**DESIGN**

Board stored as two vectors. one for the north side, one for the south side

note: when indexing from the south, should start from 1 since 0 is the pot

I have helper functions for Smart player which are eval, sow\_and\_capture, and minimax.

Eval: evaluates the current board state. The difference between the two pots is returned, a high number is returned if there is a winner

sow\_and\_capture: the capture functionality is in the game class move, which the players do not access, therefore this is a function that adds the capture rule and sow so the smart player can capture beans

Minimax: this function does the comparison of the evaluations of different board states

**SmartPlayer::chooseMove**

Heuristics: the larger the positive difference between two pots, the better. If a player had won, then return a large number, if the player lost return a small number

Choose move starts the minimax algorithm with a depth of 4 which starts the recursive calls for each possible branch and move of the original board.

The minimax will keep on growing the tree until it reaches the desired depth, at this point it will return the values that relate to the board state.

Minimax will then compare the value and the new value and set the best hole when it reaches the first recursive call. Therefore, the hole that is outputted is related to the best hole

Minimax also treats a move deeper in depth only when the turn switches sides.

**BOARD CLASS**

Board constructor

* initialize the values, check for invalid inputs. ex, < 0 holes and beans
* push on a pot for the north and south vectors
* for holes needed, push on value of beans into holes for each side, increase the number of holes

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

* check that the hole is a valid hole
* while there are still beans
* if hole is on the north side
  + get number of beans in the hole
  + set the number of beans in starting hole to 0
  + index negatively until hole = 0 or beans = 0
    - decrease the count of the beans by one
    - if there are still more beans when hole = 0, then continue to south side
    - return true
  + continue on the south side until you reach the pot.
  + If you reach the pot, skip over the south side pot.
    - decrease the count of beans by one each iteration
    - South side pot indexes up until there is no more
    - return true
* If the hole is on the south side
  + get number of beans in hole
  + set the beans in hole to 0
  + index positively until hole = m\_hole + 1
    - increase the count of the beans by one
    - if there are beans when hole = south m\_hole + 1, then loop through the north stuff
    - return true

**PLAYERS**

BadPlayer choice: I chose bad player to always choose the lowest integer pot. if it is an invalid pot, then they will choose the next greatest pot

HUMAN PLAYER virtual int chooseMove(const Board& b, Side s) const = 0;

* infinite loop to keep on running if the player chooses a move that will end in the pot, or warrants such move.
* in the loop, cin the values with a text prompt.
* check the values to see if they are valid, if not then go through the loop again and input a new value

**GAME**

void display() const;

* first row should be norths name, then the beans for north side, indexing positively to the right
* second row should be the pot values and in between the spaces
* third row should be the hole values for the south side index positively to the right, then the south person’s name

name # # # # #

# #

# # # # # name

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

* if there are still beans in play for both north and south, that means that the game should still be going
* if there are no more beans, if the number of beans in north pot is greater than south pot, then north wins, elsewise, the south wins
* if neither is greater, then there is no winner and has winner = false

bool move(Side s);

* recursive type of function
* if over, move all beans on the same side to the pot
  + return false
* if the move is possible
  + meaning that there are still beans in play for the side going, call the choose move on the side that is moving.
  + this is based on the m\_north / m\_south player
  + choose move will be the choice of hole for the player
  + if the player is not interactive, output the move that the computer player chose
  + if sow is true, meaning that the turn can still go, then call move again
* at the end of the turn, see if the number of beans in the end hole is 1
  + if yes, that means capture
  + capture opposite side and own hole with moveToPot

void play();

* every call, use status to check if there is a winner
* while the game is not over
  + call move and display
* once the game is over, if there is a winner
  + output the winning message
* if no winner, then output that there was a tie

**SMART PLAYER**

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

* call the minimax function with a depth of 4 and a timer that is started
* if the minimax returns in an invalid move, act like the bad player for this turn
* then return the best\_hole from either the minimax or the bad player methods

**int** SmartPlayer::minimax(**const** Board &b, Side s, **int** &best\_hole, **int** &value, **const** **int** depth, **int** original\_depth, Timer t) **const**{

