Tyler Stovsky

705512370

Design of my classes:

Scaffold:  
For my scaffold design, the main data structure I used to represent the board was a vector<vector<char>>. When initializing this, I created an nLevels by nColumns board with every character set to ‘ ’. When designing the checkerAt function, I just looked at the correct position in the vector and determined if it was an ‘R’, ‘B’, or otherwise vacant. For my makeMove function, I first checked if the column was vacant and, if it was, I traversed down the column until I found the closest empty spot to the bottom, and changed the character from ‘ ’ to the correct color. When displaying the scaffold, I just added ‘|’ for every even index and then at the bottom added ‘+’ for an even index and ‘-‘ for an odd index. To create undoMove, I used a stack data structure. While making a move, I pushed the column and the level onto the stack, so when I undoMove all I had to do was store the first two items on the stack, pop them both off, then change the scaffold at that spot back to ‘ ’. For numberEmpty, I simply counted the times ‘ ’ appears in the scaffold.

Game:

For my game design, the major items include a scaffold, a bool which tells me which player’s turn it is, and the two players. For my completed function, I created three helper functions which each check for a winner in different directions. The first is checkHorizontalWinner, the second is checkVerticalWinner, and the third is checkDiagonalWinner (which checks for both diagonals going north-east and south-east). These functions will return the winner if one is found, or else it will return -10, so my completed function can check if the function doesn’t return -10, it knows that there is a winner. If there is no winner but the empty squares is 0, this means we have a tie game. Every time the takeTurn function is called, it first will check for a winner, and if there is it will return right away, but otherwise it will make the move for the correct player’s turn, and then change the bool so the next call will switch the player’s turn. My play function first displays the empty board, and then will call takeTurn and then display the board repeatedly until a winner is found or there is a tie game. If two computers are playing, it will prompt the user to press enter between each move.

Player:

For my human player implementation, all that I did was prompt the user to input a column, and then check if that input was valid. An input is valid if it is between 1 and the number of columns, and if that column has a vacant spot. If the input is invalid, I re-prompt the user.

For my bad player implementation, I went from left to right and looked for the first vacant column and returned it. If there is no vacant column, I returned immediately.

SmartPlayer::chooseMove:

For this implementation, I had many helper functions which I will break down individually.

I have two ints, int COMPUTER and int HUMAN, which represent which color the computer is and which the human is (although human could be another computer, this will just represent the other player that we are trying to find the best move against).

Void SmartPlayerImpl::setColors(int color):

This function takes in the color provided from the chooseMove function, and sets it as the computers color. It will then set the opposite color as the humans color.

Bool SmartPlayerImpl::checkWin(const Scaffold& s, int player, int m\_N):

This function is almost identical to my completed function (and helper functions) from the game class, but it will actually check a win for a specific player. This is very useful for my minMax function because it can check if a move made from a specific player will result in a win for that player. This is used to either avoid this move or make this move.

Scaffold SmartPlayerImpl::copyScaffold(const Scaffold& s):

This function will create a new scaffold that is identical to the games scaffold. This is useful for my minMax function because I can make moves on a new scaffold without worrying about altering the real one.

Int SmartPlayerImpl::score(const Scaffold& s, int player, int m\_N):

This function looks from every possible direction, and scores every single possible set of m\_N (which is the amount of checkers in a row needed for a win). First it looks at every horizontal set, traversing from left to right, where it sends every single one to be scored. Then it looks at every vertical set, traversing from bottom to top, where it again sends every single one to be scored. Finally it looks at both diagonals (going north-east and south-east), where it does the same thing. This will make more sense when paired with its other two functions that go hand-in-and with it, setScore and eval.

Int SmartPlayerImpl::setScore(vector<int> checkers, int player, int m\_N):

This function takes in the set provided from the score function (which here is called checkers) and then uses three variables to count what is actually in the set. The three variables are int good, int bad, and int empty. If the set contains the same checker as the player provided (this could be COMPUTER or HUMAN), it will add 1 to good. If it contains the opposite checker as the player provided, it will add 1 to bad. If it contains a vacant spot, it will add 1 to empty. Once it figures out how many good, bad, and empty checkers there are, it will send this information to the eval function for a final scoring of the set.

