Project 3 Report

**class ScaffoldImpl**

A scaffold has a vector of vectors of ints data member m\_scaffold that stores what checker is at a location or if the location is empty. A scaffold also has a stack of ints m\_history that stores the history of moves made on the scaffold, in the form of column numbers.

**ScaffoldImpl::ScaffoldImpl(int nColumns, int nLevels)**

If nColumns or nLevels is not positive, exit program;

Resizes m\_scaffold to nColumns and nLevels and marks each location VACANT;

**int ScaffoldImpl::numberEmpty() const**

Return the number of positions minus the size of the history stack;

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

Return the value in m\_scaffold at location of column and level;

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

For each level in the scaffold starting from the top

Output a ‘|’;

For each level in the scaffold

Output the character corresponding to the value in m\_scaffold;

Output a ‘|’;

Output a newline;

Output a ‘+’;

For each column in the scaffold

Output “-+”

Output a newline;

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

If column is less than 1 or greater than m\_nColumns or color is not RED or BLACK

Return false;

For each level in the scaffold

If the column and level in m\_scaffold is VACANT

Mark location with color;

Push column onto m\_history

Return true;

Return false;

**int ScaffoldImpl::undoMove()**

If m\_history is empty, return 0;

Store m\_history’s top in an int lastCol;

Pop m\_history;

For each level in the scaffold starting from the top

If m\_scaffold at lastCol and the level is not VACANT

Change it to VACANT;

Break;

Return lastCol;

**class GameImpl**

A game has a pointer to a scaffold \*m\_scaffold, pointers to players \*m\_red, \*m\_black, \*m\_currPlayer, ints m\_n and m\_winner, which store N as in Connect N and the winner of the game.

**GameImpl::GameImpl(int nColumns, int nLevels, int N, Player\* red, Player\* black)**

Initialize all data members (\*m\_currPlayer pointers to red, m\_winner initialized with arbitrary value

Point m\_scaffold to newly allocated scaffold with nColumns and nLevels;

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

Return the value from calling function isCompleted with \*m\_scaffold, m\_n, winner;

**bool GameImpl::takeTurn()**

If the game has completed, return false;

Create int columnToMove;

Create int currColor with color of currPlayer;

If the player is interactive, prompt the player for a move;

Display the scaffold;

Store into columnToMove the return value of calling the current player’s chooseMove function;

Make the move on columnToMove with currColor;

Output that the current player has moved;

Set currPlayer to the other player;

Return true;

**void GameImpl::play()**

While takeTurn returns true, keep calling takeTurn

If both players are not interactive, pause the game after each turn and prompt the user for input to continue;

Display the scaffold;

If RED is the winner, output RED’s name has won;

Else if BLACK is the winner, output BLACK’s name has won;

Else output the game is a tie;

**int GameImpl::checkerAt(int c, int r) const**

Return the value from m\_scaffold’s checkerAt function;

**class HumanPlayerImpl**

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

If the scaffold is full, return -1;

Create an int variable to store the player’s choice of column on which to move;

Prompt player for column and store value;

While the column is out of range or full

Prompt player for valid column and store value;

Return column stored;

**class BadPlayerImpl**

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

Create an int variable to store column on which to move;

Create a copy of the scaffold;

For each of the columns of the scaffold

If a move can be made on the column with color

Set variable to the column;

Return column stored;

**class SmartPlayerImpl**

My SmartPlayerImpl::chooseMove function calls a helper function bestMove that then recursively calls itself to implement the minimax algorithm. bestMove takes in a scaffold reference, N, the current color, the computer’s color, the current recursive depth, and a vector reference. bestMove takes the place of determineBestComputerMove and determineBestHumanMove in the spec by taking in a color parameter and a computer color parameter. chooseMove creates a vector of column numbers that are ordered from the middle to the sides, which makes bestMove look for moves starting from the middle first, which gives better results. chooseMove then passes a reference of this vector to bestMove.

bestMove starts by creating a vector of Results (the Result class will be explained later) to store the evaluation of each move by rateScaffold. It then looks at every possible move on the scaffold starting from the middle column and working its way to the sides. If it’s possible to move on a column, bestMove calls rateScaffold to rate the scaffold after the move is made. If rateScaffold determines that a player has won, bestMove inserts the rating and the column on which the move was made into the Result vector. Otherwise, if the current recursive depth is less than a predefined depth limit, bestMove recursively calls itself with the other color and the depth incremented by one and inserts the result from that call into the Result vector. If the depth limit has been reached, a rating for a tie game along with the column is inserted into the vector, the move is undone, and it stops looking at further possible moves. If the limit hasn’t been reached, only the move is undone. bestMove then chooses the move with the highest rating in the Result vector to return if it is determining the best move for the computer or the move with the lowest rating if it is determining the best move for the human.

