Danning Yu, 305087992

CS32 Project 3 Report

5/21/19

Overall Project Description

For this project, there were 4 classes: Board, Player, Game, and AlarmClock. For the Board class, in addition to the public member functions that were given in the specification, I added another private member function and 3 private member variables. The member variables are an integer to track the number of holes per side of the Board, a dynamically allocated array to the store the number of beans in each of North’s holes and the pot, and another dynamically allocated array to store the number of beans in each of South’s holes and pot. The private member function was a swap function, which takes in a Board and swaps it with the current Board (the *this* board). Because the Board class contains dynamically allocated variables, in addition to the constructor, a copy constructor, assignment operator, and destructor are also manually implemented, and so the swap function is used in the assignment operator implementation, which uses the copy and swap idiom. The function in this class with nontrivial implementations is Board::sow(…), and its implementation is given as pseudocode below.

The Player class is an abstract (pure virtual) class that acts as a base class for more specialized types of Players: HumanPlayer, BadPlayer, and SmartPlayer. Player contains one member variable, m\_name, which is the name of the player, as well as a corresponding getter function, Player::name(), to access the name. All the other functions in the Player class were declared virtual, and the chooseMove(…) function was made pure virtual because each derived Player class will have a different implementation. Player’s destructor, ~Player(), was also declared virtual, as Player is a base class. The derived classes do not contain any additional member variables. The derived constructors take in a string for the name of the player, and use a member initialization list called to Player’s constructor to create the base object contained within the derived object. The HumanPlayer class has its isInteractive() function return true, as the HumanPlayer object is designed to be controlled by a human, and its chooseMove(…) function works by prompting the player for a hole number. If the player enters a bad hole number, such as 0 (the pot instead) or a hole number that is out of bounds, it reprompts the player to enter a valid hole number. The BadPlayer class has its isInteractive() function return false, as it is a computer controlled player, and its chooseMove(…) simply chooses the first hole that is not empty (basically the first legal move). The SmartPlayer class is also a computer controlled class, so its isInteractive() function returns false. Then, its chooseMove(…) uses a game tree and minimax algorithm to determine the best move with the help of a private helper function, moveHelper(…), that returns the best hole. The chooseMove(…) function simply calls the moveHelper(…) function, and then returns the value that moveHelper(…) returns, which is the best hole to move from. The structure of the moveHelper(…) function is described in pseudocode below.

The Game class contains all the public functions given in the project specification, as well as the following private member variables: a Board object to store the Board that is being used in the game, two pointers to Players (which will take on the appropriate derived class type as needed), a boolean representing if it is South’s turn or not (South’s turn if true, North’s turn if false), and a private function, moveBeansToPot(). The private function simply checks if one side of the Board is empty, and if it is, it moves all the beans on the other side of the Board to that side’s pot. This function was written to make the code more organized. The Game constructor takes in a Board, as well as two pointers to Players, one for North and the other for South. It then sets southsTurn to true, as South goes first in Kalah. The nontrivial functions in this class are Game::play() and Game::move(). The Game::play() function works by using a while loop to keep the game going until it is over (which is checked with the Game::status(…) function). Within it, it calls Game::move() so that a move is made. The Game::move(…) function makes the actual move for a Player, calling Board::sow(…) on the Board object contained within the class and then giving another move if needed (due to landing in your own pot) or capturing stones as needed. Upon the end of a turn, it sets southsTurn to the opposite value so that the next time Game::move() is called, it knows to make a move for the other side.

The AlarmClock class is a class provided by Professor Smallberg to help keep the runtime of the SmartPlayer::chooseMove(…) to under 5 seconds. The class declaration and implementation is contained within the Player.h class. The AlarmClock::timeOut() function is used to determine if my SmartPlayer::chooseMove(…) function has exceeded 5 seconds, and so it is called within the function’s helper, SmartPlayer::moveHelper(…).

Description of SmartPlayer::chooseMove(…)

The given SmartPlayer::chooseMove(…) function does not container enough parameters to effectively determine the best hole for a given board, so I created a helper function, SmartPlayer::moveHelper(…), to do all the actual work and then return the best hole, which is then returned by SmartPlayer::chooseMove(…). The moveHelper(…) function operates on the basic principle of a game tree, where the computer looks at all the possible resulting Boards that go result from it going. Stated alternatively, it looks at the possible outcomes. This represents looking ahead by 1 move, and from each possible outcome, this procedure is repeated, leading to even more possible Boards, which arise from the previous possibilities. The function makes sure to alternate between giving North and South turns, as to “virtually” play a game without actually picking a hole on the actual Board. By searching a certain number of layers (that is, a certain number of layers ahead), a game tree can be formed, which basically lists all the possible outcomes of a move.

