Project 3

1. In my **Board class** I make use of vectors data structure to store value for the beans and separately make use of two integer variables n\_pot and s\_pot to store respective pot values. I also make use of a private member function sowside which helps in the iteratively sowing of beans in after each condition is satisfied. Since the same code is being called multiple I just made that set of code a function to make things easy.

In my **SmartPlayer class** I make use of 3 private member functions (**int** eval, bool soworcapture, **int** minmax) where eval is used to evaluate the value of our board and return the value for the respective move. Bool soworcapture checks if we can again sow or capture a bean since these are two best things that can happen. And return true only if we can make another move. Int minmax implements the minmax algorithm to determine the best move possible iteratively given beans are present in the hole. It evaluate each move to a given depth and determines which move is the best for a given board and side.

In game I make use of pointers to Player in order to get the game working for different kind of players.

1. The SmartPlayer::chooseMove function primarily leverages three private functions to decide the optimal move. It iterates over all viable holes that contain beans and applies the minmax algorithm on every potential move executed on a temporary board. The board state is updated using the soworcapture function, which also determines if there's an opportunity for an additional move.

Minmax, within this context, decides whether it should minimize or maximize based on the possibility of making another move. It evaluates each feasible move, comparing the values with prior ones to decide if they're better or worse. The decision is based on whether the function aims to maximize or minimize its minmax value.

**Heuristics evaluation:**

It first checks for game ending scenarios, where one player has no beans to play with and then declares a win, loss or draw based on the total number of beans each player has. Then we check if any player has captured more than half the total beans in the game, assuming this situation will lead to a win. If neither of these conditions is met, the function considers the number of potential moves each player can make, with a bonus provided for having significantly more viable moves. Finally, if none of these situations apply, a combination of the number of captured beans and potential moves is used to compute a score, which effectively balances the elements of current advantage and future possibilities.



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

check if arguments are valid

if side is NORTH

get the number of beans in the hole

if 0 beans return false

set beans in that hole to 0

start sowing backwards in loop till end = 0

if beans = 0 store endSide, endHole value and return true

while there are beans

sow in north pot

if beans = 0 store endSide, endHole value and return true

if beans left sow in south side

check bean 0 condition again

if beans still left sow in NORTH again going backwards

check bean 0 condition again

if nothing return false

if side is SOUTH

get the number of beans in the hole

if 0 beans return false

set beans in that hole to 0

start sowing forward in loop till end = number holes

if beans = 0 store endSide, endHole value and return true

while there are beans

sow in south pot

if beans = 0 store endSide, endHole value and return true

if beans left sow in north side

check bean 0 condition again

if beans still left sow in south again going backwards

check bean 0 condition again

if nothing return false

if nothing return true

---------------------------------------------------------------------------------------------------------------------

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

if no valid move return false

infinite while loop

keep prompting user to enter valid hole

if valid then break out of loop

return selected hole

---------------------------------------------------------------------------------------------------------------------

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

if no valid move return false

loop through the board of given side

if beans in hole > 0

select that hole and break

return selected hole

---------------------------------------------------------------------------------------------------------------------

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

if no beans left return -1

loop through holes with beans:

make a copy of the board

if we can make another move

get the minmax of our move with a maximizer

if we cannot make another move

get minmax of our move with the minimizer

compare this newvalue with our old value

if greater then set it to new otherwise keep same

return move

---------------------------------------------------------------------------------------------------------------------

**int SmartPlayer::eval(const Board &board, Side s) const**

Check if the game is over

Is side won return 10000

Is side lost return -10000

If nothing return 0

If side captured more than half of the total beans return 10000

otherwise return -10000

Evaluate based on the number of possible moves each player has.

