Ryan Tran

905099275

Project 3 Writeup

Description and Design of classes

Fundamentally, for my board class, the major data structure that I used was vectors, in which I utilized two different vectors with indices equal to the number of holes that each side of the board had plus one. Utilizing the pushback function, my constructor of the board class pushed in the number “nHole” holes of bean value “nInitialBeansPerHole”, and pushed in a final value initialized at 0 to represent the pots in the very end. The private member variable names, consequently were named vector<int> north and vector<int> south, denoting the containers holding the values of holes and specific pot for both the north and the south sides of the boards. Furthermore, I additionally created the private member variable int m\_holes which denotes the number of holes on each side of the board. The purpose of the m\_holes variable was mainly for the holes() function that returned the value m\_holes, which I used extensively throughout my project to iterate through all the holes on a specific side. I also used the holes() function (derived from the value m\_holes) in many of my other functions, for invalid case checks ascertained by conditional statements, and in many cases, to obtain the index of a specific value in the vector. Thus, the m\_holes private member variable and coinciding holes() member function were indispensable for me as I wrote the board class and it’s coinciding functions.

For the player class, the only private member function that I added was the minimax function, which I prescribed to SmartPlayer, and used as a helper function for chooseMove. The reason why minimax was necessary as a helper function, was because of the fact that it needed to output two components, the best hole that would yield the best results, and the value that coincides with that best hole. In creating parameters that were pass by reference. I was able to achieve this affect, yielding the integer value of the best hole that would be passed on to SmartPlayer’s chooseMove(), while modifying the value component for recursion comparison purposes that would allow me to figure out what the best hole was. Thus, SmartPlayer’s private member function minimax was fundamentally important to yield the best hole and do the work necessary in obtaining this data.

For my game class, I defined three private member variables: m\_board which was of type Board class, m\_south which was a pointer to the south player, m\_north which was a pointer to the north player, and m\_turn which was of type Side and kept track of the Side the game was currently on. Firstly, the creation of the m\_board private variable was indispensable in the creation of the board class, because game is a class that brings the components of players and the board to create a cohesive and workable game. m\_board, m\_south, and m\_north proved to be indispensable components of the game class, because the variable m\_board were the basis by which functions like status() were tracking and move() were making game moves on. m\_south and m\_north were just as pivotal for the game class, because they created pointers to two different players, to which we assigned to either the south side or north side, and differentiated these players, allowing us to integrate and track players of differing derived classes, and calling their coinciding functions accordingly. The significance of m\_turn resides in the fact that we could keep track of which turn the game was at, allowing us to keep track of which player we should be accessing with out functions and which side of the board we should be modifying.

a description of your design for SmartPlayer::chooseMove, including what heuristics you used to evaluate board positions.

Description of Design for SmartPlayer::chooseMove

My SmartPlayer::chooseMove function was derived fundamentally by my helper function minimax. Minimax recurses through the possible choices of a game, either stopping when it finds an outcome in which one of the players wins, when the inputted depth of search is reached, or when an allotted time frame of 5 seconds has been exceeded. In order to achieve this purpose, minimax takes in the variable height (which coincides with depth), a board passed by constant reference, the side that we are currently evaluating, a variable besthole that is passed by reference and modified throughout the function, the variable value that is passed by reference and modified throughout the function, and an alarm clock that keeps track of when our five seconds has expired. Fundamentally, my minimax function is a recursive function which evaluates the difference between two pots and subtracts it from the value of the opponent on the next recursive step. Upon a favorable comparison the best hole and value is modified to represent the coinciding situation that is most favorable. The first primary heuristic I used manifests itself in the base case and the recursive step. Upon reaching the end of the depth, or upon the expiration of our clock the value would be adjusted to be the number of beans in the pot of the current side minus the pot of the other side. The reason why this heuristic is more useful than, for instance, evaluating how many beans is in one side, is because it takes into account how much more ahead or behind the player is compared to their opponent, and picks the best situation according to this facet. The other heuristic I used was if the player won, value would be set to 999 and if the player lost, value would be set to -999. This is an important heuristic in the base case, because it denotes the end of our tree, and because it assigns a variable of comparison that will yield our recursion either a direct gravitation towards or away from. To extend on the validity of my SmartPlayer::chooseMove function, I also incorporated the two rules of kalah that gives players a second turn if they land on their pot, and captures the opponents beans if the player lands on an empty hole in which their opponent’s coinciding hole is nonempty. I incorporated the first rule by utilizing conditional statements that ascertained whether the player landed on their hole, and if so, recursed and inputted the same player instead of switching to the opponent. I incorporated the capture rule by employing conditional statements that checked for the conditions that would necessitate a capture, and utilizing board’s moveToPot function in the minimax function itself, in order to replicate the effects of capture. This lends to the overall efficacy in figuring out the best hole so that minimax will yield the best results for a player.