Then, taking advantage of the fact that this is a two-player, zero-sum game, which means that a resulting Board that is good for one side is bad for the other side, the minimax algorithm can be applied to the game tree that was just created, which basically finds the best possible Board outcome and then traces that Board upwards through the game tree to determine which hole to move from that will allow the Board to eventually reach the desirable state. For this project, we were told that South should try to maximize the value of its Board, which means that North should minimize it.

To assign a value to each board, I used the following heuristic: value is equal to the number of South’s beans in its pot minus the number of beans in North’s pot. What this heuristic does is favor building a lead over the opposing side in the number of beans one has in their pot, and in the long term, it seems to work best for winning Kalah. I also tried using the difference between the sum of the current side’s beans in play and in the pot and the sum of the opposing side’s beans in play and in the pot, but only considering the difference in beans between the pots was found to work better after testing. Using this heuristic to assign a value to each board, the moveHelper(…) algorithm then propagates the values of the Boards upwards, either picking the most positive value (if moveHelper(…) is picking for South) or the most negative value (if moveHelper(…) is picking for North), and when the current Board in play is reached, it then returns the hole that if picked, can (but is not guaranteed to, due to the other side making countermoves) eventually lead to the most optimal board found for the pertinent side. The “propagating” of the values is done using recursion, as specified in the pseudocode given in the project specification.

The moveHelper function takes in 7 parameters: a Board, a Side, the original Side, the value of the best hole, the value of the board, and current layer it’s on, and an AlarmClock. The board, best hole, original side, value, and Alarm Clock are passed by reference so that they are values shared within all copies of the moveHelper function, while the current Side and layer variables are passed by value. The original Side parameter is passed by constant reference to prevent it from changing, as it stores the side corresponding to the side passed in to chooseMove(…), which is the side that moveHelper(…) is making a move for. First, moveHelper checks if the game is over or if the layer search limit is reached; if the game is over, it assigns either a very positive, very negative, or zero value to the board depending on if South won, North won, or if there was a tie. If the layer search limit is reached or the alarm clock has timed out, it simply returns without performing any search. Then, it makes all the possible moves given the board and which side it is making moves from, performing captures and going again if needed. The go again and capture code was taken directly from Game::move(). Then, it recursively calls moveHelper(…) for the opposing side, incrementing layer by 1 to show that it is now looking one more move ahead and changing the current side to the opponent so that it knows the other side is going. Then, after the recursive call and all possible moves have been examined, it compares the current best seen value to the newly calculated value from recursively calling the moveHelper function, and if the value is more favorable, it changes the best hole to the hole that led to the better Board being created and updates the value for any future comparisons. There are also checks in various places throughout the function to see if the alarm clock has timed out, and if it has, it immediately returns (exits the function) without doing anything. Pseudocode for the chooseMove(…) and moveHelper(…) functions are given below:

chooseMove(inputs: board, side to choose for; outputs: best hole to choose):

Start the AlarmClock, set to 4900 ms to allow for 100 ms of function wrap-up time

Call the moveHelper(…) function

Make a safety: if moveHelper returns a bad hole value:

Return the first hole that contains a nonzero number of beans

Return the best hole value given by moveHelper

moveHelper(inputs: board, side that’s going, the original side, best hole, value of board, current

layer, and alarm clock, output: best hole, value)

if the AlarmClock timed out:

stop running the function and return

if one side of the board is empty (that is, game is over):

move all the beans on the other side to the other side’s pot (clear the board)

assign the correct value for a winning/losing board, 0 if tied

stop running the function and return

if the layer search limit is reached:

determine the value of the current board

stop running the function and return

for each possible move:

make a copy of the board to use for the next move

sow the board (that is, make the move)

if your last bean ends up in your own pot:

recursively call moveHelper, with the same side and same layer

(same side because you’re going again)

(same layer because the move isn’t complete yet)

if a capture occurs:

perform the capture

recursively call moveHelper, passing the opposing side and layer + 1

(opposing side because it’s now the opponent’s turn)

(layer + 1 because you’re now using moveHelper on the next turn)

if neither landing in your pot nor a capture happens:

recursively call moveHelper, passing the opposing side and layer + 1

(opposing side because it’s now the opponent’s turn)

(layer + 1 because you’re now using moveHelper on the next turn)

if the value of the board due to a move made above is better than the current best value:

change the best hole to the hole that resulted in the better board

update the value with the new value for future comparisons

return because the end of the function has been reached

Pseudocode for nontrivial algorithms:

Board::sow(inputs: side to sow for, hole to sow from; outputs: if successfully sowed, ending

side, ending hole):

if the hole to sow from is invalid:

return false

determine the number of beans to move around

(that is, determine the number of beans in the hole to sow from)

if there are no beans to sow, return false

set the number of beans in the hole to sow from to 0 (“picking up” the beans)

add beans to each following hole until your pot is reached and decrement the number of

beans to move around accordingly

(the pot is now the “zero” point to sow the rest of the beans from)

Determine how many beans are needed to make a full circle around the board

Determine how many full laps can be made and how many beans will be left afterwards

To every hole on the board, plus your own pot, add a number of beans equal to how many

full laps can be made

Starting from your own hole, add beans to the following holes until the number of beans

left afterwards is 0

Record which side and hole the last bean was placed in

(basically, separate into how many beans can be given to all the holes plus your own pot,

and then sow the rest, which will be guaranteed to not loop around the board)

return true because sowing was successfully conducted

Board’s constructor (inputs: number of holes, beans per hole):

If either parameter passed in is 0 or negative, change them to the correct values

Initialize Board’s private number of holes to the number of holes passed in

Dynamically allocate two arrays, each with size of number of holes + 1

(the +1 is for the pot)

Fill the arrays, except for the 0th element, with the number of beans per hole

Set the 0th element of the arrays to 0 beans (pots start out with 0 beans)

Board’s copy constructor (input: a board, output: a new board is made):

Copy over the number of holes per side

Dynamically allocate two arrays, each with size of number of holes + 1

Copy each value inside the arrays from the old board to the new board

Board’s assignment operator:

If the left value and right value are not the same:

Make a temporary board of the right value using the copy constructor

Swap the left value with the temporary board

Board’s swap function (input: two boards; output: two swapped boards):

Swap the sizes of the boards, using a temporary variable

Swap the pointers for the boards’ two dynamically allocated arrays, using a temporary

pointer

Player::chooseMove(…):

The pseudocode for this function has already been given above.

Game::status(input: board; outputs: is game over, is there a winner, who’s the winning side):

if both side still have beans in play, the game is not over

return

(now we’re guaranteed the game is over)

set the game over to true

if the number of beans in South’s pot is equal to the number of beans in North’s pot:

there is not a winner

return

For the side who has more beans in their pot

there is a winner

winning side is the side with more beans in their pot

return

Game::move(inputs: board, side to make a move for; outputs: if a move can be made, and it

modifies the board):

if either side only has no beans in play:

move all the beans on the other side to that side’s pot

return false because the game is now over

if both side have no beans in play, return false because a move cannot be made

for the side that’s currently going, make a move for it:

If the player it’s making a move for is interactive, ask human to input hole #

Otherwise, have the player pick a move (computer player)

For as long as you land in your own pot:

Make a copy of the board

Sow from the hole that was picked by the player

If a capture occurs, end the turn immediately (break out of loop)

Change the side that’s currently going to the other side

check if one side of the board is empty, if so:

move all the beans on the other side to that side’s pot

return true because it successfully made a move

Game::play(input: a game; output: a finished game):

For as long as the game is not over:

Make a move

Check if the game is over

Once the game is over:

Determine who won or if a tie occurred using Game::status(…)

Return

Game::moveBeansToPot(inputs: board; ouputs: if any moving occurred):

If one side of the board is empty:

Loop through all the holes of the other side:

Move the beans in each hole to that side’s pot

Return true;

Return false because there isn’t a side of the board that’s empty

Game::display(input: board and players; output: a display of the game):

Print the north side’s name

Print the values of north’s holes

Print the values of north’s pot and south’s pot, adding spaces between the two values to make them spaced appropriately to make it look like a Kalah board

Print the values of south’s holes

Print south’s name

Return

Bugs, Inefficiencies, and Problems:

As far as I can tell, there do not seem to be any serious bugs with the code. However, the Board::sow(…) and Game::move(…) functions seem to be very inefficient, as I coded separate cases for if the Side that is being sowed or moved is South or North. For example, for move(…), if South is moving, I hard coded it to move any captured beans to South’s pot, and correspondingly for North’s pot, and I feel that I could have brought it together into one case, where you moved captured beans into the current Side’s pot. The same logic is used for both cases, so there are only small differences, and I think that it is possible to simply bring both of those cases together into one. However, I could not figure out how to do this for some of the other similar actions, so thus, my code for those functions is divided into South and North cases. For the SmartPlayer::moveHelper(…) function, which basically does all the work for SmartPlayer::chooseMove(…), I passed by reference and constant reference whenever possible to optimize the run speed of the function.

