**Project 3 Writeup**

**Description of classes:**

**Board:**

For the Board class, the major data structures I used are vectors which were implemented to hold the beans in the north and south sides, and it was defined to store and update the values in the holes excluding the pot, so when altering the board, we’d start at position 1 in the vector which corresponds to hole 1, and go up to and including the total number of holes which would be the last hole since the 0th position in the vector holds the pot’s values. I used vectors to store my board values because it was easy to dynamically allocate storage. I defined 2 private int variables to store the number of holes and number of beans per hole, 2 vectors for north and south with the length being the number of holes (minus the pot) and holding the number of beans in each hole, and 2 private functions, validHole(int hole) const and validSide(Side s) const that determine whether or not the hole and side are valid. It’s a valid side if the side is either north or south, and invalid if it isn’t, and it’s a valid hole if it’s between 1 and the total number of holes in the game since we exclude the pot as a possible hole to start most moves with. These were helpful to determine the validity before checking certain states of the board or making moves like sowing.

**Player:**

For the player class, I added a private string variable for the name of each player in the base class. I overloaded each chooseMove function in order to make a move depending on the type of player you have; if it’s a human player then it will take in input from the user as long as it’s a valid hole, and if it’s a bad player I implemented it to make a move using the first non-empty hole, valid.

**SmartPlayer::chooseMove:**

For the private interface of SmartPlayer, I added an evaluation function: int evaluate(Side s, Board b) const which takes care of the heuristics to evaluate the board positions for the minimax algorithm. It checks if the game is over and our side has more beans in play and in our pot, then we will be the maximizer and returns a very large value, if our side has less beans in play and in our pot, then we will be the minimizer and returns a very small value since this means the opponent will win, and otherwise returns 0 since it will be a tie. If the game is not over and our side has more than half of the beans in our pot, then you’re most likely to win so it returns a very large value, if our side has less than half of the beans in our pot, then you’re most likely to lose so it returns a very small value, and otherwise, it’ll return the difference in the pots since we want to maximize difference between the beans for our side. For the overloaded chooseMove function, chooseMove(Side s, Side turn, Board b, int depth, int& bestHole, int& value, JumpyTimer& time) const, I added the parameters to check whose side is choosing the move, whose turn it is, the current board state, the depth of the recursion we want to check, the best hole to choose, the value associated with the best hole, and a time limit on the time it takes for the smart player to choose a move. This function recursively determines a player's best move and returns the best hole to sow from and the value that can be achieved if that move is made, or -1 if the game is over or we terminate the recursion once the depth is 4, or if the time is 5 seconds. We begin by initializing the value to a very small number if we’re on our side and it’s our turn or a very large number if it’s not our turn, and making a copy of the board to make the move on for the player and its opponents countermove. At the end, we apply the minimax algorithm by updating the value if we see a larger value and we’re the maximizer, or updating the value if we see a smaller value and we’re the minimizer and setting the best hole accordingly, and then unmaking the move by setting the copy board back to the original. Finally, within the original virtual function we call the overload chooseMove function with the same side and turn, 0 for the depth, and best hole and value variables, and return the best hole.

**Game:**

For the game class, I added a Board variable, 2 pointers to Players, and a Side variable to indicate the turn. I didn’t add any data structures here since all this class is essentially doing is managing the game and configuring the board display.

**Note of Inefficiencies:**

Since the SmartPlayers heuristics are designed such that it plays against a player that will choose the best possible move, when it plays against the BadPlayer, it does not always win since the BadPlayer simply chooses the first available, valid hole, not necessarily the best possible move. I set my depth to be a max of 10 as I found it worked best when the depth wasn’t larger than this value since the SmartPlayer would end up winning more often once it was 10.