Pseudocode

bool Board::sow(Side s, int hole, Side& endSide, int& endHole)

Check for invalid conditions that would make sowing impossible

return false to denote unsuccessful sow

Create flag to keep track of which side we are at, index for position, and bean hand to stop the iteration

Repeatedly until we run out of beads:

Check for condition which depends on which side we started on, where we are currently at, and if we reached the end of the vector

Increment index we are at in the vertex or set to 0 if we reached the end of vertex

Update flag based off what side we are on

Decrement number of beans

Give bean to position and make consideration that we only give to pot if we are on that side

Update endSide variable based on flag

Update endHole variable based on index we ended up on in vector

Return true to denote successful sow

bool Board::moveToPot(Side s, int hole, Side potOwner)

Check for invalid hole entered

Return -1 to denote that the function did not succeed in moving to pot

Utilizing conditional statements, we keep track of the potOwner and side s we want to transfer to so that we can specify the vertex we are transferring to and from

If North is inputted for potOwner

If the inputted hole we are taking from is North

Add beans from hole to North pot

Set hole to 0 beans

If the inputted hole we are taking from is South

Add beans from hole to north’s pot

Set hole to 0 beans

If South is inputted for PotOwner

If the inputted hole we are taking from is North

Add beans from hole to north’s pot

Set hole to 0 beans

If the inputted hole we are taking from is South

Add beans from hole to north’s pot

Set hole to 0 beans

Return true to denote successful transfer

void Game::status(bool& over, bool& hasWinner, Side& winner) const

initialize countNorth and countSouth variable to count number of times we encounter hole that is empty

Repeatedly until we check all the holes for both side of the board

If the number of beans on the current north side’s hole is 0

Increment countNorth

If the number of beans on the current south side’s hole is 0

Increment countSouth

If countNorth and countSouth is not equal to the number of holes we have

Denote that the game is not over

Otherwise the game is over

If the number of beans in both pots are equal

Denote that the game has no winner

Otherwise check who has more beans in their pot and modify winner accordingly

bool Game::move()

Call status function which we use to track if the game is over

If game is over

Return false to denote that no move is allowed

Otherwise

Create conditional statement that checks if our player is interactive

We will modify the interface so that the user has to press enter to continue, so that the moves don’t just happen all at once

Create storage variable hole\_num that holds what hole we are gonna use to move from

Store within hole\_num the number ascertained by the chooseMove function from player that operates based off the player type

Call sow function and then proceed to check certain conditions so that we can implement the rules of kalah

If we end up on our own hole

Give the player another turn

If we end up at an empty hole and the opponent’s hole across is not empty

Capture the beans

Otherwise, change turn

If either side of the board is empty

To modify the board correctly, transfer all of the beans on each side to their respective pots so that we update the final pots to winning scores

void Game::play()

Check to see if either side of the board is empty

Sweep both sides into their respective pots so the final board is updated to how it shoul be

Otherwise

Display the initial state of the board

Call status function to check that the game is not over

Repeatedly until game is over

Call previously defined move function

Call display function

Call status function to update whether the game is in fact over

Display the winner if there is one

void SmartPlayer::minimax(int height, const Board &b, Side s, int &bestHole, int &value, AlarmClock& ac) const

Initialize height that we will go up to be a constant 4 and value to be set at -999 in the beginning for comparison purposes

Base case to stop recursion or impede function when either side of board has no beans

Set bestHole to -1

Update value depending on one of three condition

If the final value in our pot equals the final value in our opponent’s pot

Set value to 0

If the final value in our pot is greater than the final value in our opponent’s pot