I observed an interesting phenomenon where if I search too many layers (that is, if I search too deep), the SmartPlayer actually makes moves are less optimal than if it searches fewer layers. It seemed like the optimal number of layers was 5: less than that and it wouldn’t search deep enough, and more than that, it would do worse than if it only search 5 layers. I theorize this is because the algorithm I implemented for SmartPlayer is a depth first search, so when I search many layers, it hits the time limit, which means it has search to the bottom of a single branch of the game tree, while the preferable approach is to a breadth first search, determining the value of all the boards for a particular layer before moving on to the next layer. Thus, changing from a depth-first search to a breadth-first search would be an area of possible improvement for the future.

Test cases:

My test cases are given below as a series of asserts that I placed in my Main.cpp file:

{//Test Board.cpp

//testing if you pass in bad parameters to the constructor

Board bad1(-1, -1); //first is holes, second is beans per hole

Board bad2(0, -1);

Board bad3(-1, 0);

Board bad4(0, 0);

assert(bad1.beans(Side::NORTH, 0) == 0);

assert(bad1.beans(Side::SOUTH, 0) == 0);

assert(bad2.beans(Side::NORTH, 0) == 0);

assert(bad2.beans(Side::SOUTH, 0) == 0);

assert(bad3.beans(Side::NORTH, 0) == 0);

assert(bad3.beans(Side::SOUTH, 0) == 0);

assert(bad4.beans(Side::NORTH, 0) == 0);

assert(bad4.beans(Side::SOUTH, 0) == 0);

assert(bad1.holes() == 1);

assert(bad1.beansInPlay(Side::SOUTH) == 0);

assert(bad1.beansInPlay(Side::NORTH) == 0);

assert(bad1.totalBeans() == 0);

assert(bad2.holes() == 1);

assert(bad2.beansInPlay(Side::SOUTH) == 0);

assert(bad2.beansInPlay(Side::NORTH) == 0);

assert(bad2.totalBeans() == 0);

assert(bad3.holes() == 1);

assert(bad3.beansInPlay(Side::SOUTH) == 0);

assert(bad3.beansInPlay(Side::NORTH) == 0);

assert(bad3.totalBeans() == 0);

assert(bad3.holes() == 1);

assert(bad3.beansInPlay(Side::SOUTH) == 0);

assert(bad3.beansInPlay(Side::NORTH) == 0);

assert(bad3.totalBeans() == 0);

//now test everything else for Board

Board b1(3, 2); //3 holes, 2 beans per pot

// NORTH

// 2 2 2

//0 0

// 2 2 2

// SOUTH

assert(b1.beans(Side::NORTH, 0) == 0); //pots should start with 0 beans

assert(b1.beans(Side::SOUTH, 0) == 0);

assert(b1.beansInPlay(Side::NORTH) == 6); //3 holes \* 2 beans per hole = 6 beans

assert(b1.beansInPlay(Side::SOUTH) == 6);

assert(b1.totalBeans() == 12); //6 beans per side \* 2 sides = 12 beans

assert(b1.holes() == 3); //number of holes should be set to 3

assert(b1.beans(Side::SOUTH, -1) == -1); //-1 is invalid hole #

assert(b1.beans(Side::SOUTH, 4) == -1); //4 is out of bounds

assert(b1.beans(Side::SOUTH, 1) == 2); //should be 2 beans per hole

assert(b1.beans(Side::NORTH, -1) == -1); //repeat above tests for north side

assert(b1.beans(Side::NORTH, 4) == -1); //4 is out of bounds

assert(b1.beans(Side::NORTH, 1) == 2); //should be 2 beans per hole

assert(b1.moveToPot(Side::SOUTH, 0, Side::SOUTH) == false); //cannot move from pot to pot

assert(b1.moveToPot(Side::SOUTH, 4, Side::SOUTH) == false); //cannot move from out of bounds hole

assert(b1.moveToPot(Side::SOUTH, 0, Side::NORTH) == false); //repeat above tests with other side

assert(b1.moveToPot(Side::SOUTH, 4, Side::NORTH) == false);

assert(b1.moveToPot(Side::SOUTH, 3, Side::SOUTH) == true); //should work, 3 is within bounds

// NORTH

// 2 2 2

//0 2

// 2 2 0

// SOUTH

assert(b1.beansInPlay(Side::SOUTH) == 4); //beans in play should be decreased

assert(b1.beansInPlay(Side::NORTH) == 6); //beans in play for other side should be unchanged

assert(b1.beans(Side::SOUTH, 3) == 0); //all the beans should have been moved

assert(b1.beans(Side::NORTH, 3) == 2); //beans on other side shouldn't be affected

assert(b1.totalBeans() == 12); //total # of beans should be unchanged

assert(b1.setBeans(Side::NORTH, -1, 3) == false); //-1 is out of bounds

assert(b1.setBeans(Side::NORTH, 1, -3) == false); //cannot set beans to be negative

assert(b1.setBeans(Side::NORTH, -1, -1) == false); //combination of two test cases above

assert(b1.setBeans(Side::NORTH, 0, 5) == true); //0 is the pot, beans is positive, should succeed

