1.

Board: I use vector to store data. I create two vector arrays(for South and for North) with the size of the the holes not including the pot(I use two integer values to represent the beans in the pot). I also use two integer values to keep track of hole numbers and total beans.

Player: In addition to what’s required, I declared another private member helper function(Evaluation) in class SmartPlayer, which will be explained in details later in the pseudocode. I implemented the chooseMove() function separately for each derived class. I also implemented isInteractive() separately in humanPlayer class.

Game: The private members contain one board object, two player pointers and a side object(to record what current turn is).

2.

I implemented an additional private member function *Evaluation(const Board& b, Side s, int& thehole, int times) const* to help. In this function, I explored 6 depths into the current game(the tree) and made it returns a measurement corresponding with each choice. I use value(difference in beans between South and North pot (South minus North) ) to measure how good each choice is. In the chooseMove function, I simply returned the thehole parameter in the Evaluation function. The details about the specific structure of this function are represented in the pseudocode.

3.

Board::Sow

Check if the input hole number is valid, if not then return false

Check for side, south or north, and check for positive beans number

if the side is south

divide beans by all holes+1 to know how many rounds the beans should be added to each hole(in case the bean number is really large)

each hole plus bean number that equals rounds including the south pot

calculate the remaining beans need to be added after these rounds

check if the beans end on the south side

if yes, add the beans, set endside and endhole, return

if not, add the beans to the south side first

check if beans end in the pot

if yes, if yes, add the beans, set endside and endhole, return

if not, check if beans end on the north side

if yes, add the beans, set endside and endhole, return

if not, add the beans to the north side first, add the remaining beans to south side, set endside and endhole, return

if the side is north

do the same thing, switching the sides

SmartPlayer::Evaluation (returns an int)

Make a copy of board to make change

- Check if some side is going to win(the beans number in one pot is greater than half of the total beans)(this step saves time)

if yes, return positive and negative infinity depending on the winner (positive if south wins and negative if north wins)

- Check if the game ends

if ends, set value to beans in South pot minus beans in the North pot (the difference in beans), return positive and negative infinity depending on the winner

- Check if depth>=6 (to limit the time)

if it does, set value to the difference(South-North) between pots, return value

Loop to find holes that contains beans(>0), for each of such hole

use sow function to move the beans in that hole

check for special cases

if capture

move beans to pot accordingly

if last bean ends up in pot

call Evaluation function again without changing the time(depth)

unmake the move

use map to record the return value of the previous Evaluation function and position(loop parameter)

Call Evaluation function again with (times+1)

unmake the move

use map to record the return value of the previous Evaluation function

To find the “best” measurement in the map according to the side

if side is South

find the largest difference (should be positive)

if side is North

find the smallest difference (should be negative)

Return value(the difference)

SmartPlayer::chooseMove

Call evaluation function

Return the hole

BadPlayer::chooseMove

loop to find the first none zero hole on their side\

return the position of that hole

Game::move

check for status

if game not over

check for which side’s turn

call the sow function to move beans according to play’s choice

check if it ends in this side’s pot

if yes, check if game is over

if not, display, then return move function it self (recursion)

check if there is a capture (the endhole is empty in the old board or the endhole is the same hole player chose) (endhole shouldn’t be pot)

if yes, move beans to pot accordingly

set turn to the opposite side

return true

if game is over

move all the beans to the pot according to the side

return false

Game::play

Check for status

while game not over

display

enter to continue

move

check for status

if game over

move (in order to collect all remaining beans into the pots)

display and announce the winners

4.

I had trouble dealing with the capture situation. Originally, I wrote my codes to test if the beans in the endHole minus one would be zero. Then I realized that the beans can be added to the same hole more than once if there are enough beans. I have to create a board copy to compare if the hole is originally empty. Then I realized that I have to consider the situation the beans ended up in the same hole player chose. So I kept adding condition to it and the function itself becomes really messy. I haven’t found a better way to do it through.

I also had a hard time writing the SmartPlayer::chooseMove function. It’s hard to think through the recursion. I also didn’t know how to end the function(where to return) based on the time. After a few discussion with my friends, we decide to set the depth to six. Also it’s hard to compare and assign the value(the difference) to each position(node). We did research to use map to record the value with each node(position), and that is more convenient.

5.

///// to test Board

//constructor and basic setters and getters

Board b = Board(5,1);

assert(b.totalBeans()==10);

b.setBeans(SOUTH, 1, 1);

b.setBeans(SOUTH, 2, 2);

b.setBeans(SOUTH, 3, 3);

b.setBeans(SOUTH, 4, 4);

b.setBeans(SOUTH, 5, 18);

assert(b.totalBeans()==33);

Side one;

int two;

// function sow - end up one same side

// - on different side

// - multiple rounds

b.sow(SOUTH, 1, one, two);

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

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

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

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

//0334

b.sow(SOUTH, 2, one, two);

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

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

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

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

assert(b.beans(SOUTH, 5)==19);

// 004519

b.sow(SOUTH, 5, one, two);

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

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

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

assert(b.beans(SOUTH, 4)==6);

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

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

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

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

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

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

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

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

b.setBeans(NORTH, 5, 1);

b.setBeans(NORTH, 4, 2);

b.setBeans(NORTH, 3, 3);

b.setBeans(NORTH, 2, 4);

b.setBeans(NORTH, 1, 18);

assert(b.totalBeans()==46);

assert(b.beansInPlay(NORTH)==28);

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

b.sow(NORTH, 1, one, two);

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

assert(b.beans(SOUTH, 2)==4);

assert(b.beans(SOUTH, 3)==7);

assert(b.beans(SOUTH, 4)==8);

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

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

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

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

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

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

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

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

Board b(3, 2); //construct board

/////////// for Player

HumanPlayer hp("hp");

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

BadPlayer bp("bp");

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

SmartPlayer sp("sp");

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

Board b(3, 2);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 0);

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

assert(n == 1 );

n = bp.chooseMove(b, SOUTH);

assert(n == 1);

n = sp.chooseMove(b, SOUTH);

assert(n == 1);

/////////// for Game

Board b1(6, 2);

Game g=Game(b1, &sp, &bp);

g.play();

// play contains multiple other game functions and they can be tested through player

//// play the game, smartplayer should win

//// the first two moves for smartplayer should be 5, 3

// other game tests

BadPlayer bp1("bp12");

BadPlayer bp2("bp2");

Board b(2, 0);

b.setBeans(SOUTH, 1, 1);

b.setBeans(NORTH, 1, 5);

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) == 5 && g.beans(NORTH, 2) == 0 &&

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

//

g.move();

g.status(over, hasWinner, winner);

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

g.beans(NORTH, 1) == 5 && g.beans(NORTH, 2) == 0 &&

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

g.move();

g.status(over, hasWinner, winner);

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

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

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

g.move();

g.status(over, hasWinner, winner);

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

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

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

g.move();

assert(over && g.beans(NORTH, POT) == 5 && g.beans(SOUTH, POT) == 1 &&

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

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

assert(hasWinner && winner == NORTH);

cout<<"YESSS"<<endl;