* if there are no beans on the player side, or depth = 0 or time is finished
  + change the value to the current evaluation of the board
  + return the best\_hole default which is -1
* Create the subbranches of the board
  + for the number of holes
    - create a temp board using the passed in board (to not change the current board state)
    - call the sow\_and\_capture for this move and save its result as the end\_hole
    - if the end hole is 0, that means that the player can go again
      * recursive call minimax again and keep the side the same. do not count this as a turn
    - else means that the turn changed
      * recursive call minimax, changing the side to the opponent and decreasing depth
    - compare the new value from the minimax to the old value. if side is north and value is greater, than change it
    - if the depth original is even, that means that the player wants the maximum value
    - if the depth original is odd that means that the player wants the minimum value

**int** SmartPlayer::sow\_and\_capture(Board &b, Side s, **int** hole, Side &endSide) **const**{

* This is an altered sow that has the functionality to capture beans
* Before capture was in the game class so it is needed here
* complete a sow move
* if the number of beans that ended on is 1, and is on player side, and the opponent has beans on the opposite side
  + complete a capture
* return the end hole

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

* heuristic
  + difference in the number of beans in the two pots
* if the side is the winner, return 5000
* if the side is the loser return -5000
* else return the net difference between the two holes

**BUGS**

* Bug where the best move was turning out to be 0
  + fixed by only setting the best\_hole when the depth was at the top and if the move was a valid choice
* The game display would not redisplay after a sweep.
  + moved the sweep functionality to the play function and only was called after the game was stated to be over by the status function
* SmartPlayer was acting like a dumb player after a little bit
  + This was because my evaluation was not evaluating the right way and I tried to be smarter by stating that a player won if they had more than half the total beans
* Difficulty in figuring out which of the parameters each of the function would have needed. I didnt know what parameters to pass in
* I do not know if the timer is doing anything because the recursive call is also stopped by the depth == 0

**TEST CASES**

**// BOARD TEST FUNCTIONS**

**Side endside;**

**int endhole;**

**// there is no capturing in the board class**

**Board b1(3, 0);**

**assert(b1.beansInPlay(NORTH) == 0);**

**assert(b1.beans(NORTH, 2) == 0);**

**assert(b1.sow(NORTH, 2, endside, endhole ) == false);**

**Board b2(5, 3);**

**assert(b2.beansInPlay(NORTH) == 15);**

**assert(b2.beansInPlay(SOUTH) == 15);**

**assert(b2.beans(NORTH, 2) == 3);**

**assert(b2.sow(NORTH, 3, endside, endhole) == true);**

**assert(endside == NORTH && endhole == 0);**

**// the capture functions work**

**assert(b2.moveToPot(NORTH, 2, SOUTH) == true);**

**assert(b2.beans(SOUTH, 0) == 4);**

**assert(b2.totalBeans() == 30);**

**// testing board constructor**

**Board board3(-2, -3);**

**assert(board3.holes() == 1);**

**assert(board3.beans(NORTH, 1) == 0);**

**assert(board3.beans(SOUTH, -1) == -1);**

**// testing the beans function**

**Board board4(6, 4);**

**assert(board4.beans(NORTH, 7) == -1);**

**assert(board4.beans(SOUTH, 0) == 0);**

**assert(board4.beans(SOUTH, -2) == -1);**

**// testing beans in play and totalbeans**

**Board board5(4, 3);**

**assert(board5.beansInPlay(NORTH) == 12);**

**assert(board5.beansInPlay(SOUTH) == 12);**

**assert(board5.beansInPlay(Side(5)) == -1);**

**assert(board5.totalBeans() == 24);**

**// testing sow**

**Side endSide;**

**int endHole;**

**assert(!board5.sow(Side(3), 2, endSide, endHole));**

**assert(!board5.sow(NORTH, 5, endSide, endHole));**

**assert(!board5.sow(SOUTH, -1, endSide, endHole));**

**assert(board5.sow(NORTH, 1, endSide, endHole));**

**assert(endSide == SOUTH && endHole == 2);**

**// testing movetopot**

**Board board6(3, 2);**

**assert(!board6.moveToPot(NORTH, 0, NORTH));**

**assert(!board6.moveToPot(NORTH, 4, NORTH));**

**assert(!board6.moveToPot(Side(2), 2, NORTH));**

**// HUMANPLAYER TESTS**

**// HumanPlayer p1("p");**

**// assert(p1.name() == "p");**

**//**

**// BAD PLAYER TESTS**

#include <iostream>

#include "BadPlayer.h"

void runTest(const Board& board, Side side) {

BadPlayer player("BadPlayer");

int move = player.chooseMove(board, side);

std::cout << "Chosen move: " << move << std::endl;