// NORTH

// 2 2 2

//5 2

// 2 2 0

// SOUTH

assert(b1.totalBeans() == 17); //total # of beans should have changed

assert(b1.beans(Side::NORTH, 0) == 5); //beans in north's pot should now be 5

assert(b1.beans(Side::SOUTH, 0) == 2); //beans in south's pot should be changed

assert(b1.setBeans(Side::NORTH, 3, 1)); //change up some values of the board

assert(b1.setBeans(Side::SOUTH, 1, 4));

assert(b1.setBeans(Side::SOUTH, 3, 1));

// NORTH

// 2 2 1

//5 2

// 4 2 1

// SOUTH

assert(b1.totalBeans() == 19); //test if total beans is updated

assert(b1.beansInPlay(Side::NORTH) == 5); //check the totals

assert(b1.beansInPlay(Side::SOUTH) == 7); //check the totals

assert(b1.beans(Side::SOUTH, 0) == 2); //check all the values of the board

assert(b1.beans(Side::SOUTH, 1) == 4);

assert(b1.beans(Side::SOUTH, 2) == 2);

assert(b1.beans(Side::SOUTH, 3) == 1);

assert(b1.beans(Side::NORTH, 0) == 5);

assert(b1.beans(Side::NORTH, 1) == 2);

assert(b1.beans(Side::NORTH, 2) == 2);

assert(b1.beans(Side::NORTH, 3) == 1);

Board b2(b1); //test copy constructor

assert(b2.totalBeans() == 19); //test if total beans is copied correctly

assert(b2.beansInPlay(Side::NORTH) == 5); //check the totals

assert(b2.beansInPlay(Side::SOUTH) == 7); //check the totals

assert(b2.beans(Side::SOUTH, 0) == 2); //check all the values of the board

assert(b2.beans(Side::SOUTH, 1) == 4);

assert(b2.beans(Side::SOUTH, 2) == 2);

assert(b2.beans(Side::SOUTH, 3) == 1);

assert(b2.beans(Side::NORTH, 0) == 5);

assert(b2.beans(Side::NORTH, 1) == 2);

assert(b2.beans(Side::NORTH, 2) == 2);

assert(b2.beans(Side::NORTH, 3) == 1);

assert(b2.holes() == 3); //check # of holes were set correctly

assert(b2.moveToPot(Side::SOUTH, 2, Side::SOUTH) == true); //move some beans

assert(b2.moveToPot(Side::NORTH, 1, Side::SOUTH) == true);

// NORTH (Board b2)

// 0 2 1

//5 6

// 4 0 1

// SOUTH

assert(b2.beans(Side::SOUTH, 0) == 6); //pot beans should be updated

assert(b2.beans(Side::NORTH, 1) == 0); //# of beans in hole should be updated

assert(b2.beans(Side::SOUTH, 2) == 0);

assert(b1.beans(Side::SOUTH, 0) == 2); //pot for b1 should be unchanged

assert(b1.beans(Side::NORTH, 1) == 2); //board for b1 should be unchanged

assert(b1.beans(Side::SOUTH, 2) == 2);

b2 = b2; //test assignment operator, this should do nothing

assert(b2.beans(Side::SOUTH, 0) == 6); //check all the values of the board, should be unchanged

assert(b2.beans(Side::SOUTH, 1) == 4);

assert(b2.beans(Side::SOUTH, 2) == 0);

assert(b2.beans(Side::SOUTH, 3) == 1);

assert(b2.beans(Side::NORTH, 0) == 5);

assert(b2.beans(Side::NORTH, 1) == 0);

assert(b2.beans(Side::NORTH, 2) == 2);

assert(b2.beans(Side::NORTH, 3) == 1);

Board b3(6, 6); //make some other board to test assignment operator

assert(b3.totalBeans() == 72);

assert(b3.holes() == 6);

assert(b3.beansInPlay(Side::SOUTH) == 36);

b3 = b1;

assert(b3.totalBeans() == 19); //test if total beans is updated

assert(b3.holes() == 3); //test if # of holes is updated

assert(b3.beansInPlay(Side::NORTH) == 5); //check the totals

assert(b3.beansInPlay(Side::SOUTH) == 7); //check the totals

assert(b3.beans(Side::SOUTH, 0) == 2); //check all the values of the board

assert(b3.beans(Side::SOUTH, 1) == 4);

assert(b3.beans(Side::SOUTH, 2) == 2);

assert(b3.beans(Side::SOUTH, 3) == 1);

assert(b3.beans(Side::NORTH, 0) == 5);

assert(b3.beans(Side::NORTH, 1) == 2);

assert(b3.beans(Side::NORTH, 2) == 2);

assert(b3.beans(Side::NORTH, 3) == 1);

assert(b3.moveToPot(Side::SOUTH, 2, Side::SOUTH) == true); //move some beans

assert(b3.moveToPot(Side::NORTH, 1, Side::SOUTH) == true);

