The design of my classes:

Board:

I used vector to store the beans. Besides the number of holes, I used another private member variable to store the size of the vector. Also, I wrote two map functions to map the index of the vector and the hole.

Player:

Human Player:

Besides changing the isInteractive to return false and allow player to enter the choice, nothing is much different from the base class.

Bad Player:

It always chooses the first not empty hole.

Smart Player:

The design is mainly around the chooseMove function, which will be discussed in the next section.

Game:

Besides what’s given on the spec, I defined a Boolean value isSouthTurn to determine the player’s turn.

Design for SmartPlayer::chooseMove()

I defined three private methods. The first one is choose(), which is the recursing function written based on the given pseudocode. The second one is evaluate(), this function specifically evaluates whether a move is good or bad base one the difference between pots. The third one is whichSide(). This function determines if we are evaluating things from our side or from the opponent side so that we can apply the minimax algorithm.

I also defined a private variable depth to keep track of the number of recursion.

Pseudocode

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

{

Turn(s,hole) into the index of the member vector

If the hole is invalid or a pot

Return false;

Int beansLeft = the beans in (s,hole);

Set beans in (s,hole) to 0

int currentHole = index + 1;

base one s, decide which pot to skip

while(beansLeft > 0){

if currentHole do not need to be skiped:

put one bean in position currentHole;

increment currentHole;

if currentHole is out of the range of index, set it to zero}

currentHole -- is the index of the end hole;

map the index to get endHole and endSide;

return true;

}

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

{

Map (s,hole) to get the index of the vector;

Int num = beans in (s,hole)

Set the beans in (s,hole) to be zero;

Map pot to get the index of pot

Set the beans in (potOwner, pot) to num

Return true;

}

int Board::map(Side s, int holes) const // give holes, return index

{

If the hole is invalid

Return -1;

If s is SOUTH:

If holes is 0

Return 1+the number of holes in one side;

Return holes;

If s is NORTH:

If hole is 0:

Return 0;

Return( total number of holes(including pots) in the whole board – hole);

}

void Board::map(int index, Side& s, int& hole) const //give index, return holes

{

If index is invalid:

Return;

If index is no bigger than (the number of holes on one side + 1) and greater than 0

S is SOUTH

If index is not (the number of holes on one side + 1)

Index is equal to holes;

Else

Hole is 0;

If index is larger than (the number of holes on one side + 1) or is equal to 0:

S is NORTH;

If index is not 0

Hole = total number of holes in the board – index

Else

Hole is 0

}

void SmartPlayer::choose(Side s, const Board& b, int& bestHole, int& difference, const int& depth, AlarmClock& ac) const

{

if either north or south has 0 beans in play,

bestHole = -1

value = value of this position

return

if the depth is larger than the set maximum value or time is up

bestHole = -1

value = value of this position

return

copy board b to temp

for every hole h the player can choose

"make" the move in temp

while end up in the pot and get an extra round

choose(s,temp,nextBesthole,nextDifference,depth,ac)

if there is no more beans to sow

break;

if end up in a hole that was empty just before and the corresponding opponent hole is not empty(a capture):

move the beans in this hole h and the opponent’s beans in h to the my pot

value = value of this position

if value is a winning value

this hole h is the bestHole;

return;

chooseMove(opponent, temp, ,nextBesthole,nextDifference,depth++,ac )

// recurse for the opponent and increment depth

Unmake the move by setting temp = b;

nextDifference is better for the player than best seen so far,

bestHole = h

value = nextDifference

return

}

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

{

If either side has 0 beans in play

Compare the beans in play in s and in the opponent side

If in s is bigger, return 99\*whichSide // whichSide returns 1 if s is the smartplayer’s side and -1 if s is the opponent’s side

If opponent is bigger, return -99\*whichSide;

If they are the same, return 0;

Return (the beans in s pot– beans in s’s opponent’s pot);

}

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

{

If either side has 0 beans in play

Over = true;

If north has different beans than south

hasWinner is true;

if north has more beans

winner is north, return

if north has fewer beans

winner is south, return

over is false, return;

}

Bool Game::move()

{

Do

{

If either side has 0 beans

Move all the beans to the pots

Return false;

If is south’s turn

South player choose move

Sow the hole chosen by south

Else if is north’s turn

North player choose move

Sow the hole chosen by north

}while endHole is 0;

If there is a capture

Move the beans in the current hole and the opponent’s corresponding hole to the pot