}

int main() {

// Create a board with 4 holes and 6 initial beans in each hole

Board board(4, 6);

// Test case 1: No beans in play for South side

board.setBeans(SOUTH, 2, 0);

runTest(board, SOUTH);

// Expected output: Chosen move: -1

// Test case 2: First valid move is at hole 3 for North side

board.setBeans(NORTH, 1, 0);

board.setBeans(NORTH, 2, 0);

board.setBeans(NORTH, 3, 3);

board.setBeans(NORTH, 4, 0);

runTest(board, NORTH);

// Expected output: Chosen move: 3

// Test case 3: First valid move is at hole 1 for South side

board.setBeans(NORTH, 3, 0);

board.setBeans(SOUTH, 1, 2);

board.setBeans(SOUTH, 2, 0);

board.setBeans(SOUTH, 3, 0);

board.setBeans(SOUTH, 4, 1);

runTest(board, SOUTH);

// Expected output: Chosen move: 1

return 0;

// Smart Player

// testing if it chooses a best hole

cout <<s.minimax(b555, SOUTH, best\_hole, value, 3, 3, t);

// testing sow and capture

cout << s.sow\_and\_capture(b555, NORTH, 2, end\_side);

assert(s.eval(b555) == 1);

s.sow\_and\_capture(b555, NORTH, 1, end\_side);

// testing eval

cout << s.eval(b555, NORTH);

// testing minimax again

cout << s.minimax(b555, NORTH, best\_hole, value, 2, 2, t);

cout << best\_hole;

// Test case 1: No moves available for South side

Board board1(3, 0);

board1.setBeans(NORTH, 1, 2);

board1.setBeans(NORTH, 2, 1);

board1.setBeans(NORTH, 3, 2);

BadPlayer bp1("bad 1");

BadPlayer bp2("bad 2");

Game game1(board1, &bp1, &bp2);

game1.move(SOUTH);

assert(game1.beans(NORTH, POT) == 0 && game1.beans(SOUTH, POT) == 0 &&

game1.beans(NORTH, 1) == 2 && game1.beans(NORTH, 2) == 1 && game1.beans(NORTH, 3) == 2 &&

game1.beans(SOUTH, 1) == 0 && game1.beans(SOUTH, 2) == 0 && game1.beans(SOUTH, 3) == 0);

// Test case 2: Game over with a winner

Board board2(3, 0);

board2.setBeans(NORTH, 1, 1);

board2.setBeans(NORTH, 2, 0);

board2.setBeans(NORTH, 3, 0);

board2.setBeans(SOUTH, 1, 0);

board2.setBeans(SOUTH, 2, 1);

board2.setBeans(SOUTH, 3, 0);

BadPlayer bp3("bad 3");

BadPlayer bp4("bad 4");

Game game2(board2, &bp3, &bp4);

game2.move(NORTH);

bool over;

bool hasWinner;

Side winner;

game2.status(over, hasWinner, winner);

assert(over && hasWinner && winner == NORTH);

// Test case 3: Game over with no winner (tie)

Board board3(3, 0);

board3.setBeans(NORTH, 1, 1);

board3.setBeans(NORTH, 2, 0);

board3.setBeans(NORTH, 3, 0);

board3.setBeans(SOUTH, 1, 0);

board3.setBeans(SOUTH, 2, 1);

board3.setBeans(SOUTH, 3, 1);

BadPlayer bp5("bad 5");

BadPlayer bp6("bad 6");

Game game3(board3, &bp5, &bp6);

game3.move(NORTH);

game3.status(over, hasWinner, winner);

assert(over && !hasWinner && winner == NORTH);

return 0;

}