If side has significantly more possible moves return 5000

otherwise return -5000

If nothing then return a score based on the difference in the number of beans and possible moves

---------------------------------------------------------------------------------------------------------------------

**bool SmartPlayer::soworcapture(Board &board, int move, Side s) const**

sow the board

if we cannot sow

return false

if we can make a capture

capture

return false

if we can take another turn

return true

return false

---------------------------------------------------------------------------------------------------------------------

**Int SmartPlayer::minmax(const Board& board, Side s, bool isMax ,int depth, JumpyTimer& timer) const**

if no more moves available

if depth less than 0

is time exceeds 4.9s

return value of board

if we are maximizing

set bestvalhole to the most negative val

loop thru all possible moves

is time exceeds 4.9s

return value of board

create a copy of the board

sow the move

if we can sow we want to maximize

set newval to get the minmax value for this move one depth below

else minimize the opponents move

set newval to get minmax value for this move one depth below

if the minmax value is greater than bestvalhole

set bestvalhole to newval

else we are now going to minimize

set bestvalhole to the most positive val

loop thru all possible moves

is time exceeds 4.9s

return value of board

create a copy of the board

sow the move

if we can sow do it with maximizer

set newval to get the max value for this move one depth below

else sow with minimizer

set newval to get minmax value for thus move one depth below

if the minmax value is less than bestvalhole

set bestvalhole to newval

return bestvalhole

---------------------------------------------------------------------------------------------------------------------

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

if beans in play more than 0 on either side

over is false

if north side has more beans than south side

over is true

hasWinner is true

winner is north

return

if south side has more beans than north side

over is true

hasWinner is true

winner is south

return

if both have same number of beans

over is true

hasWinner is false

return

---------------------------------------------------------------------------------------------------------------------

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

if game is over

if there are beans on either side

move all beans on respective sides into respective pots

return false

infinite loop

is side is south

set hole to south’s move

store south’s name

else

set hole to north’s move

store north’s name

if invalid move

move all beans on respective sides into respective pots

break

if we can sow

display

if bean in play on side s not 0

continue

else

break

else if we can capture

capture

break

else

break

break

if beans in play on side s is 0

display

cout respective text

sweep remaining beans in respective pots

return true

---------------------------------------------------------------------------------------------------------------------

**void Game::play()**

set variable to keep track of current sides turn

If both are interactive

set presenter variable to true

check status

if not over

display game board

if current players move

make a move

else

opponent makes a move

check status again

if game not over and both interactive

prompt user to press enter

switch player sides

if there is a winner

print winner’s name

if not then

print it is a tie



I initially had problem in figuring out how to sow and loop back over the board following the conditions specified. Initially I had weird result because logic error but then figured it out.

Getting the minimax function to work was a big headache as it took time to figure out how to recursively call the function and determine the best possible value. This resulted in inefficient and wrong moves initially. Furthermore, implementing time class was an issue as I couldn’t figure out how to call the class and check implementation within 5 seconds. After being able to do that I was able to see the time and remove repeated code making it more efficient. The code works within limit to a depth of 9.



**Test Cases**

**Testing board**

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doBoardTests()

{

Board b(3, 2);

assert(b.holes() == 3 && b.totalBeans() == 12 &&

b.beans(SOUTH, POT) == 0 && b.beansInPlay(SOUTH) == 6);

b.setBeans(SOUTH, 1, 1);

b.moveToPot(SOUTH, 2, SOUTH);

assert(b.totalBeans() == 11 && b.beans(SOUTH, 1) == 1 &&

b.beans(SOUTH, 2) == 0 && b.beans(SOUTH,

POT) == 2 && b.beansInPlay(SOUTH) == 3);

Side es;

int eh;

b.sow(SOUTH, 3, es, eh);

assert(es == NORTH && eh == 3 && b.beans(SOUTH, 3) == 0 &&

b.beans(NORTH, 3) == 3 && b.beans(SOUTH, POT) == 3 && b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 7);

}

**Testing player**

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doPlayerTests()

{

HumanPlayer hp("Marge");

assert(hp.name() == "Marge" && hp.isInteractive());

BadPlayer bp("Homer");

assert(bp.name() == "Homer" && !bp.isInteractive());

SmartPlayer sp("Lisa");

assert(sp.name() == "Lisa" && !sp.isInteractive());

board b(3, 2);

b.setBeans(SOUTH, 2, 0);

cout << "=========" << endl;

int n = hp.chooseMove(b, SOUTH);

cout << "=========" << endl;

assert(n == 1 || n == 3);

n = bp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3);

n = sp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3);

}

**Testing Game**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doGameTests()

{

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(3, 0);

b.setBeans(SOUTH, 1, 2);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 2);

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

bool over;

bool hasWinner;

Side winner;

