffold object board’s makeMove function with corresponding color and column

Set redTurn variable to false

Return true

Else

Call Black player’s choseMove function and assign resulting column to variable lastColumn

Call Scaffold object board’s makeMove function with corresponding color and column

Set redTurn variable to true

Return true

Psuedocode for GameImpl::play()

Create an integer called winner with initial value of 0

If both the Red player and Black player are not interactive

While the game is not completed

Display the current board (scaffold)

Prompt the user to press enter every few moves

Call game’s takeTurn function

Else

While the game is not completed and the playing board is not full

Display the current board (scaffold)

Call game’s takeTurn function

Display the board one more time

If the winner is red

Print out appropriate message

Else if the winner is black

Print out appropriate message

Else

Print out that the game is a tie

Psuedocode for SmartPlayer::chooseMove

Create a temporary scaffold

Check if there is an immediate winning move to be made

If there is, make it

Check if the other player is about to make a winning move

If they are, block that move

If there is no winning move to be made from either side, drop a checker in the leftmost free column

**Known bugs/ serious inefficiencies/ notable problems**

Known bugs

One of the known bugs I had in my code for the longest time was dealing with is integrating the functions between the user’s input (always 1 greater than what the corresponding coordinate is in the vector playing field) and the internal functions. I would get a lot of bad excess messages as a result. In addition, I had trouble having the moves actually appear on the playing field after every turn, as I had to configure the user’s input with how the board actually displayed and registered the move.

Serious inefficiencies

One inefficiency that I am sure could be fixed is my complete function in the Game class. My approach for checking whether the player has won after every move is to check each one of the possible 7 winning ways by using an iterative algorithm and incrementing a variable. There can surely be a quicker way to do this, such as starting at one point between two common directions of winning (say bottom left-hand corner for a northwest and southeast win) and working through the entire path once instead of twice.

Notable problems

One of the notable problems I encountered while working on the project was incorporating the user’s column choice with my internal code, since the user always chose a column that was 1 greater than what its corresponding location was in the actual vector in the Scaffold class. Thus, I initially had problems converting this to code and would get bad access errors, until I realized that an efficient way to solve the problem was to simply delegate all the conversion to functions of the Scaffold class, and treat the Game class’ function implementations with the user’s raw input. That way, the code became a lot cleaner in Game and I didn’t have to worry about conversion.

Another notable problem I encountered was implementing the complete function in the Game class. The reason I initially had problems with this function was because I wasn’t sure how to keep my checking in boundaries, until I realized that it is precisely the target amount “N” that should determine how many steps forward and backward I should be looking. Furthermore, I grouped similar winning patterns together for efficiency: up and down in the right diagonal direction were grouped together, as were up and down in the left diagonal direction, and the right and left directions for checking if the player won horizontally.

The last and without a doubt the most difficult aspect of this project was SmartPlayer. Although I was not ultimately able to implement it with the recursive algorithm provided in the spec, I made the player smart enough to tie the game in most situations. Obviously the problem with this model is that the computer is still relatively dumb, as it can only look 1 more ahead for whether the other player is about to win.

**Test cases**

Testing ScaffoldImpl

Testing a regular Scaffold setting:

Scaffold s(3, 2);

assert(s.cols() == 3 && s.levels() == 2 &&

s.numberEmpty() == 6);

assert(s.makeMove(1, RED));

assert(s.makeMove(1, BLACK));

assert(!s.makeMove(1, RED));

assert(s.numberEmpty() == 4);

assert(s.checkerAt(1, 1) == RED && s.checkerAt(1, 2) == BLACK);

assert(s.checkerAt(2, 1) == VACANT);

s.display();

Testing Scaffold with invalid parameters (should terminate program):

Scaffold test1(0, 2);

Testing Scaffold with invalid column inputs

Scaffold test2(5, 5);

assert(test2.cols() == 5 && test2.levels() == 5 &&

test2.numberEmpty() == 25);

assert(!test2.makeMove(0, RED)); // can't make a move out of bounds to the left

assert(!test2.makeMove(6, RED)); // can't make a move out of bounds to the right

assert(test2.makeMove(1, RED));

assert(test2.checkerAt(1, 1) == RED);

assert(test2.numberEmpty() == 24);

test2.undoMove();

assert(test2.numberEmpty() == 25);

for (int i = 1; i < 6; i++)

test2.makeMove(2, BLACK);

assert(!test2.makeMove(2, BLACK)); // column is full so can't move there

test2.display();