**Test Cases:**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <cassert>

#include <iostream>

**int** main()

{

//BOARD:

// we check that the board is initialized right when initialized with invalid variables

Board invalid(-1, -1);

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

assert(invalid.beans(NORTH, 0) == 0);

assert(invalid.beans(SOUTH, 0) == 0);

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

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

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

assert(invalid.beansInPlay(SOUTH) == 0);

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

// Creating a board

Board b(5, 3);

assert(b.beans(NORTH, 0) == 0);

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

**for** (**int** i = 1; i <= 5; i++)

{

assert(b.beans(NORTH, i) == 3); //checks that beans() works as well

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

}

assert(b.beansInPlay(NORTH) == 15);

assert(b.beansInPlay(SOUTH) == 15);

assert(b.totalBeans() == 30);

// beans

assert(b.beans(NORTH, 20) == -1);

assert(b.beans(NORTH, 0) == 0); //beans() should work for pot as well

// move to pot

assert(!b.moveToPot(NORTH, 0, NORTH));

assert(!b.moveToPot(NORTH, 6, SOUTH));

assert(b.moveToPot(NORTH, 5, NORTH));

assert(b.beans(NORTH, 5) == 0); // check that all beans have been moved

assert(b.beansInPlay(NORTH) == 12);

assert(b.moveToPot(NORTH, 5, NORTH)); //should still work even though beans in that hole is 0

//test sow

Board c(6, 4);

Side endSide;

**int** endhole;

assert(c.sow(NORTH, 3, endSide, endhole));

assert(endhole == 1); // if sowed from north's side and hole 3, should end up in hole 1 of south side

assert(endSide == SOUTH);

assert(c.sow(NORTH, 1, endSide, endhole)); // if sowed from north's side and hole 1, should end up in hole 4 of south side

assert(endhole == 4);

assert(endSide == SOUTH);

assert(!c.sow(NORTH, 0, endSide, endhole));

assert(c.setBeans(SOUTH, 1, 20));

assert(c.sow(SOUTH, 1, endSide, endhole));

assert(endhole == 6);

assert(endSide == NORTH);

assert(c.sow(SOUTH, 1, endSide, endhole));

assert(endhole == 2);

assert(endSide == SOUTH);

cout << "Passed all Board Tests\n";

//PLAYER:

// Human player

HumanPlayer will("will");

assert(will.isInteractive());

// Bad player

Board d(6, 4);

assert(d.setBeans(SOUTH, 1, 0));

assert(d.setBeans(SOUTH, 2, 0));

assert(d.setBeans(SOUTH, 3, 0));

assert(d.setBeans(SOUTH, 4, 0));

assert(d.setBeans(SOUTH, 5, 0));

assert(d.setBeans(SOUTH, 6, 0));

BadPlayer bad("Dumb");

assert(!bad.isInteractive());

assert(bad.name() == "Dumb");

// Bad player should choose the the first hole that is availiable

assert(bad.chooseMove(d, SOUTH) == -1);

assert(bad.chooseMove(d, NORTH) == 1);

d.sow(NORTH, 1, endSide, endhole);

assert(bad.chooseMove(d, NORTH) == 2);

d.sow(NORTH, 2, endSide, endhole);

assert(bad.chooseMove(d, NORTH) == 1);

cout << "Passed all Player Tests\n";

//SMARTPLAYER:

// Testing smartPlayer player make the best decision

Board p1(4, 0);

SmartPlayer sp("Lisa");

p1.setBeans(NORTH, 0, 4);

p1.setBeans(NORTH, 1, 0);

p1.setBeans(NORTH, 2, 0);

p1.setBeans(NORTH, 3, 1);

p1.setBeans(SOUTH, 0, 1);

p1.setBeans(SOUTH, 1, 2);

p1.setBeans(SOUTH, 2, 0);

p1.setBeans(SOUTH, 4, 1);

/\*4 0 0 1 0

2 0 0 1 1\*/

assert(sp.chooseMove(p1, SOUTH) == 4 || 1); // hole 4 and 1 would both lead to victory

cout << "Passed all SmartPlayer Tests\n";

//GAME:

// initializing a game itself

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board board(3, 0);

board.setBeans(SOUTH, 1, 3);

board.setBeans(NORTH, 2, 2);

board.setBeans(NORTH, 3, 3);