**Result SmartPlayerImpl::bestMove(Scaffold& toRate, int N, int color, int computerColor, int currDepth, const vector<int>& cols)**

Create and initialize a variable to store the other player’s color;

Create a vector of Results to store each move’s evaluation;

For each column in the cols vector

If a move can be made on the column

Create bool variable to pass into rateScaffold;

Rate the scaffold and store if a player has won into bool variable;

If won, insert rating and column into Result vector;

If not won and current depth is less than depth limit

Insert into Result vector the return rating of calling bestMove with the other player’s color and the column

If not won and current depth is within depth limit

Insert into Result vector the rating of a tie and the column;

Undo the move on the scaffold;

Break;

Undo the move on the scaffold;

If bestMove is determining the computer’s best move

Return the Result with the greatest rating;

If bestMove is determining the human’s best move

Return the Result with the least rating;

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

If the scaffold is empty, return 0;

Create a copy of the scaffold;

Create a vector of ints that orders the column numbers from middle first to sides last;

Return the column returned by Result::moveCol() called on the Result returned by calling function bestMove with the copy of the scaffold and the currDepth parameter set to 0;

I wrote a helper function SmartPlayerImpl::rateScaffold that gives a rating to a scaffold depending on whether the game has been won, whether computer or the human won, and whether the game is a tie. The rating for wins depends on the current recursive depth, which is a parameter passed into the function.

**int SmartPlayerImpl::rateScaffold(const Scaffold& s, int N, int computerColor, int currDepth, bool& won)**

Create an int variable winner;

If calling isCompleted with the scaffold, N, and winner as parameters returns that the game has been won

Set won to true;

If winner indicates a tie game, return 0;

Else if winner is the computer, return 1000 (any large number will do)- currDepth;

Else (human is the winner), return -(1000 - currDepth);

Otherwise, set won to false and return 0;

**support.h and support.cpp**

These files contain the prototype and definition of a helper function isCompleted. This function takes in a scaffold, N, and an int winner reference. It determines whether the game has been won, and if so, it sets winner to the color of the winner and returns true. Otherwise, if the game is a tie game, it sets winner to TIE\_GAME and returns true. If the game has not been won and it is not tied, the function returns false.

**bool isCompleted(const Scaffold& s, int N, int& winner)**

Initialize bool variables representing a vertical, horizontal, left diagonal. right diagonal wins to false;

For each position in the scaffold

If the position is not VACANT

If an additional N-1 elements up is within the bounds of the scaffold,

Set vertWon to true;

For each of the additional N-1 elements

If any element is not the same as the element at the position

Set vertWon to false;

If an additional N-1 elements right is within the bounds of the scaffold,

Set horzWon to true;

For each of the additional N-1 elements

If any element is not the same as the element at the position

Set horzWon to false;

If an additional N-1 elements both up and right is within the bounds of the

scaffold

Set rightDiagWon to true;

For each of the N-1 elements diagonally right and up If any element is not the same as the element at the position

Set rightDiagWon to false;

If an additional N-1 elements both up and left is within the bounds of the scaffold

Set leftDiagWon to true;

For each of the N-1 elements diagonally left and up

If any element is not the same as the element at the position

Set leftDiagWon to false;

If any of the bool variables is true

Set winner to the color of the checker at the position;

Return true;

If the scaffold is full

Set winner to TIE\_GAME;

Return true;

Return false;

**class Result**

A Result stores two ints: m\_rating and m\_moveCol. m\_rating is the return value of SmartPlayerImpl::rateScaffold. m\_moveCol is the column on which a move corresponds to the rating. This class also has constructor and accessor functions that allow a Result to be constructed and its data members to be accessed. SmartPlayerImpl::bestMove’s return value is a Result, which allows the function to essentially return two values.

**An inefficiency in my project**

An inefficiency in my program resides in the isCompleted helper function. This function looks at every position in the scaffold, and if the position is vacant and there are at least N-1 positions to the right of it, above it, diagonally up right of it, or diagonally left down of it, it checks whether these N-1 positions are the same color as the original position. If they are, then this indicates that the game has been won. A potentially faster way of the checking whether a game has been won is only checking around where the last checker has been dropped, instead of looking at every position on the game board every time the function is called. This could in some cases significantly decrease the number of positions to be evaluated. Although the speed increase for a single function call is small, this function is called a very large number of times by my SmartPlayerImpl::chooseMove function so this change could potentially speed up my program by a good amount. I did not have the time to reimplement this function so it still uses my original implementation. Although a bit slower, my original implementation still works correctly.