Set value to 999

If the final value in our pot is less than the final value in our opponent’s pot

Set value to -999

Return to escape

Base case to stop recursion or impede function if our alarm times out

Update value to be the beans in our pot minus to beans in our opponent’s

Base case to stop recursion or impede function if we reach our maximum depth

Update value to be the beans in our pot minus to beans in our opponent’s

Iterating through all holes of number that have beans in them (so we do not waste time)

Sow from the spot we are at the create possible board outcomes

If rules of capture are satiated

Move the hole that was captured along with our corresponding hole to the pot

Recurse using minimax depending on specific conditions

If condition is satiated where the player ends on their own pot and account for a repeat turn

Compare current situation of value to recursed situation of value, inputting the same side as an argument

If our new value is more ideal

Set bestHole to i

Set value to the more ideal value

Compare current situation of value to recursed situation of value, inputting the opposite side as argument

Set bestHole to i

Set value to the more ideal value

Return to escape

Note of Known Bugs and Problems with Project

Firstly, for Player.cpp the conditional statement for the timedOut function does not work as fully intended in the compiler of visual studio. Upon timing the amount of time it takes to return a move, the time can take anywhere from eight to ten seconds. In order to counteract this issue, I set the depth to 4, so that I never ran into the issue where time would be a problem for the running of minimax.

Another issue that I ran into in the creation of this project was the implementation and organization large functions like minimax. If I were able to go back, I would utilize the given function opponent to allow for better readability of someone looking at the code. Futhermore, I would create helper functions instead of just coding the work all in one function, such as evaluate would yield a corresponding number based off how good a move was. This would yield to the efficacy of my project, and would further lend to the readability of my code in part by people looking at it.

Lastly, another problem I ran into when it came to the creation of this project was the fact that everything built up on other components. When I was creating the project and adding other components to each other, I would test the newly added component and realize that it did not work fully as intended because the smaller components that the new component built off upon did not consider all cases or had logical errors. Although I was able to sort out the issues that I found through my test cases, if I were to go back, I would approach an extensive project such as this one by building incrementally.

Test Cases

Board v(4, 0); //ensures we can make a board with 0 beans per hole

assert(v.beansInPlay(SOUTH) == 0); //ensures beansInPlay works with sides that have totals of 0

assert(v.beansInPlay(NORTH) == 0);

Side finalSide;

int finalHole;

Board negativeBeans(4, -1);

for (int i = 1; i <= 4; i++)

{

assert(negativeBeans.beans(SOUTH, i) == 0); //if initial beans is set to negative act as if it is 0

assert(negativeBeans.beans(NORTH, i) == 0);

}

Board negativeHoles(-1, 4);

assert(negativeHoles.holes() == 1); //if initial holes is set to negative act as if it is 1

assert(negativeHoles.beans(SOUTH, 1) == 4); //ensures that we are creating the necessarily hole

Board zeroHoles(0, 4);

assert(zeroHoles.holes() == 1); //if initial holes is set to zero act as if it is 1

Board b(4, 4);

assert(b.holes() == 4);

assert(b.beans(SOUTH, 2) == 4); //used to test bean function's ability to count the number of beans in a holes

//negative values, 0, and values that are bigger than the number of holes on a side shold return -1

assert(b.beans(SOUTH, 5) == -1);

assert(b.beans(SOUTH, -2) == -1);

//test to make sure that we can properly count the number of beans on both north and south's sides

assert(b.beansInPlay(SOUTH) == 16);

assert(b.beansInPlay(NORTH) == 16);

assert(b.totalBeans() == 32);

b.setBeans(NORTH, 0, 4);

assert(b.beans(NORTH, 0) == 4); //used to test bean function's ability to count the number of beans in pot

//the following testing is to ensure that sow properly disburses the beans, and also updates the finalSide and finalHole correctly

assert(b.sow(SOUTH, 2, finalSide, finalHole) == true);

assert(b.beans(SOUTH, 2) == 0);

assert(b.beans(SOUTH, 3) == 5);

assert(b.beans(SOUTH, 4) == 5);

assert(b.beans(SOUTH, 0) == 1);

assert(b.beans(NORTH, 4) == 5);

assert(finalSide == NORTH && finalHole == 4);