// NORTH (Board b3)

// 0 2 1

//5 6

// 4 0 1

// SOUTH

// NORTH (Board b2)

// 0 2 1

//5 6

// 4 0 1

// SOUTH

// NORTH (board b1)

// 2 2 1

//5 2

// 4 2 1

// SOUTH

assert(b1.beans(Side::SOUTH, 0) == 2); //pot for b1 should be unchanged

assert(b1.beans(Side::NORTH, 1) == 2); //board for b1 should be unchanged

assert(b1.beans(Side::SOUTH, 2) == 2);

assert(b2.beans(Side::SOUTH, 0) == 6); //check all the values of the board are updated

assert(b2.beans(Side::SOUTH, 1) == 4);

assert(b2.beans(Side::SOUTH, 2) == 0);

assert(b2.beans(Side::SOUTH, 3) == 1);

assert(b2.beans(Side::NORTH, 0) == 5);

assert(b2.beans(Side::NORTH, 1) == 0);

assert(b2.beans(Side::NORTH, 2) == 2);

assert(b2.beans(Side::NORTH, 3) == 1);

Board temp(b1); //make a copy of b1

int holeResult = -99;

Side endSide = Side::SOUTH;

assert(b1.sow(Side::SOUTH, -1, endSide, holeResult) == false);

assert(b1.sow(Side::SOUTH, 0, endSide, holeResult) == false);

assert(holeResult == -99 and endSide == Side::SOUTH); //values of holeResult and endSide should be uncahnged

assert(b1.sow(Side::SOUTH, 1, endSide, holeResult) == true);

// NORTH (board b1)

// 2 2 (2) //parentheses indicates ending hole

//5 3

// 0 3 2

// SOUTH

assert(endSide == Side::NORTH && holeResult == 3);

assert(b1.beans(Side::SOUTH, 0) == 3); //check all the values of the board are updated

assert(b1.beans(Side::SOUTH, 1) == 0);

assert(b1.beans(Side::SOUTH, 2) == 3);

assert(b1.beans(Side::SOUTH, 3) == 2);

assert(b1.beans(Side::NORTH, 0) == 5);

assert(b1.beans(Side::NORTH, 1) == 2);

assert(b1.beans(Side::NORTH, 2) == 2);

assert(b1.beans(Side::NORTH, 3) == 2);

b1.setBeans(Side::NORTH, 1, 20);

assert(b1.beans(Side::NORTH, 1) == 20); //check beans were set correctly

// NORTH (board b1)

// 20 2 2

//5 3

// 0 3 2

// SOUTH

b1.sow(Side::NORTH, 1, endSide, holeResult); //test sow for a large number

cout << "Ending side is " << endSide << " holeResult is " << holeResult << endl;

// NORTH (board b1)

// 2 5 5 //parentheses indicates ending hole

//8 3

// 3 6 5

// SOUTH

assert(endSide == Side::NORTH && holeResult == 2);

assert(b1.beans(Side::SOUTH, 0) == 3); //check all the values of the board are updated

assert(b1.beans(Side::SOUTH, 1) == 3);

assert(b1.beans(Side::SOUTH, 2) == 6);

assert(b1.beans(Side::SOUTH, 3) == 5);

assert(b1.beans(Side::NORTH, 0) == 8);

assert(b1.beans(Side::NORTH, 1) == 2);

assert(b1.beans(Side::NORTH, 2) == 5);

assert(b1.beans(Side::NORTH, 3) == 5);

b1 = temp; //revert b1 back to starting state

holeResult = -99; //reset holeResult and endSide

endSide = Side::SOUTH;

// NORTH (board b1)

// 2 2 1

//5 2

// 4 2 1

// SOUTH

assert(b1.sow(Side::NORTH, -1, endSide, holeResult) == false); //now repeat all tests for other side

assert(b1.sow(Side::NORTH, 0, endSide, holeResult) == false);

assert(holeResult == -99 and endSide == Side::SOUTH); //values of holeResult and endSide should be uncahnged

assert(b1.sow(Side::NORTH, 1, endSide, holeResult) == true);

// NORTH (board b1)