Testing Scaffold’s undo move function

Scaffold test3(3, 3);

for (int i = 1; i <= 3; i++)

test3.makeMove(1, BLACK);

test3.display();

assert(test3.numberEmpty() == 6);

assert(test3.undoMove() == 1);

assert(test3.numberEmpty() == 7);

assert(test3.undoMove() == 1);

assert(test3.numberEmpty() == 8);

assert(test3.undoMove() == 1);

assert(test3.numberEmpty() == 9);

assert(test3.undoMove() == 0); // nomore moves to undo

assert(test3.checkerAt(25, 100) == VACANT); // should still return vacant even though it's out of scope

Testing Scaffold for having its values initialized to VACANT

Scaffold test4(6, 6);

for (int i = 1; i <= 6; i++)

for (int j = 1; i <= 6; j++)

{

assert(test4.checkerAt(i, j) == VACANT); // all values initialized to vacant

}

Testing Scaffold’s undoMove function further

Scaffold test5(5, 5);

for (int i = 1; i <= 5; i++)

test5.makeMove(i, BLACK);

assert(test5.undoMove() == 1);

assert(test5.undoMove() == 2);

assert(test5.undoMove() == 3);

assert(test5.undoMove() == 4);

assert(test5.undoMove() == 5);

Testing Player classes

Testing all 3 Player classes for valid column input

Scaffold s(3, 2);

s.makeMove(1, RED);

s.makeMove(1, BLACK);

cout << "=========" << endl;

int n = hp.chooseMove(s, 3, RED);

cout << "=========" << endl;

assert(n == 2 || n == 3);

n = bp.chooseMove(s, 3, RED);

assert(n == 2 || n == 3);

n = sp.chooseMove(s, 3, RED);

assert(n == 2 || n == 3);

Testing HumanPlayer class for valid column input

Scaffold s2(5, 5);

HumanPlayer human1("human1");

assert(human1.name() == "human1" && hp.isInteractive());

int n1 = human1.chooseMove(s2, 3, BLACK);

assert(n1 >= 1 && n1 <= 5);

TestingBadPlayer class for valid column input

Scaffold s3(5, 5);

BadPlayer bp1("badplayer1");

BadPlayer bp2("badplayer2");

while (s3.numberEmpty() != 0) // checker should exactly alternate

{

int i = bp1.chooseMove(s3, 5, RED);

s3.makeMove(i, RED);

i = bp2.chooseMove(s3, 5, BLACK);

s3.makeMove(i, BLACK);

}

Testing Game class

Testing the general flow of the game through game’s specific functions (completed, checkerAt, and takeTurn)

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Game g(2, 2, 2, &bp1, &bp2);

int winner;

assert(!g.completed(winner));

g.takeTurn(); // Red's first move

assert(!g.completed(winner) &&

(g.checkerAt(1, 1) == RED || g.checkerAt(2, 1) == RED));

g.takeTurn(); // Black's first move

assert(!g.completed(winner));

g.takeTurn(); // Red's second move; Red must win

assert(g.completed(winner) && winner == RED);

Testing a general game between one HumanPlayer and one BadPlayer

HumanPlayer hp1("HumanPlayer1");

BadPlayer bp\_1("BadPlayer1");

Game g2(5, 5, 3, &hp1, &bp\_1);

int winner\_1 = 0;

assert(!g2.completed(winner\_1));

g2.play();

Testing a general game between two BadPlayers (also checking for appropriate “press enter to continue” message)

BadPlayer bp\_2("BadPlayer2");

BadPlayer bp\_3("BadPlayer3");

Game g3(5, 5, 4, &bp\_2, &bp\_3);

g3.play();

Testing a general game between two HumanPlayers and whether the game works correctly under these circumstances

HumanPlayer hp2("HumanPlayer2");

HumanPlayer hp3("HumanPlayer3");

Game g4(7, 7, 5, &hp2, &hp3);

g4.play();

Testing a general game between one HumanPlayer and one SmartPlayer to see whether the game works correctly under these circumstances

HumanPlayer hp("Artiom");

SmartPlayer sp("Kedar");

Game g(5, 5, 4, &hp, &sp);

g.play();