Board c(4, 9);

assert(c.sow(SOUTH, 5, finalSide, finalHole) == false); //we cannot sow beginning at the pot or bigger than the alotted holesmove

assert(c.sow(SOUTH, 0, finalSide, finalHole) == false); //we cannot sow pots

assert(c.sow(SOUTH, -1, finalSide, finalHole) == false); //we cannot sow negative hole numbers

assert(c.sow(SOUTH, 4, finalSide, finalHole) == true); //the following is to ensure multuple components of sow function works

assert(c.beans(NORTH, 4) == 10);

assert(c.beans(NORTH, 3) == 10);

assert(c.beans(NORTH, 2) == 10);

assert(c.beans(NORTH, 1) == 10);

assert(c.beans(NORTH, 0) == 0); //this part ensures that if we start from the south side, the sow function should not add to north's pot

assert(c.beans(SOUTH, 1) == 10); //we cycle back to the south side

assert(c.beans(SOUTH, 2) == 10);

assert(c.beans(SOUTH, 3) == 10);

assert(c.beans(SOUTH, 4) == 1);

assert(c.beans(SOUTH, 0) == 1); //we should still be able to add to our own pot

assert(c.beans(NORTH, 4) == 10); //This value should be the same to signal that we escaped the iterations

Board sowTest(4, 4);

sowTest.sow(SOUTH, 1, finalSide, finalHole);

assert(finalHole == 0 && finalSide == SOUTH); //ensures that when we land at the final Hole, endSide is set to the correct side and the finalHole is 0

sowTest.sow(NORTH, 4, finalSide, finalHole);

assert(finalHole == 0 && finalSide == NORTH); //ensures that when we land at the final Hole, endSide is set to the correct side and the finalHole is 0

//Following test cases ensure that all of the values on board updated to the right value

assert(sowTest.beans(SOUTH, 0) == 1);

assert(sowTest.beans(SOUTH, 1) == 0);

assert(sowTest.beans(SOUTH, 2) == 5);

assert(sowTest.beans(SOUTH, 3) == 5);

assert(sowTest.beans(SOUTH, 4) == 5);

assert(sowTest.beans(NORTH, 0) == 1);

assert(sowTest.beans(NORTH, 4) == 0);

assert(sowTest.beans(NORTH, 3) == 5);

assert(sowTest.beans(NORTH, 2) == 5);

assert(sowTest.beans(NORTH, 1) == 5);

Board d(4, 4);

//test cases to ensure that move pot changes the pot value and hole necessary

assert(d.moveToPot(SOUTH, 0, SOUTH) == false); //we cannot movetoPot from Pot

assert(d.moveToPot(SOUTH, -1, SOUTH) == false); //we cannot movetoPot from negative hole

assert(d.moveToPot(SOUTH, 5, SOUTH) == false); //we cannot movetoPot from Pot that is bigger than holes

assert(d.moveToPot(SOUTH, 1, SOUTH) == true);

assert(d.beans(SOUTH, 1) == 0);

assert(d.beans(SOUTH, 0) == 4);

assert(d.moveToPot(NORTH, 1, SOUTH) == true); //ensures we return true when we move beans to pot sucessfully

assert(d.beans(NORTH, 1) == 0);

assert(d.beans(SOUTH, 0) == 8);

assert(d.moveToPot(SOUTH, 2, NORTH) == true);

assert(d.beans(SOUTH, 2) == 0);

assert(d.beans(NORTH, 0) == 4);

assert(d.moveToPot(NORTH, 2, NORTH) == true);

assert(d.beans(NORTH, 2) == 0);

assert(d.beans(NORTH, 0) == 8);

//the following asserts ensure setBeans correctly modifes the number of beans in a specific hole

assert(d.setBeans(SOUTH, -1, 3) == false); //we want to return false if holes is negative

assert(d.setBeans(SOUTH, 5, 3) == false); //we want to return false if hole number exceeds the number of holes we have

assert(d.setBeans(NORTH, 0, 20)==true); //ensures that we return true when we modify a hole sucessfully

assert(d.setBeans(NORTH, 1, 19)==true);

assert(d.setBeans(SOUTH, 0, 18)== true);

assert(d.setBeans(SOUTH, 1, 17)== true);

assert(d.beans(NORTH, 0) == 20); //ensures that we can modify the pot with our setBeans function

assert(d.beans(NORTH, 1) == 19); //ensures we can modify the hole with our setBeans functions

assert(d.beans(SOUTH, 0) == 18); //ensures that we can modify the pot with our setBeans function

assert(d.beans(SOUTH, 1) == 17); //ensures we can modify the hole with our setBeans functions