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

**bool** over;

**bool** hasWinner;

Side winner;

// Homer

// 0 2 3

// 0 0

// 3 0 0

// Bart

g.status(over, hasWinner, winner);

assert(!over && !hasWinner && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 2 && g.beans(NORTH, 3) == 3 &&

g.beans(SOUTH, 1) == 3 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

// // 0 2 3

// // 0 1

// // 0 0 2

g.status(over, hasWinner, winner); // tests that game takes care of capture correctly

assert(!over && !hasWinner && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 1 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 2 && g.beans(NORTH, 3) == 3 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 2);

g.move(NORTH);

// // 0 1 0

// // 4 1

// // 0 0 2

g.status(over, hasWinner, winner);

assert(!over && !hasWinner && g.beans(NORTH, POT) == 4 && g.beans(SOUTH, POT) == 1 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 2);

g.move(SOUTH);

// // 1 0 1

// // 4 2

// // 0 0 0

g.status(over, hasWinner, winner);

assert(over && hasWinner && g.beans(NORTH, POT) == 4 && g.beans(SOUTH, POT) == 2 &&

g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 1 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

g.move(NORTH);

// // 0 1 0

// // 5 2

// // 0 0 0

g.status(over, hasWinner, winner); // tests that the game does the sweeping for you

assert(over && g.beans(NORTH, POT) == 5 && g.beans(SOUTH, POT) == 2 &&

g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 0 &&

g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

assert(hasWinner && winner == NORTH);

cout << "Passed all Game Tests\n";

}

**Pseudocode:**

**Board:**

**Board::Board(int nHoles, int nInitialBeansPerHole)**

If holes or beans are invalid

set holes to 1

set beans to 0

set m\_nHoles to nHoles

set m\_nBeans to nInitialBeansPerHole

push 0s into north and south vector

Loop through from 1 to nHoles

push back the beans in each vector

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

check if hole is valid

if start size is NORTH

get amount of beans at hole

if zero return

set beans at hole to zero

start position for first loop is one hole to the left of chosen hole

while loop for looping through the board

backwards while loop for north

set the start for next loop to be the last hole

loop through south from first hole to last

return true

if start side is SOUTH

make a count of how many beans

make starting hole = 0

starting position for the first loop is one past the chosen hole

while loop for looping through the board

loop through south

set starting position for next loop to one

add to south's pot

loop backwards through north

return true

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

{

If invalid Side or hole

return false

int beans;

If on south side

{

get number of beans at the hole

Set beans at hole to 0

}

else // it's the north side

{

get number of beans at the hole

Set beans at hole to 0

}

If south’s pot

{

Increment souths pot with num of beans

return true;

}

else // it's north's pot

{

Increment norths pot with num of beans

return true;

}

}

**bool Board::validHole(int hole) const**

For loop through all holes excluding pot

{

if hole is valid

return true

}

return false

**​​bool Board::validSide(Side s) const**

{

If Side is valid

return true;

else { return false; }

}

**Player:**

**int HumanPlayer::chooseMove(const Board& b, Side s) const**

{

If no beans on either side

return -1;

Set inp to 0

While input is invalid

{

Keep prompting the player until they give a valid move

If valid inp given

break

}

return inp;

}

**int BadPlayer::chooseMove(const Board& b, Side s) const**

if no move available then return -1

Loop through available holes with beans in them

choose the first available hole with beans

return move

**int SmartPlayer::chooseMove(const Board& b, Side s) const**

{

JumpyTimer time(1000);

int bestHole;

int value;

Call overloaded chooseMove with side, side’s turn, current board, 0 depth, bestHole and value

return bestHole;

}

**int SmartPlayer::int evaluate(Side s, Board b) const**

if one of the sides has zero beans

if Side won

return 999999

if Side lost

return -999999

else tied { return 0 }

if beans in Side’s pot > ½ total beans

return 999999;

if beans in Side’s pot < ½ total beans

return -999999;

else { return side's pot - other side's pot }

**void SmartPlayer::chooseMove(Side s, Side turn, Board b, int depth, int& bestHole, int& value, JumpyTimer& time) const**