// 0 2 1

//6 2

// (5) 2 1

// SOUTH

assert(endSide == Side::SOUTH && holeResult == 1);

assert(b1.beans(Side::SOUTH, 0) == 2); //check all the values of the board are updated

assert(b1.beans(Side::SOUTH, 1) == 5);

assert(b1.beans(Side::SOUTH, 2) == 2);

assert(b1.beans(Side::SOUTH, 3) == 1);

assert(b1.beans(Side::NORTH, 0) == 6);

assert(b1.beans(Side::NORTH, 1) == 0);

assert(b1.beans(Side::NORTH, 2) == 2);

assert(b1.beans(Side::NORTH, 3) == 1);

b1.setBeans(Side::SOUTH, 1, 20);

assert(b1.beans(Side::SOUTH, 1) == 20); //check beans were set correctly

// NORTH (board b1)

// 0 2 1

//6 2

// 20 2 1

// SOUTH

b1.sow(Side::SOUTH, 1, endSide, holeResult); //test sow for a large number

cout << "Ending side is " << endSide << " holeResult is " << holeResult << endl;

// NORTH (board b1)

// (3) 5 4 //parentheses indicates ending hole

//6 5

// 2 5 4

// SOUTH

assert(endSide == Side::NORTH && holeResult == 1);

assert(b1.beans(Side::SOUTH, 0) == 5); //check all the values of the board are updated

assert(b1.beans(Side::SOUTH, 1) == 2);

assert(b1.beans(Side::SOUTH, 2) == 5);

assert(b1.beans(Side::SOUTH, 3) == 4);

assert(b1.beans(Side::NORTH, 0) == 6);

assert(b1.beans(Side::NORTH, 1) == 3);

assert(b1.beans(Side::NORTH, 2) == 5);

assert(b1.beans(Side::NORTH, 3) == 4);

//test Player.cpp

SmartPlayer sp("Test 1");

HumanPlayer hp("Test 2");

BadPlayer bp1("Test 3");

BadPlayer bp2("Test 4");

assert(sp.name() == "Test 1");

assert(hp.name() == "Test 2");

assert(bp1.name() == "Test 3");

assert(bp2.name() == "Test 4");

assert(sp.isInteractive() == false);

assert(hp.isInteractive() == true);

assert(bp1.isInteractive() == false);

assert(bp2.isInteractive() == false);

//test Game.cpp

Game g(b1, &bp1, &bp2);

// NORTH (board b1)

// 3 5 4

//6 5

// 2 5 4

// SOUTH

g.display();

bool gameOver = true;

bool hasWinner = true;

Side winningSide = Side::NORTH;

g.status(gameOver, hasWinner, winningSide);

assert(gameOver == false); //game is not over

assert(hasWinner == true); //hasWinner should be unchanged

assert(winningSide == Side::NORTH); //side should be unchanged

g.move(); //south should move from hole 1

// NORTH (board b1)

// 3 5 4

//6 5

// 0 6 5

// SOUTH

assert(g.beans(Side::SOUTH, 0) == 5); //check all the values of the board are updated

assert(g.beans(Side::SOUTH, 1) == 0);

assert(g.beans(Side::SOUTH, 2) == 6);

assert(g.beans(Side::SOUTH, 3) == 5);

assert(g.beans(Side::NORTH, 0) == 6);

assert(g.beans(Side::NORTH, 1) == 3);

assert(g.beans(Side::NORTH, 2) == 5);

assert(g.beans(Side::NORTH, 3) == 4);

g.status(gameOver, hasWinner, winningSide);

assert(gameOver == false); //game is not over

assert(hasWinner == true); //hasWinner should be unchanged

assert(winningSide == Side::NORTH); //side should be unchanged

g.move(); //north goes

// NORTH (board b1)

// 0 5 4

//7 5

// 1 7 5

// SOUTH

assert(g.beans(Side::SOUTH, 0) == 5); //check all the values of the board are updated

assert(g.beans(Side::SOUTH, 1) == 1);

assert(g.beans(Side::SOUTH, 2) == 7);

assert(g.beans(Side::SOUTH, 3) == 5);

assert(g.beans(Side::NORTH, 0) == 7);

assert(g.beans(Side::NORTH, 1) == 0);

g.status(gameOver, hasWinner, winningSide);

assert(gameOver == false); //game is not over

assert(hasWinner == true); //hasWinner should be unchanged

assert(winningSide == Side::NORTH); //side should be unchanged

g.move(); //south goes

// NORTH (board b1)

// 0 5 4

//7 5

// 0 8 5

// SOUTH

g.move(); //north goes

// NORTH (board b1)