//Tests for HumanPlayer

HumanPlayer test1("dog");

assert(test1.name() == "dog"); //ensures that name function works correctly

assert(test1.isInteractive() == true); //tests to make sure that isInteractive of HumanPlayer returns true

Board test5(4, 4);

test5.setBeans(SOUTH, 3, 0);

cout << "TO PASS THIS TEST CASE YOU MUST ENTER 1" << endl;

assert(test1.chooseMove(test5, SOUTH) == 1);

//For this test, I also entered 3 to make sure that it properly prompted for another value

//Upon entering 1, this should return 1

//Tests for BadPlayer

BadPlayer test2("cat");

assert(test2.name() == "cat"); //ensures that name function works correctly

assert(test2.isInteractive() == false); //tests to make sure that isInteractive of HumanPlayer returns false

Board test6(4, 4);

test6.setBeans(SOUTH, 1, 0);

assert(test2.chooseMove(test6, SOUTH) == 2); //we should return the first available hole with lowest position that contains beans

test6.setBeans(SOUTH, 2, 0);

test6.setBeans(SOUTH, 3, 0);

assert(test2.chooseMove(test6, SOUTH) == 4); //we should return the first available hole with lowest position that contains beans

//Tests for SmartPlayer

SmartPlayer test3("bird");

assert(test3.name() == "bird"); //ensures that name function works correctly

assert(test3.isInteractive() == false); //tests to make sure that isInteractive of HumanPlayer returns false

Board Test7(6, 0);

Test7.setBeans(SOUTH, 5, 1);

Test7.setBeans(NORTH, 3, 2);

Test7.setBeans(NORTH, 6, 8);

Test7.setBeans(SOUTH, 3, 2);

Test7.setBeans(SOUTH, 2, 2);

// 0 0 2 0 0 8

//0 0

// 0 2 2 0 1 0

assert(test3.chooseMove(Test7, SOUTH) == 5); //This is a guaranteed win that we want smartPlayer to recognize

Board Test8(6, 1);

Test8.setBeans(NORTH, 5, 0);

Test8.setBeans(SOUTH, 5, 8);

Test8.setBeans(SOUTH, 6, 0);

// 1 1 1 1 0 1

//0 0

// 1 1 1 1 8 0

assert(test3.chooseMove(Test8, SOUTH) == 5); //if South does not pick hole 5, North wins no matter what

Board Test9(6, 1);

assert(test3.chooseMove(Test9, SOUTH) == 6); //if South picks 1, they get another turn, which is ideal

// 1 1 1 1 1 1

//0 0

// 1 1 1 1 1 1

Board InvalidBoard1(4, 0); //We will create Boards that are expected for choseMove function to return false

InvalidBoard1.setBeans(SOUTH, 1, 0);

Board InvalidBoard2(4, 0);

InvalidBoard2.setBeans(NORTH, 1, 0);

Board InvalidBoard3(4, 0);

//No matter which side has no beans, we want to return -1 if we try to make a move from either side for HumanPlayer

assert(test1.chooseMove(InvalidBoard1, SOUTH) == -1);

assert(test1.chooseMove(InvalidBoard2, SOUTH) == -1);

assert(test1.chooseMove(InvalidBoard1, NORTH) == -1);

assert(test1.chooseMove(InvalidBoard2, NORTH) == -1);

//No matter which side has no beans, we want to return -1 if we try to make a move from either side for HumanPlayer

assert(test2.chooseMove(InvalidBoard1, SOUTH) == -1);

assert(test2.chooseMove(InvalidBoard2, SOUTH) == -1);

assert(test2.chooseMove(InvalidBoard1, NORTH) == -1);

assert(test2.chooseMove(InvalidBoard2, NORTH) == -1);

