# Project 3 Report

Overview:

The project is comprised of Game, Scaffold, and Player class which interact with each other.

Game controls the overall sequence of the game, using Scaffold and Player to control the display and logic while playing.

The major data structures used in this project was stacks, vectors, and lists. They were mainly used to store information about the state of the game, such as moveHistory.

The biggest feature of this project is the use of recursion in SmartPlayer’s chooseMove function. It is explained in detail later.

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

*Create an AlarmClock set for 8750ms*

*Return the value of findBestMove(s, N, color, ac)*

This function starts out by creating an AlarmClock, then calls and returns the value of findBestMove function, which does all the work.

The value of 8750ms was chosen because when testing the code on the school’s linux servers, using the release configuration, this value ensured running time was below 10 seconds. Essentially, it was an extra safe value when running the benchmarks.

I will first explain the other utility functions that were created to assist findBestMove:

Firstly, I created an enum called State, which include, REDWIN, BLACKWIN, TIE, and PLAYING.

This first function is the following, which returns a state depending on the Scaffold supplied to it. It is pretty much identical to Game’s completed function:

state SmartPlayerImpl::checkState(const Scaffold& s, int N)

*Loop through all possible locations in the Scaffold*

*Conduct horizontal checks for each checker, if there are atleast N number in a row, return a state corresponding to the winning color*

*Conduct vertical checks for each checker, if there are atleast N number in a row, return a state corresponding to the winning color*

*Conduct up-right diagonal checks for each checker, if there are atleast N number in a row, return a state corresponding to the winning color*

*Conduct down-right checks diagonal for each checker, if there are atleast N number in a row, return a state corresponding to the winning color*

*If Scaffold is full, return TIE*

*Return PLAYING*

Following this function, I made a function that evaluates a state, return a score. Observe below:

int SmartPlayerImpl::evaluateState(const Scaffold& s, int N, int color, int depth, AlarmClock& ac)

*If AlarmClock timed out, return 0*

*Declare a currentState and assign it to the value of checkState(s, N)*

*If currentState is not PLAYING, meaning the game concluded in some way*

*If the state is REDWIN and color is RED, or if its BLACKWIN and color is BLACK (this player has won)*

*Return 100000 – depth*

*Else if the state is REDWIN and color is BLACK, or if its BLACKWIN and color is RED (this player has lost)*

*Return -100000 + depth*

*Else if the state is TIE*

*Return 0;*

*If you reach this point, game is still ongoing, so return -1*

This function scores a state of the game according to whether it’s a win, return a large number, or a loss, returning a large negative number. Note how these return values are also adjusted with depth. This function will be an integral part of the minimax algorithm.

Now let’s start from the beginning with the findBestMove function, which finds the best move by calling the minimax algorithm in each playable column:

int SmartPlayerImpl::findBestMove(const Scaffold& s, int N, int color, AlarmClock& ac)

*Make a copy of Scaffold s and store it*

*Declare int oppColor as the opposite of color*

*Declare and initialize bestScore with a large negative number*

*Create a vector of ints, called bestMoves, which will, as the name suggests, store the bestMoves*

*Create a list of ints, called possibleMoves, which will, as the name suggests, store a list of the possible moves given the scaffold*

*Loop through all columns in the Scaffold*

*If AlarmClock has expired*

*If bestMoves is empty, return first item from possibleMoves*

*Return first item from bestMoves*

*If it is possible to play the column with color*

*Play the column*

*Declare and initialize an int called score, which will be -miniMax(s, N, oppColor, 0(depth), ac) This will use the minimax algorithm to determine a score*

*If this score is greater than bestScore*

*bestScore takes on score’s value*

*clear bestMoves vector*

*store current column into bestMoves*

*Else if score is equal to bestScore*

*Store current column into bestMoves*

*Undo the move*

*If bestMoves vector is empty, return the first item from possibleMoves list*

*Return the first item from bestMoves*

Now I will explain the minimax function, which returns a score by recursively calculating the winning possibilities of the move.

int SmartPlayerImpl::miniMax(const Scaffold& s, int N, int color, int depth, AlarmClock& ac)

*If the AlarmClock has timed out*

*Return 0;*

*Declare int oppColor as the opposite of color*

*Declare and initialize an int called moveVal which is equal to the score of the scaffold, using evaluteState function described earlier*

*If moveVal isn’t -1, meaning game has concluded in some way, return moveVal (This is the terminal node)*

*Declare and initialize an int called bestScore with a large negative number*

*Loop through all columns*

*If the AlarmClock has timed out*

*Return 0;*

*If the current column is playable*

*Play the column*

*Declare and initialize an int called score, which will be -miniMax(s, N, oppColor, depth + 1, ac) This will recursively call minimax*

*If score is greater than bestScore*

*bestScore takes on score’s value*

*Undo the move*

*Return bestScore*

That explains my chooseMove function for the SmartPlayer. Using a minimax algorithm, it evaluates each possible move while adjusting for depth. Then it stores the best moves and plays them. If time runs out, it will play the first best move it found, if there are none, it plays the first possible move.

Pseudocode for other Non-Trivial Algorithms:

ScaffoldImpl::ScaffoldImpl(int nColumns, int nLevels)

*If nColumns and nLevels are valid*

*Construct a 2D vector composed of chars with a width = 2 \* columns + 1, and a height = levels + 1.*

*Add bottom row comprised of alternating ‘+’ and ‘-‘, with ‘+’ an the edges.*

*Add in top row comprised of alternating spaces and ‘|’. Then loop through all rows copying the above rows excluding the bottom row.*

*Else, invalid Scaffold error*

ScaffoldImpl::undoMove()

*If moveHistory stack is empty, return 0.*

*Using the top value on the moveHistory stack, which represents the last column played,*

*Loop to the top most checker in the column and delete it.*

*Pop moveHistory*

*Return the column*

GameImpl::completed(int& winner)

*Loop through all possible locations in the Scaffold*

*Conduct horizontal checks for each checker, if there are atleast N number in a row, Set winner to the corresponding color and return true*

*Conduct vertical checks for each checker, if there are atleast N number in a row, Set winner to the corresponding color and return true*

*Conduct up-right diagonal checks for each checker, if there are atleast N number in a row, Set winner to the corresponding color and return true*

*Conduct down-right checks for each checker, if there are atleast N number in a row, Set winner to the corresponding color and return true*

*If Scaffold is full, set winner to TIE\_GAME and return true*

*Return false*

Notable Bug Fixes:

* (May 13th) Scaffold’s display function would display the scaffold rotated 90 degrees. Changing the nested for loop to first go through levels then columns fixed the issue.
* (May 15th) Game’s completed function was faulty, as it did not detect completed sets if it didn’t include a piece in the first column. Fixed this bug by correcting changing upper bounds of loops to Scaffold’s columns and levels.
* (May 16th ) First design of SmartPlayer::chooseMove only focused on offensive moves and never took moves that stopped the opponent from winning. Realized that it wasn’t weighing winning moves from the opponent properly. More about this in the SmartPlayer::chooseMove section of the report.