{

If no beans in either or holes invalid

{

bestHole = -1;

Set value to result from evaluate

return;

}

if depth == 10

{

bestHole = -1;

Set value to result from evaluate

return;

}

if time elapsed >= 4900

{

bestHole = -1;

Set value to result from evaluate

return;

}

If on our side

{

value = -999999;

v2 = -999999;

}

Else

{

value = 999999;

v2 = 999999;

}

Copy the board

Loop through all holes excluding pot

{

If theres beans in the hole

{

Sow the copy board

If land in pot

Call chooseMove with same side and turn

if ended in a capture

{

move beans from our hole to our pot

move beans from opponents hole to our pot

}

Call chooseMove for opponent

// minimax

If it’s our turn and our side

{

If value seen > original value

{

value = v2;

bestHole = i;

}

}

else // minimizer

If value seen < original value

{

value = v2;

bestHole = i;

}

}

}

Unmake the move

}

return;

}

**Game:**

**Game::Game(const Board& b, Player\* south, Player\* north)** : m\_board(b), m\_south(south), m\_north(north), m\_turn(SOUTH) {}

**void Game::display() const**

{

Print north player’s name

For loop through num of holes

Print num of beans and space

Print beans in norths pot on next line

For loop from 0 to num of holes

Print right amount of spaces in between the pots

Print beans in norths pot on next line

For loop through num of holes

Print num of beans and space

Print north player’s name

}

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

{

If can still make a move

{

over = false;

return;

}

If souths turn and south doesn’t have beans

{

over = true;

}

If souths turn and south has beans

{

over = false;

return;

}

If norths turn and north doesn’t have beans

{

over = true;

}

If norths turn and north has beans

{

over = false;

return;

}

If north has more beans in pot

{

over = true;

hasWinner = true;

winner = NORTH;

return;

}

If south has more beans in pot

{

over = true;

hasWinner = true;

winner = SOUTH;

return;

}

If north beans in pot == south beans in pot

{

over = true;

hasWinner = false;

return;

}

}

**bool Game::move(Side s)**

{

If Side is north

{

Make north’s move

If CANT make a move

{

For loop through all valid holes to move from

{

sweep all the beans from souths side into souths POT when north can't make a move

}

return false;

}

else // CAN make a move

{

if( able to sow )

{

if( endHole is POT ) // get another move

{

Display the board

Call move with the same Side

}

}

if( can capture )

{

move beans from norths hole to norths pot

move beans from souths hole to norths pot

}

return true;

}

}

else // side is SOUTH

{

make the move and store the hole

If CANT make a move

{

For loop through all valid holes to move from

{

sweep all the beans from souths side into souths POT when north can't make a move

}

return false;

}

else // CAN make a move

{

if( can sow )

{

if( endHole is POT )

{

Display the board

Call move with the same Side

}

}

if( can capture )

{

move beans from souths hole to souths pot

move beans from norths hole to souths pot

}

return true;

}

}

}

**void Game::play()**

{

Check status

Set bool pressEnter to false

Display the game

If neither players interactive

{

pressEnter = true; // you want to prompt the user to press enter

}

While game isn’t over

{

if(pressEnter)

{

std::cout << "Press ENTER to continue";

std::cin.ignore(10000, '\n');

}

Call move with south side

Display the board

Check status

if(pressEnter)

{

std::cout << "Press ENTER to continue";

std::cin.ignore();

}

Call move with south side

Display the board

Check status

}

If beans in play on north or south is 0

{

Loop through num of valid holes

{

Move beans in norths side to norths pot

Move beans in souths side to souths pot

}

Display baord

}

If there’s a winner

{

If winner is NORTH

{

std::cout << "The winner is " + m\_north->name() + "." << std::endl;

}

else // winner is SOUTH

{

std::cout << "The winner is " + m\_south->name() + "." << std::endl;

}

}

else if no winner

{

std::cout << "The game resulted in a tie." << std::endl;

}

}