//No matter which side has no beans, we want to return -1 if we try to make a move from either side for SmartPlayer

assert(test3.chooseMove(InvalidBoard1, SOUTH) == -1);

assert(test3.chooseMove(InvalidBoard2, SOUTH) == -1);

assert(test3.chooseMove(InvalidBoard1, NORTH) == -1);

assert(test3.chooseMove(InvalidBoard2, NORTH) == -1);

assert(test3.chooseMove(InvalidBoard3, NORTH) == -1);

assert(test3.chooseMove(InvalidBoard3, SOUTH) == -1);

cout << "Passed all Player and Board Test cases" << endl << endl;

HumanPlayer player("Ryan");

BadPlayer comp("Computer");

HumanPlayer\* playerpoint = &player; //we create a pointer to player

BadPlayer\* comppoint = &comp; //we create a pointer to comp

Game g(d, comppoint, playerpoint); //This ensures that our constructor for Game works appropriately

bool overCheck;

bool hasWinnerCheck;

Side winnerCheck;

Board statusCheck(4,0);

statusCheck.setBeans(SOUTH, 1, 4); //ensures that our status check function works and updates over, hasWinner, and winner correctly

statusCheck.setBeans(NORTH, 0, 5); //we want to ensure that we are comparing the final items in the pots, adding the leftover beans to the correct side to determine the winner

Game statusGame(statusCheck, comppoint, playerpoint);

statusGame.status(overCheck, hasWinnerCheck, winnerCheck);

assert(overCheck == true && hasWinnerCheck == true && winnerCheck == NORTH);

Board statusCheck2(4, 0);

statusCheck2.setBeans(SOUTH, 1, 6); //ensures that our status check function works and updates over, hasWinner, and winner correctly

statusCheck2.setBeans(NORTH, 0, 5); //we want to ensure that we are comparing the final items in the pots, adding the leftover beans to the correct side to determine the winner

Game statusGame2(statusCheck2, comppoint, playerpoint);

statusGame2.status(overCheck, hasWinnerCheck, winnerCheck);

assert(overCheck == true && hasWinnerCheck == true && winnerCheck == SOUTH);

Board statusCheck3(4, 0);

statusCheck3.setBeans(SOUTH, 1, 6); //ensures that our status check function works and updates over, hasWinner, and winner correctly

statusCheck3.setBeans(NORTH, 1, 6); //we want to make sure that if there exist beans on both sides of our board, the game is not over and hasWinner and winner are unmodiied from previous conditions

Game statusGame3(statusCheck3, comppoint, playerpoint);

statusGame3.status(overCheck, hasWinnerCheck, winnerCheck);

assert(overCheck == false && hasWinnerCheck == true && winnerCheck == SOUTH); //hasWinnerCheck and winnerCheck should remain unchanged with calling of status on this board

hasWinnerCheck = false;

winnerCheck = NORTH;

assert(overCheck == false && hasWinnerCheck == false && winnerCheck == NORTH); //hasWinnerCheck and winnerCheck should remain unchanged with calling of status on this board

winnerCheck = SOUTH;

Board statusCheck4(4, 0); //ensures that our status check function works and updates over, hasWinner, and winner correctly when the result should yield a tie game

statusCheck4.setBeans(SOUTH, 1, 5); //These inputs should yield a tie game

statusCheck4.setBeans(NORTH, 0, 5); //we want to ensure that we are comparing the final items in the pots, adding the leftover beans to the correct side to determine the winner

Game statusGame4(statusCheck4, comppoint, playerpoint);

statusGame4.status(overCheck, hasWinnerCheck, winnerCheck);

assert(overCheck == true && hasWinnerCheck == false && winnerCheck == SOUTH); //winnerCheck should be unchanged

winnerCheck = NORTH;

assert(overCheck == true && hasWinnerCheck == false && winnerCheck == NORTH); //winnerCheck should be unchanged

//the following test cases are used so that we can test move

Board statusCheck5(4, 0);

statusCheck5.setBeans(SOUTH, 1, 5); //We want to test the game yields false if invalid game

Game gameCheck5(statusCheck5, &player, &comp);

assert(gameCheck5.move() == false);

Board statusCheck6(4, 0);

statusCheck6.setBeans(NORTH, 1, 5); //We want to test the game yields false if invalid game

Game gameCheck6(statusCheck6, &player, &comp);

assert(gameCheck6.move() == false);

