Project 3 Report:

1. Implementing the different classes:

Class Scaffold:

The constructor resizes the vector array based on the given dimensions, unless the passed parameters are invalid

The constructor also has a stack of coordinates that stores the coordinate of the last entered checker. This helps when implementing the undoMove() function.

The numberEmpty(), checkerAt(), and display() functions work essentially in the same way. They traverse through the scaffold and count the number of VACANT spots, or return the checker at the given spot or print out the checker at the given spot with simple formatting.

The makeMove() function, which is the most used function in this class accepts a column from the caller and checks if the integer is valid( i.e. column>0 and <=s.cols()). If it is, it checks if there is space in that column to add another checker there. Once that’s done, the coordinated of this position is pushed into the stack “last”.

The undoMove() function reassigns the value VACANT to the coordinate of the last.top(), and pops it off from it.

Class Human/BadPlayer:

The choose move function for interactive player first asks for an input for a column number. If the inputted value is invalid, it prints “INVALID INPUT” and prompts the user to reenter a new value, until it is valid. Once a valid value is received, the function checks if a new checker can be added to that column. If not, the function starts the process again.

The BadPlayer just traverses through the entire scaffold, and pushes its color to the first VACANT spot it finds. The way it traverses through the scaffold assures that it will always place the checker at a valid position.

Class Game:

The complete(…) function is the longest function by far in this project. It runs four nested loops that traverse through all occupied positions (with limitations made to prevent the if condition to overflow out of the scaffold) running a third loop that runs N times. It checks if checkerAt(i,j)==checkerAt(i,j+k) where k=1, k=2,k=3…k=N. If at any point the inner loop runs fully without getting a single false condition, we break out of the loops and update winner to the value at (i,j), and return true.

We check if checkerAt(i,j)==checkerAt(i,j+k) for finding horizontal connections,

checkerAt(i,j)==checkerAt(i+k,j) for vertical connections

checkerAt(i,j)==checkerAt(i+k,j+k) for L to R upwards diagonals, and

checkerAt(i,j)==checkerAt(i-k,j+k) for L to R downward diagonals.

Pseudocode for takeTurn():

If the game is already completed

Return false;

Else

If Red’s turn:

Run the chooseMove function for the type of player Red is

Append the grid with the value chooseMove returns

Switch turn to Black

else:

Run the chooseMove function for the type of player Black is

Append the grid with the value chooseMove returns

Switch turn to Red

return true;

Pseudocode for play()

While the game is not completed

Display the grid for the user to see

takeTurn()

if a==red

print “Red wins”

else if a==black

print “black wins”

else

print ”Tie”

SmartPlayer::chooseMove

For this function I’ve tried to follow the algorithm in the spec as much as possible. I call a “determineBestComputerMove” from the body of chooseMove. As arguments, I pass the scaffold, N, a new integer ‘depth’, the color, and a stack& of ints to store the ratings.

Inside determineBestComputerMove(…) the function iterates through the columns. For each column, it checks of the move can be made, and it then makes the move. If that move completed the game, the function then checks what the result was. If it was favorable to the computer (the winner==color), it returns 1. -1 for a loss, and a 0 for a draw. In case of a victory, the vector is appended to add the column number that resulted in the victory. Having a stack here allows us to execute steps in the right order.

If the game is not completed in that step, the function calls determineBestHumanMove(…) is called. Functionally, this is identical to determineBestComputerMove. With small adjustments (like passing the opposite color as argument, adding 1 to the depth parameter etc.).

These functions call each other until the game ends. Once one iteration sees the end of the game, the appropriate column number is pushed into the stack, and the loop starts again with the next valid position. Upon testing, I found this application of the given algorithm is far from accurate. It, however, still performs better than BadPlayer.

1. The game::completed function runs 3 nested loops 4 times. This usually adds to the already large processing time of the game. To reduce pointless checking and iterating, instead of using break, I use the *goto* keyword. If I can confirm that the game is over in the first stem itself, I can skip all the looping and go directly to the return statement.
2. Test Cases:

#include "provided.h"

#include <iostream>

#include <cassert>

using namespace std;

int main()

{

HumanPlayer hp1("Bart");

BadPlayer bp1("Homer");

Scaffold s1(3, 3);

assert(s1.cols() == 3);

assert(s1.levels() == 3);

assert(s1.makeMove(1, RED));

assert(s1.checkerAt(1, 1) == RED);

assert(s1.makeMove(2, BLACK));

assert(s1.checkerAt(2, 1) == BLACK);

assert(s1.numberEmpty() == 7);

assert(s1.undoMove()==2);

assert(hp1.isInteractive());

assert(!bp1.isInteractive());

assert(hp1.name() == "Bart");

assert(bp1.name() == "Homer");

BadPlayer bp2("Moe");

Game g(3, 3, 2, &bp2, &bp1);

int a;

assert(!g.completed(a));

g.takeTurn();

assert(!g.completed(a));

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

g.takeTurn();

assert(!g.completed(a));

assert(g.checkerAt(1, 2) == BLACK || g.checkerAt(2, 2) == BLACK);

SmartPlayer sp1("einstein");

SmartPlayer sp2("pauli");

Game g2(3, 3, 2, &sp1, &sp2);

g2.takeTurn();

g2.takeTurn();

assert(!g2.completed(a));

g2.takeTurn();

assert(g2.completed(a));

}