If either side has 0 beans

Move all the beans into pots

Set turns to the other player

Return true

}

Void Game::play()

{

Check the status of the game

While game is not over

Move

Check the status of the game

If there is a winner

Cout the winner’s name

Else

Cout it’s a tie

}

Test Cases

//testing constructor of board

Board test(6, 4);

Board bad(-1, -1);

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

// testing holes

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

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

//testing beans()

for (int i = 7; i < 100; i++)

{

assert(test.beans(SOUTH, i) == -1);

assert(test.beans(NORTH, i) == -1);

}

assert(test.beans(SOUTH, -1) == -1);

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

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

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

assert(test.beans(NORTH, 1) == 4);

Side endSide = NORTH;

int endHole = 0;

test.sow(NORTH, 1, endSide, endHole);

assert(endSide == SOUTH);

assert(endHole == 3);

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

assert(test.beans(SOUTH, 1) == 5);

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

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

test.sow(SOUTH, 5, endSide, endHole);

assert(test.beans(SOUTH, POT)==1);

assert(endSide == NORTH);

assert(endHole == 5);

assert(test.moveToPot(NORTH, 4, SOUTH));

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

assert(test.beans(SOUTH, POT) == 5);

assert(test.setBeans(SOUTH, POT, 3));

assert(test.beans(SOUTH, POT) == 3);

Board test2(3, 2);

assert(!test2.sow(SOUTH, POT, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 5);

assert(!test2.sow(NORTH, POT, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 5);

for (int i = 4; i < 100; i++)

{

assert(!test2.sow(SOUTH, POT, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 5);

assert(!test2.sow(NORTH, POT, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 5);

}

assert(test2.sow(SOUTH, 1, endSide, endHole));

assert(endSide == SOUTH);

assert(endHole == 3);

assert(test2.sow(NORTH, 1, endSide, endHole));

assert(endSide == SOUTH);

assert(endHole == 1);

assert(test2.sow(SOUTH, 2, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 3);

assert(test2.sow(NORTH, 2, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 0);

Board test3(3, 10);

assert(test3.sow(SOUTH, 1, endSide, endHole));

assert(endSide == SOUTH);

assert(endHole == 0);

assert(test3.beans(SOUTH, 2) == 12);

assert(test3.beans(SOUTH, 3) == 12);

assert(test3.beans(endSide, endHole) == 2);

assert(test3.beans(NORTH, 0) == 0);

assert(test3.beans(NORTH, 1) == 11);

assert(test3.beans(NORTH, 2) == 11);

assert(test3.beans(NORTH, 3) == 11);

//test moveToPot

assert(!test3.moveToPot(NORTH, 0, SOUTH));

assert(!test3.moveToPot(NORTH, 0, NORTH));

assert(!test3.moveToPot(SOUTH, 0, NORTH));

assert(!test3.moveToPot(SOUTH, 0, SOUTH));

assert(test3.beans(SOUTH, 2) == 12);

assert(test3.beans(SOUTH, 3) == 12);

assert(test3.beans(endSide, endHole) == 2);

assert(test3.beans(NORTH, 0) == 0);

assert(test3.beans(NORTH, 1) == 11);

assert(test3.beans(NORTH, 2) == 11);

assert(test3.beans(NORTH, 3) == 11);

assert(test3.beans(SOUTH, 1) == 1);

assert(test3.moveToPot(NORTH, 1, NORTH));

assert(test3.beans(NORTH, 1) == 0);

assert(test3.beans(NORTH, 0) == 11);

assert(!test3.moveToPot(NORTH, -1, SOUTH));

assert(!test3.moveToPot(NORTH, -1, NORTH));

assert(!test3.moveToPot(SOUTH, -1, NORTH));

assert(!test3.moveToPot(SOUTH, -1, SOUTH));

assert(test3.beans(NORTH, 1) == 0);

assert(test3.beans(NORTH, 0) == 11);

assert(test3.beans(NORTH, 2) == 11);

assert(test3.beans(NORTH, 3) == 11);

assert(test3.beans(SOUTH, 2) == 12);

assert(test3.beans(SOUTH, 3) == 12);

assert(test3.beans(SOUTH, 1) == 1);

for (int i = 4; i < 50; i++)