Board statusCheck7(4, 0);

statusCheck7.setBeans(NORTH, 4, 1);

statusCheck7.setBeans(NORTH, 2, 2);

statusCheck7.setBeans(SOUTH, 4, 1);

statusCheck7.setBeans(SOUTH, 3, 2);

statusCheck7.setBeans(SOUTH, 2, 3);

statusCheck7.setBeans(SOUTH, 1, 1);

Game gameCheck7(statusCheck7, &player, &comp);

cout << "Enter 4, 3, and 2 to test this test case" << endl;

assert(gameCheck7.move() == true); //This test case ensures that move doesn't return true until we make a move that doesn't give the player another turn (we don't land on our pot) for humanPlayer

gameCheck7.display();

SmartPlayer cry("cry");

Game gameCheck8(statusCheck7, &cry, &player);

assert(gameCheck8.move() == true); //This test case ensures that move doesn't return true until we make a move that doesn't give the player another turn (we don't land on our pot), we also ensure the SmartPlayer picks optimally and that capture works as intended

gameCheck8.display();

Board statusCheck9(4, 0);

statusCheck9.setBeans(SOUTH, 3, 2);

statusCheck9.setBeans(NORTH, 3, 2);

cout << endl << "newTest" << endl << "Enter 3 followed by 4 to test this case" << endl;

Game gameCheck9(statusCheck9, &player, &cry); //This test case ensures move returns true even when we end at a pot and there are no beans left

assert(gameCheck9.move() == true);

BadPlayer Baddie("Baddie");

//This test case is to ensure that move doesn't return true until we make a move that does not land on the pot

Board StatusCheck10(4, 0);

StatusCheck10.setBeans(NORTH, 3, 2);

StatusCheck10.setBeans(SOUTH, 1, 4);

StatusCheck10.setBeans(SOUTH, 2, 2);

Game GameCheck10(StatusCheck10, &Baddie, &comp);

assert(GameCheck10.move() == true);

GameCheck10.display();

Board StatusCheck11(4, 0);

StatusCheck11.setBeans(SOUTH, 1, 1);

StatusCheck11.setBeans(NORTH, 2, 5);

Game GameCheck11(StatusCheck11, &Baddie, &comp);

assert(GameCheck11.move() == true); //The following test cases ensure that capture succeeds and that move returns true

assert(GameCheck11.beans(NORTH, 2) == 0);

assert(GameCheck11.beans(SOUTH, 1) == 0);

assert(GameCheck11.beans(SOUTH, 0) == 6);

Board testMoveBoard(5, 2);

HumanPlayer Aa("Ryan");

HumanPlayer Ba("Derek");

HumanPlayer\* playerA = &Aa; //we create a pointer to player

HumanPlayer\* playerB = &Ba; //we create a pointer to comp

Game testMoveGame(testMoveBoard, playerA, playerB);

//testMoveGame.play(); //when we input 1

Board testFunct(6, 4);

testFunct.setBeans(SOUTH, 1, 0);

testFunct.setBeans(SOUTH, 2, 0);

testFunct.setBeans(SOUTH, 3, 0);

testFunct.setBeans(SOUTH, 4, 0);

testFunct.setBeans(SOUTH, 5, 0);

testFunct.setBeans(SOUTH, 6, 0);

SmartPlayer XX("cat");

assert(XX.chooseMove(testFunct, NORTH) == -1); //we wanna make sure that chooseMove does not work if either side has no choices of marbles to pick from

assert(XX.chooseMove(testFunct, SOUTH) == -1); //we wanna make sure that chooseMove does not work if either side has no choices of marbles to pick from

Board humanTest(3, 0);

humanTest.setBeans(SOUTH, 1, 4);

SmartPlayer XT("ME");

assert(XT.chooseMove(humanTest, SOUTH)== -1); //if either side has 0 beans, we need to return -1

humanTest.setBeans(SOUTH, 1, 0);

humanTest.setBeans(NORTH, 1, 4);

assert(XT.chooseMove(humanTest, SOUTH) == -1); //if either side has 0 beans we need to return -1

Board finalBoard(6, 4);

Game finalGameTest(finalBoard, &XT, &XX);

finalGameTest.play();

//final testing to see if BadPlayer in south position and SmartPlayer in north positionwork as intended for play function

Board finalBoard2(6, 4);

BadPlayer BP2("Chris");

