Kyle Kodani

Project 3 Report

**DESCRIPTION:**

For the player classes, I added no private data members, but I had some helper functions for the SmartPlayerImpl class. These functions simply divided up the work to be done in the chooseMove() function.

For the scaffold class, I chose to represent the scaffold using a vector of vectors of ints. This way, the scaffold can be manipulated like a 2D array but with the ability to resize the size of each dimension. I also had two ints to represent the number of rows and columns.

For the game class, I had two pointers to players, scaffold, and an int m\_win representing the number of adjacent pieces needed to win. I also had an int m\_winner that was assigned an arbitrary value, but gets reassigned upon game completion to reflect the outcome of the game. Lastly, I had an int m\_turn that counts the turns that have passed. Its purpose was to determine whether whose turn it was (e.g. all odd turns are red, even turns are black).

In the game class I added some helper functions. I had a connectedN() function which checked for N in a row, column or diagonal, depending on the input. The checkWin() function called connectedN() 4 times, one for each possible victory direction (horizontal, vertical, diagonal down, and diagonal up). The checkWin() function is called by the completed() function. These helper functions simply exist to divide and simplify the work that must be done by the completed() function, and are only ever called within the completed() function.

**SmartPlayer::chooseMove:**

(note: this is a description of what I tried, it doesn’t fully work as intended)

I had a rating function which worked in essentially the same way that completed() from the game class works, but it returns ints that represent game states. I had a determineBestComputerMove() and a determineBestHumanMove(). These were essentially the same function, but the human function was looking for the opposite result that the computer function was looking for. The chooseMove function calls determineBestComputerMove().

dBCM() has a vector of vector of ints to store the multiple paths through the game tree, and it has a vector of ints to represent each individual path. dBCM() loops through all the empty columns and drops a piece each one (creating separate instances for ach play). For each play, it gets a rating and stores it in a path, or it calls dBHM(), and stores that result. It undoes the move it made and stores path in the collection of paths. The function then compares all the paths that have been stored and chooses either the fastest win, the longest draw, or the longest loss (I have a problem here where the comp doesn’t detect an immediate loss). The function returns the column that corresponds to the chosen path.

**PSEUDOCODE:**

*completed(int& winner)*

for each row in scaffold

for each column in scaffold

if the spot at (row, column) is RED, and red just made a move

if that spot is in a winning connection

set winner to RED and return true

else if the spot at (row, column) is BLACK, and black just made a move

if that spot is in a winning connection

set winner to BLACK and return true

if the scaffold is full

set winner to TIE\_GAME and return true

return false

*checkWin(int c, int r)*  //helper function in GameImpl

check for N adjacent pieces at (r, c) on the scaffold horizontally, vertically, diagonally up, and diagonally down

*connectedN(int c, int r, int cmod, int rmod)* //helper function in GameImpl

loop i times, where i is the number of adjacent pieces needed to win

if the color of the original piece at (r,c) does not match the color of the next piece, return false (the next piece is the next piece in one of the four possible directions and is determined by cmod and rmod)

return true

*takeTurn()*

if game is completed return false

if it’s red’s turn, have red make a move, increment number of turns, and return true

else it’s black’s turn, have black make a move, increment number of turns, and return true

*play()*

print intro messages to console

display scaffold

while the game is not completed

take a turn, display the scaffold

if both players are computers, prompt user to press enter to continue

check winner and print a message reflecting the winner

*makeMove(int c, int color)*

if parameter is out of bounds, return false

for each item in column c, starting at bottom

if item is VACANT, change it to color and return false

return false

*undoMove(int c)*

if parameter is out of bounds, return VACANT

for each item in column c, starting at top

if item is not VACANT, change it to vacant

return vacant

*chooseMove(const Scaffold& s, int N, int color)* //bad player

for each column in the scaffold, starting at left

if there is VACANT spot in that column, return column number

return -1

**BUGS, PROBLEMS, ETC:**

SmartPlayer isn’t fully functional in that it doesn’t always do what it should. It can play the game, but it can’t properly detect when a loss is imminent, but when given the opportunity to win, it will take it. I have commented out my algorithm for smartPlayer’s chooseMove() and replaced it with badPlayers. This was easily the hardest part of the project. One other problem I had was to test for win conditions. I ended up testing for four directions on each piece on the board, which seems inefficient.

**TEST CODE:**

Scaffold s(3,2);

assert(s.cols() == 3 && s.rows() == 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.checker(1, 1) == BLACK && s.checker(1, 2) == RED);

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

These tests are designed to make sure the scaffold class and all its functions work.

To test the other functions, I just played lots of games against myself or a computer.

Example:

while(true)

{

Scaffold s(7,6);

HumanPlayer hp1("bob");

HumanPlayer hp2("james");

Game g(s, 4, &hp1, &hp2);

g.play();

}