{

assert(!test3.moveToPot(NORTH, i, SOUTH));

assert(!test3.moveToPot(NORTH, i, NORTH));

assert(!test3.moveToPot(SOUTH, i, NORTH));

assert(!test3.moveToPot(SOUTH, i, SOUTH));

assert(test3.beans(NORTH, 1) == 0);

assert(test3.beans(NORTH, 0) == 11);

assert(test3.beans(NORTH, 2) == 11);

assert(test3.beans(NORTH, 3) == 11);

assert(test3.beans(SOUTH, 2) == 12);

assert(test3.beans(SOUTH, 3) == 12);

assert(test3.beans(SOUTH, 1) == 1);

assert(test3.beans(SOUTH, 0) == 2);

}

//test setBeans

for (int i = -1; i < 100; i++)

{

assert(!test3.setBeans(SOUTH, i, 1));

assert(!test3.setBeans(NORTH, i, 1));

if (i == -1)

{

i = 3;

}

}

assert(test3.beans(NORTH, 1) == 0);

assert(test3.beans(NORTH, 0) == 11);

assert(test3.beans(NORTH, 2) == 11);

assert(test3.beans(NORTH, 3) == 11);

assert(test3.beans(SOUTH, 2) == 12);

assert(test3.beans(SOUTH, 3) == 12);

assert(test3.beans(SOUTH, 1) == 1);

assert(test3.beans(SOUTH, 0) == 2);

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

{

assert(!test3.setBeans(SOUTH, i, -1));

assert(!test3.setBeans(NORTH, i, -1));

}

assert(test3.beans(NORTH, 1) == 0);

assert(test3.beans(NORTH, 0) == 11);

assert(test3.beans(NORTH, 2) == 11);

assert(test3.beans(NORTH, 3) == 11);

assert(test3.beans(SOUTH, 2) == 12);

assert(test3.beans(SOUTH, 3) == 12);

assert(test3.beans(SOUTH, 1) == 1);

assert(test3.beans(SOUTH, 0) == 2);

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

{

assert(test3.setBeans(SOUTH, i, 2));

assert(test3.setBeans(NORTH, i, 2));

}

SmartPlayer s("Smart");

HumanPlayer h("Human");

BadPlayer b("Bad");

//test name and isinteractive;

assert(s.name() == "Smart");

assert(h.name() == "Human");

assert(b.name() == "Bad");

assert(!s.isInteractive());

assert(!b.isInteractive());

assert(h.isInteractive());

/\*SmartPlayer a("Adam");

SmartPlayer b("Bob");

Board board(6, 4);\*/

/\*board.setBeans(NORTH, 0 ,9);

board.setBeans(NORTH, 1, 1);

board.setBeans(NORTH, 2, 1);

board.setBeans(NORTH, 3, 50);

board.setBeans(NORTH, 4, 4);

board.setBeans(NORTH, 5, 3);

board.setBeans(SOUTH, 0, 10);

board.setBeans(SOUTH, 1, 4);

board.setBeans(SOUTH, 2, 50);

board.setBeans(SOUTH, 3, 50);

board.setBeans(SOUTH, 4, 2);

board.setBeans(SOUTH, 5, 0);\*/

Board New(6, 10);

while (New.beansInPlay(SOUTH) != 0)

{

int hole = b.chooseMove(New, SOUTH);

assert(hole > 0 && hole < 7);

New.sow(SOUTH, hole, endSide, endHole);

}

New = New2;

while (New.beansInPlay(NORTH) != 0)

{

int hole = s.chooseMove(New, NORTH);

assert(hole > 0 && hole < 7);

New.sow(NORTH, hole, endSide, endHole);

}

//test Game

//test status

Game a(New, &b, &s);

bool over, haswinner;

Side winner;

a.status(over, haswinner, winner);

assert(over);

assert(haswinner);

assert(SOUTH);

Game x(bad, &s, &b);

x.status(over, haswinner, winner);

assert(over);

assert(!haswinner);

//test move

Board y(3, 2);

y.setBeans(SOUTH, 1, 1);

y.setBeans(NORTH, 3, 1);

// 2 2 1

// 0 0

// 1 2 2

Game k(y, &h, &b);

k.move();

// 2 0 1

// 0 4

// 0 0 3

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

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

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

assert(k.beans(SOUTH, POT) == 4);

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

//test play

//k.play();

//k.status(over, haswinner, winner);

////choose 3 and 1

//assert(over);

//assert(!haswinner);

// if choose 1 then 3, human wins