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 0 && g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 2 && g.beans(SOUTH, 1) == 2 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 && g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 1 && g.beans(NORTH, 3) == 0 && g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 && g.beans(NORTH, 1) == 1 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 && g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 1 && g.beans(SOUTH, 3) == 0);

g.move(SOUTH);

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(SOUTH, POT) == 3 && 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) == 1);

g.move(NORTH);

g.status(over, hasWinner, winner);

assert(over && g.beans(NORTH, POT) == 1 && g.beans(SOUTH, POT) == 4 && g.beans(NORTH, 1) == 0 && g.beans(NORTH, 2) == 0 && g.beans(NORTH, 3) == 0 && g.beans(SOUTH, 1) == 0 && g.beans(SOUTH, 2) == 0 && g.beans(SOUTH, 3) == 0);

assert(hasWinner && winner == SOUTH);

}

**Testing Board**

int nHoles = 6;

int nInitialBeansPerHole = 4;

Board a(nHoles, nInitialBeansPerHole);

assert(a.beansInPlay(NORTH) == 4\*6);

assert(a.beansInPlay(SOUTH) == 4\*6);

assert(a.totalBeans() == 4\*6\*2);

assert(a.setBeans(NORTH, 3, 0) == 1);

assert(a.setBeans(NORTH, -1, 0) == 0);

assert(a.setBeans(NORTH, 7, 0) == 0);

assert(a.totalBeans() == 4\*6\*2-4);

assert(a.beansInPlay(NORTH) == 4\*5);

assert(a.beansInPlay(SOUTH) == 4\*6);

assert(a.setBeans(NORTH, 3, 4) == 1);

Side endSide = SOUTH;

int endHole = 999;

assert(a.sow(NORTH, 3, endSide, endHole) == 1

&& endSide == SOUTH

&& endHole == 1);

assert(a.totalBeans() == 4\*6\*2);

assert(a.beansInPlay(NORTH) == 4\*6-2);

assert(a.beansInPlay(SOUTH) == 4\*6+1);

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

assert(a.totalBeans() == 4\*6\*2);

Board board(6, 3);

//test beans

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

assert(board.beans(NORTH, 1) == 3);

assert(board.beans(NORTH, 2) == 3);

assert(board.beans(NORTH, 3) == 3);

assert(board.beans(NORTH, 4) == 3);

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

assert(board.beans(NORTH, 6) == 3);

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

assert(board.beans(NORTH, -1) == -1);

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

assert(board.beans(SOUTH, 1) == 3);

assert(board.beans(SOUTH, 2) == 3);

assert(board.beans(SOUTH, 3) == 3);

assert(board.beans(SOUTH, 4) == 3);

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

assert(board.beans(SOUTH, 6) == 3);

assert(board.beans(SOUTH, 7) == -1);

assert(board.beans(SOUTH, 8) == -1);

assert(board.beans(SOUTH, 8) == -1);

//test totalbeans and set and beansInPlay

assert(board.beansInPlay(NORTH) == 18);

assert(board.beansInPlay(SOUTH) == 18);

assert(board.totalBeans() == 2\*18);

assert(board.setBeans(NORTH, 1, 20));

assert(board.beansInPlay(NORTH) == 18 - 3 + 20);

assert(board.beansInPlay(SOUTH) == 18);

assert(!board.setBeans(NORTH, -1, 20));

assert(!board.setBeans(NORTH, 9, 20));

assert(board.beans(NORTH, 1) == 20);

assert(board.totalBeans() == 18 \* 2 - 3 + 20);

assert(board.setBeans(NORTH, 0, 20));

assert(board.totalBeans() == 18\*2 - 3 + 20 \* 2);

//test movetopot

assert(board.moveToPot(NORTH, 2, NORTH));

assert(board.beans(NORTH, 0) == 23);

assert(board.moveToPot(NORTH, 1, NORTH));

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

assert(board.moveToPot(NORTH, 2, SOUTH));

assert(board.moveToPot(SOUTH, 2, SOUTH));

assert(!board.moveToPot(SOUTH, 0, SOUTH));

**Testing 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("BadComputer");

assert(!bad.isInteractive());

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

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);

**Testing smart player**

BadPlayer bp("baddy");

SmartPlayer sp("smarty");

Board b(4,4);

Game g(b,&bp,&sp);

g.play();

// worked as intended