Int SmartPlayerImpl::eval(int good, int bad, int empty, int m\_N):  
The way my scoring system works is that it checks the amount of good checkers, bad checkers, and empty checkers in a set and then rates the position based on that. Here is a chart for my scores:

Amount in a row to win = 2

|  |  |  |  |
| --- | --- | --- | --- |
| Good checkers | Bad Checkers | Empty Space | Score |
| 2 | 0 | 0 | 5050 |
| 1 | 0 | 1 | 500 |
| 0 | 1 | 1 | -550 |
|  | 2 | 0 | -5000 |

Amount in a row to win = 3

|  |  |  |  |
| --- | --- | --- | --- |
| Good checkers | Bad Checkers | Empty Space | Score |
| 3 | 0 | 0 | 50050 |
| 2 | 0 | 1 | 5000 |
| 1 | 0 | 2 | 500 |
| 0 | 1 | 2 | -550 |
| 0 | 2 | 1 | -5050 |
| 0 | 3 | 0 | -50000 |

Amount in a row to win >= 4 (this value is m\_N)

For (int k = 0; k < m\_N - 1; k++)

|  |  |  |  |
| --- | --- | --- | --- |
| Good checkers | Bad Checkers | Empty Space | Score |
| m\_N | 0 | 0 | 5\*10^(m\_N + 1) + 50 |
| m\_N - 1 | 0 | 1 | 5\*10^(m\_N) |
| m\_N - k | 0 | k | 5\*10^(m\_N - k) |
| 0 | m-N - k | k | -(5\*10^(m\_N - k) + 50) |
| 0 | m\_N - 1 | 1 | -(5\*10^(m\_N) + 50) |
| 0 | m\_N | 0 | -5\*10^(m\_N + 1) |

Vector<int> SmartPlayerImpl::minMax(const Scaffold& s, int depth, int alpha, int beta, int player, int m\_N, Alarmclock& ac):

This is my main helper function for chooseMove, which does the minMax algorithm to calculate which column to return. It returns a vector<int> because it needs to return two values which are necessary to the algorithm. The first value in the vector is the score, and the second is the current best column. The function starts by setting the best move to the first vacant column, which is good in case we can’t find a best move for some reason. Once the depth hits zero in the recursive algorithm, it will return the score and the column. It starts by trying to maximize the computers score by going through each column and playing a move in a temporary copy of the scaffold. It then evaluates this new position by recursively calling the function for the other player, where it tries to minimize the other player’s score. It does this for the set depth. After completing the recursive calls, it checks if the new score is better than the old score, and if it is that means we have found a new best move and it will set the second index to the new best column. I also have implemented alpha-beta pruning, which is where I try to get rid of moves that can’t possibly be correct. This makes my algorithm faster.

Int SmartPlayerImpl::chooseMove(const Scaffold& s, int N, int color):

In the actual chooseMove function, all I do is create an alarm clock with a time set to 9900, call the setColors function in order to set the correct colors to each player, and then I return my minMax function with the arguments minMax(s, 8, 0 - INT\_MAX, INT\_MAX, COMPUTER, N, ac), where I set the depth to 8 because I wanted the player to be smart but not too slow to think. I return the 1st index of this function because that is the best column in the vector I return.