SmartPlayer SP2("Say");

Game finalGame2(finalBoard2, &BP2, &SP2);

finalGame2.play();

//final testing to see if HumanPlayer in south position and HumanPlayer in north position work as intended for play function

Board finalBoard3(6, 4);

HumanPlayer BP3("Chris");

HumanPlayer SP3("Say");

Game finalGame3(finalBoard3, &BP3, &SP3);

finalGame3.play();

//final testing to see if SmartPlayer in south position and HumanPlayer in north position work as intended for play function

Board finalBoard4(6, 4);

SmartPlayer BP4("Chris");

HumanPlayer SP4("Say");

Game finalGame4(finalBoard4, &BP4, &SP4);

finalGame4.play();

//The following test cases are used to ensure that the result that is depicted is shown correctly and yields the correct winner

Board finalBoard5(4, 0);

finalBoard5.setBeans(SOUTH, 0, 4);

finalBoard5.setBeans(NORTH, 0, 3);

finalBoard5.setBeans(NORTH, 1, 1);

SmartPlayer BP5("Chris");

HumanPlayer SP5("Say");

Game finalGame5(finalBoard5, &BP5, &SP5);

finalGame5.play(); // This should yield no winner and sweep the north side when the south side has no moves avilable in the very beginning

assert(finalGame5.beans(SOUTH, 0) == 4); //The beans should have been swept

assert(finalGame5.beans(NORTH, 0) == 4); //The beans should have been swept

for (int i = 1; i <= 4; i++)

{

assert(finalGame5.beans(NORTH, i) == 0); //all the holes should posess 0 beans

assert(finalGame5.beans(SOUTH, i) == 0); //all the holes should posess 0 beans

}

Board finalBoard6(4, 0);

finalBoard6.setBeans(NORTH, 0, 4);

finalBoard6.setBeans(SOUTH, 0, 3);

finalBoard6.setBeans(SOUTH, 1, 1);

SmartPlayer BP6("Chris");

HumanPlayer SP6("Say");

Game finalGame6(finalBoard6, &BP6, &SP6);

finalGame6.play(); // This should yield no winner and sweep the north side when the south side has no moves avilable in the very beginning

assert(finalGame6.beans(SOUTH, 0) == 4); //The beans should have been swept

assert(finalGame6.beans(NORTH, 0) == 4); //The beans should have been swept

for (int i = 1; i <= 4; i++)

{

assert(finalGame6.beans(NORTH, i) == 0); //all the holes should posess 0 beans

assert(finalGame6.beans(SOUTH, i) == 0); //all the holes should posess 0 beans

}

Board finalBoard7(4, 0);

finalBoard7.setBeans(NORTH, 0, 5);

finalBoard7.setBeans(SOUTH, 0, 3);

finalBoard7.setBeans(SOUTH, 1, 1);

SmartPlayer BP7("Chris");

HumanPlayer SP7("Say");

Game finalGame7(finalBoard7, &BP7, &SP7);

finalGame7.play(); //testing to make sure Say is declared as winner

assert(finalGame7.beans(SOUTH, 0) == 4); //The beans should have been swept

assert(finalGame7.beans(NORTH, 0) == 5); //testing to make sure all leftover beans sent to pot

finalBoard7.setBeans(NORTH, 1, 2);

finalBoard7.setBeans(NORTH, 0, 3);

finalBoard7.setBeans(SOUTH, 1, 0);

Game finalGame8(finalBoard7, &BP7, &SP7);

finalGame8.play(); //testing to make sure Say is declared as winner

assert(finalGame8.beans(SOUTH, 0) == 3); //The beans should have been swept

assert(finalGame8.beans(NORTH, 0) == 5); //testing to make sure all leftover beans sent to pot

finalBoard7.setBeans(NORTH, 1, 0);

finalBoard7.setBeans(NORTH, 0, 3);

finalBoard7.setBeans(SOUTH, 1, 3);

Game finalGame9(finalBoard7, &BP7, &SP7);

finalGame9.play(); //testing to make sure Chris is declared as winner

assert(finalGame9.beans(SOUTH, 0) == 6); //The beans should have been swept

assert(finalGame9.beans(NORTH, 0) == 3);//testing to make sure all leftover beans sent to pot