// 1 0 4

//8 5

// 1 9 6

// SOUTH

assert(g.beans(Side::SOUTH, 0) == 5); //check all the values of the board are updated

assert(g.beans(Side::SOUTH, 1) == 1);

assert(g.beans(Side::SOUTH, 2) == 9);

assert(g.beans(Side::SOUTH, 3) == 6);

assert(g.beans(Side::NORTH, 0) == 8);

assert(g.beans(Side::NORTH, 1) == 1);

assert(g.beans(Side::NORTH, 2) == 0);

assert(g.beans(Side::NORTH, 3) == 4);

g.move(); //south goes

// NORTH (board b1)

// 1 0 4

//8 5

// 0 10 6

g.move(); //north goes

// NORTH (board b1)

// 0 0 4

//9 5

// 1 10 6

// SOUTH

//NORTH GOES AGAIN

// NORTH (board b1)

// 1 1 0

//10 5

// 1 10 6

// SOUTH

assert(g.beans(Side::SOUTH, 0) == 5); //check all the values of the board are updated

assert(g.beans(Side::SOUTH, 1) == 1);

assert(g.beans(Side::SOUTH, 2) == 10);

assert(g.beans(Side::SOUTH, 3) == 6);

assert(g.beans(Side::NORTH, 0) == 10);

assert(g.beans(Side::NORTH, 1) == 1);

assert(g.beans(Side::NORTH, 2) == 1);

assert(g.beans(Side::NORTH, 3) == 0);

for (int i = 0; i < 14; i++) { g.move(); }

//move 14 times, will be 1 move away from game over

g.status(gameOver, hasWinner, winningSide);

assert(gameOver == false); //game is not over

assert(hasWinner == true); //hasWinner should be unchanged

assert(winningSide == Side::NORTH); //side should be unchanged

g.move(); //last move, south goes

assert(g.move() == false); //game is now over, cannot do any more moves

g.status(gameOver, hasWinner, winningSide);

assert(gameOver == true); //game is not over

assert(hasWinner == true); //hasWinner should be unchanged

assert(winningSide == Side::NORTH); //side should be unchanged

g.display();

Board b4(2, 2); //2 holes

b4.setBeans(Side::NORTH, 0, 5);

b4.setBeans(Side::SOUTH, 0, 5);

b4.setBeans(Side::SOUTH, 1, 0);

b4.setBeans(Side::SOUTH, 2, 0);

b4.setBeans(Side::NORTH, 1, 0);

b4.setBeans(Side::NORTH, 2, 0);

// NORTH (board b4)

// 0 0

//5 5

// 0 0

// SOUTH

Game g2(b4, &bp1, &bp2);

assert(g2.move() == false); //game is already over

gameOver = false;

hasWinner = true;

winningSide = Side::NORTH;

g2.status(gameOver, hasWinner, winningSide);

assert(gameOver == true && hasWinner == false); //game is over but no winner (tie)

assert(winningSide == Side::NORTH); //winning side should be unchanged

Board b5 = b4;

b5.setBeans(Side::SOUTH, 0, 6); //make south win now

Game g3(b5, &bp1, &bp2);

g3.display();

// NORTH (board b4)

// 0 0

//5 6

// 0 0

// SOUTH

gameOver = false;

hasWinner = true;

winningSide = Side::NORTH;

g3.status(gameOver, hasWinner, winningSide);

assert(gameOver == true && hasWinner == true); //game is over, winner exists

assert(winningSide == Side::SOUTH); //winning side should be changed

//now test capturing

// NORTH (Board b2)

// 0 2 1

//5 6

// 4 0 1

// SOUTH

b2.setBeans(Side::SOUTH, 1, 1); //add beans to a hole

b2.setBeans(Side::SOUTH, 2, 0); //clear out a hole for captuing

b2.setBeans(Side::SOUTH, 3, 2); //add beans to a hole

b2.setBeans(Side::NORTH, 1, 0);

b2.setBeans(Side::NORTH, 2, 4); //make opposing hole nonempty

b2.setBeans(Side::NORTH, 3, 1);

// NORTH (Board b2)

// 0 4 1

//5 6

// 1 0 2

// SOUTH

Game g4(b2, &bp1, &bp2);

g4.move();

// NORTH (Board b2)

// 0 0 1

//5 11

// 0 0 2

// SOUTH

//capture occurs, beans in north's hole 2 are cpatured

assert(g4.beans(Side::SOUTH, 0) == 11); //check all the values of the board are updated

assert(g4.beans(Side::SOUTH, 1) == 0);

assert(g4.beans(Side::SOUTH, 2) == 0);

assert(g4.beans(Side::SOUTH, 3) == 2);

assert(g4.beans(Side::NORTH, 0) == 5);

assert(g4.beans(Side::NORTH, 1) == 0);

assert(g4.beans(Side::NORTH, 2) == 0);

assert(g4.beans(Side::NORTH, 3) == 1);

cerr << "All tests passed! Yay!" << endl;

}