Pseudocode:

**Scaffold:**

**void ScaffoldImpl::display() const:**

Loop through the board:  
 If the index is even, display a ‘|’

Otherwise, display the checker at the current position

Loop through the columns:

If the index is even, display a ‘+’

Otherwise display a ‘-‘

**int ScaffoldImpl::checkerAt(int column, int level) const:**

If the column and level are valid:

If the spot at the column, level is ‘R’ return RED

If the spot at column, level is ‘B’ return BLACK

Otherwise return VACANT

**bool ScaffoldImpl::makeMove(int column, int color):**

Check if column has a vacant spot

If not, return false

Convert color’s integer to its ASCII value

Traverse up the column:

If the spot is vacant:  
 Set the spot to the color

Push the column and level onto a stack

Break

**int ScaffoldImpl::undoMove():**

If no moves have been made, return

Store column and level from stack

Pop column and level off the stack

Replace that column and level with a vacant space

Return the column

**Game:**

**int GameImpl::checkDiagonalWinner() const:**

Loop through the scaffold:

At each spot go down 1 and right 1:

Count the number of the same color in a row

If N in a row:

Return that color

Otherwise reset the counter back to 0

Loop through the scaffold:

At each spot go up 1 and right 1:

Count the number of the same color in a row

If N in a row:

Return that color

Otherwise reset that counter back to 0

**int GameImpl::checkHorizontalWinner() const:**

Loop through the scaffold:

At each spot go right 1:

Count the number of the same color in a row

If N in a row:

Return that color

Otherwise reset the counter back to 0

**int GameImpl::checkVerticalWinner() const:**

Loop through the scaffold:

At each spot go up 1:

Count the number of the same color in a row

If N in a row:

Return that color

Otherwise reset the counter back to 0

**bool GameImpl::completed(int& winner) const:**

If a horizontal, vertical, or diagonal winner, set winner return true

Else, if there are no empty spots, set winner to tie game and return true

Otherwise, the game isn’t over and return false

**bool GameImpl::takeTurn():**

If the game is completed, set winner and return true

If it’s red’s turn:

Make red’s move

Otherwise, make black’s move

**void GameImpl::play():**

Display empty board

While the game isn’t over:

Take someone’s turn

Display the board

If two computers are playing:

Prompt the user to press enter in between moves

Announce the winner of the game

**Player:**

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

If no spots are available, do nothing

While the input is invalid:

Ask the user for input

If the input is within the columns:

If there is a vacant spot in the column, return input

Otherwise, if there is no vacant spot then set the input to be invalid

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

If no spots are available, do nothing

Loop through scaffold from left to right:

If the column has a vacant space, return that column

**Scaffold SmartPlayerImpl::copyScaffold(const Scaffold& s):**

Create a new empty scaffold

Loop through current scaffold:

Make same move in new scaffold as the current one

**bool SmartPlayerImpl::checkWin(const Scaffold& s, int player, int m\_N):**

Loop through the scaffold:

At each spot go right 1:

Count the number of the player’s color in a row

If N in a row:

Return true

Otherwise reset the counter back to 0

Loop through the scaffold:

At each spot go up 1:

Count the number of the player’s color in a row

If N in a row:

Return true

Otherwise reset the counter back to 0

Loop through the scaffold:

At each spot go down 1 and right 1:

Count the number of the player’s color in a row

If N in a row:

Return true

Otherwise reset the counter back to 0

Loop through the scaffold:

At each spot go up 1 and right 1:

Count the number of the player’s color in a row

If N in a row:

Return true

Otherwise reset that counter back to 0

**vector<int> SmartPlayerImpl::minMax(const Scaffold& s, int depth, int alpha, int beta, int player, int m\_N, AlarmClock& ac):**

If no vacant spots, do nothing

Find first open column in the scaffold

If depth is zero:

Return current score and first open column

If it’s the computers turn:

Create a vector with the lowest possible score and the first empty column

If the other player is going to win:

Return worst possible score

Loop through each possible move:  
 Create a copy of the board and try the move

Find the move based on the state of the new board

If we find a better score:

Replace the score

Set new best move

Set alpha as the max of alpha and the current score

If alpha is greater than or equal to beta:

Break

Return the vector contacting the score and current best move

Otherwise, it’s the other player’s turn:

Create a vector with the highest possible score and the first empty column

If the computer is going to win:

Return best possible score

Loop through each possible move:  
 Create a copy of the board and try the move

Find the move based on the state of the new board

If we find a worse score:

Replace the score

Set new worst move

Set beta as the min of beta and the current score

If alpha is greater than or equal to beta:

Break

Return the vector contacting the score and the current worst move

**void SmartPlayerImpl::setColors(int color):**

Set computer to the color

If computer is RED:

Set human to BLACK

Otherwise set human to RED

int SmartPlayerImpl::chooseMove(const Scaffold& s, int N, int color):

Set the colors

Return the best column from the minMax algorithm

**int SmartPlayerImpl::eval(int good, int bad, int empty, int m\_N):**

// N is the amount of checkers needed in a row for a win

If we need 2 in a row to win:  
 If there is 2 good checkers in a row, add the highest score

If there’s 1 good checker in a row and 1 empty, add second highest score

If there’s 1 bad checker in a row and 1 empty, add second worst score

If there’s 2 bad checkers in row, add the worst score

If we need 3 in a row to win:

If there is 3 good checkers in a row, add the highest score

If there’s 2 good checkers in a row and 1 empty, add second highest score

If there’s 1 good checker in a row and 2 empty, add third highest score

If there’s 1 bad checker in a row and 2 empty, add third worst score

If there’s 2 bad checkers in a row and 1 empty, add second worst score

If there is 3 bad checkers in a row, add the worst score

If we need 4 or more in a row to win:

Loop until we get down to 2 empty spaces

If there’s N good checkers in a row, add highest score

If there’s N - 1 good checkers and 1 empty space, add second highest score

If there’s N - k good checkers and k empty spaces, add k + 1 highest score

If there’s N - k bad checkers and k empty spaces, add k + 1 worst score

If there’s N - 1 bad checkers and 1 empty space, add second worst score

If there’s N bad checkers, add worst score

**int SmartPlayerImpl::setScore(vector<int> checkers, int player, int m\_N):**

Loop through set of checkers:

If we see the player’s checker, increase good checkers by 1

If we see either player’s checker, increase bad checkers by 1

If we see a vacant space, increase empty by 1

Subtract good from bad to get just the bad

Evaluate the good, bad, and empty checkers

**int SmartPlayerImpl::score(const Scaffold &s, int player, int m\_N):**

// In this function, we are finding every single possible set of m\_N and scoring it

Loop through the scaffold horizontally:

Store the entire level in a vector

Break up the vector into a set of m\_N:

Score the set

Loop through the scaffold vertically:

Store the entire column in a vector

Break up the vector into a set of m\_N:

Score the set

Loop through the scaffold going NE:

Store the entire diagonal in a vector

Break up the vector into a set of m\_N:  
 Score the set

Loop through the scaffold going SE

Store the entire diagonal in a vector

Break up the vector into a set of m\_N:

Score the set

**Notes:**

A serious bug/inefficiency I have is that sometimes my minmax algorithm will fail to find a best column, and therefore it will just return the first empty space. I was trying to figure out for a long time why this happened, but I ran out of time.

I had many problems while creating this, but the biggest was definitely trying to learn and understand how to create the SmartPlayer. This was completely new to me and took many hours for me to understand what was going on. After I finally got it to work, I had to play around with the rating function for a long time because sometimes it would love to block and hate to win when it had the chance, and other times it would hate to do either. It still isn’t perfect but I have it at a point where it works pretty well.

I also was getting strange errors when I had big columns and small levels or vice versa (like a 10x1). Xcode and g32 weren’t really telling me exactly what was going on, but I eventually found out that I accidentally was building a vector of size columns to store levels, and a vector of size levels to store columns, which obviously won’t work but when we have, for example, a 7x6 where the numbers are close, this issue doesn’t occur if you don’t use all the space in the vector.

I also had an issue with my completed function when I was trying to check for a win, and the issue came to be that my checkerAt function was looking at indices out of bounds, so I needed to add extra constraints to my for loops.