**Test Cases**

Scaffold s(7,6);

assert(s.cols() == 7 && s.levels() == 6); // check the number of columns and levels is correct

assert(s.numberEmpty() == 7\*6); // check that numberEmpty returns the correct number of empty positions

assert(s.undoMove() == 0); // check that it's not possible to undo a move on an empty scaffold

assert(s.makeMove(1, RED)); // check that it's possible to make a move

assert(s.makeMove(1, BLACK)); // check that it's possible to make a move

assert(s.makeMove(4, RED)); // check that it's possible to make a move

assert(s.numberEmpty() == 7\*6-3); // check that the number of empty positions has decreased appropriately

assert(s.undoMove() == 4); // check that undoMove returns the correct column of the last move

assert(s.checkerAt(4, 1) == VACANT); // check that undoMove has indeed removed the checker

assert(s.numberEmpty() == 7\*6-2); // check that the number of empty positions has increased by 1

assert(s.checkerAt(1, 2) == BLACK); // check that other checkers are unaffected by undoMove

assert(s.makeMove(1, RED)); // check that it's possible to make a move

assert(s.makeMove(1, BLACK)); // check that it's possible to make a move

assert(s.makeMove(1, RED)); // check that it's possible to make a move

assert(s.makeMove(1, BLACK)); // check that it's possible to make a move

assert(s.numberEmpty() == 6\*6); // check that an entire column is full

assert(!s.makeMove(1, RED)); // check that it's not possible to make a move on a full column

assert(s.checkerAt(1, 6) == BLACK); // check that the checker at the top level is correct

assert(s.makeMove(2, RED)); // check that it's possible to make a move

assert(s.makeMove(7, BLACK)); // check that it's possible to make a move

assert(s.undoMove() == 7); // check that it's possible to undo a move

assert(s.undoMove() == 2); // check that it's possible to undo two moves in a row

assert(s.numberEmpty() == 6\*6); // check that the number of empty positions is correct

for (int i = 0; i < 6; i++)

s.undoMove(); // undo all the moves on the scaffold

for (int i = 1; i <= s.cols(); i++)

for (int j = 1; j <= s.levels(); j++)

assert(s.checkerAt(i, j) == VACANT); // check that the scaffold is indeed empty

Scaffold test(6,5);

SmartPlayer sp("Computer"); // create a smart player

SmartPlayer sp2("Computer 2"); // create another smart player

BadPlayer bp("Homer"); // create a bad player

BadPlayer bp2("Homer 2"); // create another bad player

HumanPlayer hp("Justin"); // create a human player

assert(sp.name() == "Computer" && !sp.isInteractive()); // check that player's name is correct and isn't interactive

assert(!bp.isInteractive()); // check that player isn't interactive

assert(hp.isInteractive()); // check that human player is interactive

int badCol = bp.chooseMove(test, 4, BLACK);

assert(badCol > 0 && badCol <= 6); // check that column chosen is valid

assert(test.makeMove(badCol, BLACK)); // check that bad player can make a valid move

assert(test.checkerAt(badCol, 1) == BLACK); // check that bad player actually made a move on the scaffold

int humanCol = hp.chooseMove(test, 4, RED);

assert(humanCol > 0 && humanCol <= 6); // check that column chosen is valid

assert(test.makeMove(humanCol, RED)); // check that human player can make a valid move

if (humanCol == badCol) // check that human player actually made a move on the scaffold

assert(test.checkerAt(humanCol, 2) == RED);

else

assert(test.checkerAt(humanCol, 1) == RED);

Game g(7, 6, 4, &bp, &sp); // create a game

int winner = 2; // create a variable to pass into Game::completed

assert(!g.completed(winner) && winner == 2); // check that the game is not completed and winner is not changed

assert(g.takeTurn()); // check that it's possible to take a turn on the game

assert(g.takeTurn()); // check that it's possible to take another turn

assert(!g.completed(winner) && winner == 2); // check that the game still isn't completed

g.play(); // finish playing the game

assert(g.completed(winner)); // check that the game is completed after it's played

assert(winner == BLACK); // check that smart player has won the game

assert(!g.takeTurn()); // check that it's not possible to play the game after it's completed

Game g2(7, 6, 4, &bp, &bp2);

g2.play(); // check that it's possible for a bad player to play another bad player

assert(g.completed(winner)); // check that the game is completed after it's played

Game g3(7, 6, 4, &sp, &sp2);

g3.play(); // check that it's possible for a smart player to play another smart player

assert(g.completed(winner)); // check that the game is completed after it's played

Game g4(4, 4, 3, &hp, &sp);

g4.play(); // check that a human player can play the game

assert(g.completed(winner)); // check that the game is completed after it's played

assert(!g4.takeTurn()); // check that it's not possible to play the game after it's completed

cout << "All tests passed" << endl;