a. I used vector as the data structure to store the number of beans in each pot, since it have O(1) as the Big O for random access. I created three helper functions. One is for SmartPlayer::chooseMove, the reason is explained below. Another one is called sstatus, for the purpose of checking the current status of the game. The third one is called cmove, used to make a complete move.

b. description for SmartPlayer::chooseMove: the recursion stops under the following circumstances: the game is over; it reaches a certain depth that I set up (6, 7, or 10); the difference between the number of beans in two pots is greater than half of the total number of beans at the beginning of each game; there is only one available choice for the SmartPlayer.

I build a function called tra to do the recursion instead of using chooseMove because the parameters of chooseMove is not enough to gather information during each recursion.

I used a map in tra function. The key for this map is the difference between the number of beans in both pots, the value is the hole number that if a SmartPlayer choose, it will cause that difference. If the side is NORTH, we pick the first item in the map; if the side is SOUTH, we pick the last item in the map, since map is in ascending order.

During each call to tra, it will make a not-so-complete move. If a move causes an additional move, the cmove function (a function I created to make a complete move) will return 1, telling the tra function that it needs another decision for the current Side. Therefore, tra will call itself without changing the Side nor depth. However, all the captures are done after cmove returns.

c. pseudo codes:

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

If hole number is invalid, return false;

If there is nothing in that hole, return false;

Pick up all the beans in that hole; set the number of beans there to zero

Depending on which Side s is, do the following:

While there are still beans remaining in our hand:

Check if we reach the NORTH or SOUTH pot, if we do, then check out Side to decide whether to put a bean in it or skip it, and then change s to Side.

If we do not, then put the bean inside that hole and go counterclockwise in the Board.

Set endSide to s, endHole to the ending hole.

bool Game::move()

for every hole in both NORTH and SOUTH side, check if there are still possible moves.

If on one side, all holes are empty, return false.

Use cmove function to make a complete move.

Give turn to the opponent, return true.

void Game::play()

use status function to check the current status of this game.

Display the Board.

While the game does not stop

Wait for the user to press enter to continue printing and moving.

Call the move function

Call the display function

Check the status of the game

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

While the user gives illegal input

Get what the user enters

If the user did not enter anything, continue;

If the user did not enter a digit, continue;

If the hole the user enters has no beans or it’s a pot or it’s negative, continue;

Else, return the number the user enters.

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

Classify the current board as small, medium, or large depending on total non-empty holes and number of beans that are in play.

If the board is large, we construct a game tree of depth 6.

If the board is medium, we construct a game tree of depth 7

If the board is large, we construct a game tree of depth 10

Return the hole number given by the game tree, constructed by tra function.

Int SmartPlayer::tra(const Board& b, Side s, int d, int& r) const

Create a copy of the board, used to simulate movement.

Create a map, used to map the “value” of each choice to chosen hole number. It will sort its item in an ascending order.

Use the sstatuts helper function to check current status of the board.

If it is over or depth is zero

return 10000000 (a very big value) if the winner is SOUTH, -10000000 (a very small value) if the winner is NORTH; else, return the difference between number of beans of both pots.

If the difference between number of beans of two pots is greater than half of the total number of beans at the beginning of each game (which means the player wins or loses the game no matter how it chooses the hole)

Set the hole number, passed in by reference, to the one with smallest number.

Return the big value if the winner is SOUTH, the small value if the winner is NORTH

For each available hole:

If there is only one choice, set the hole number to that choice, return the difference between the number of beans in two pots,

Else, call the cmove function.

If that function returns 1,

call tra function with the same Side, the same depth.

Store the returned value together with the selected hole into map

Restore the board

Else

Call tra function with different Side, with a depth minus one from the original

Store the returned value together with the selected hole into map

Restore the board

If the side is SOUTH, set the hole number to its second value in the end; return its first value in the end.

If the side is NORTH, set the hole number to its second value in the end; return its first value in the first.

Void sstatus (const Board& b,bool& over, bool& hasWinner, Side& winner)

For each hole in NORTH and SOUTH side

If there is non empty hole in both side

Set over to false;

Return

Set over to true

For each hole in NORTH and SOUTH side

If the current hole is non-empty, add the number of beans in it to total number of beans of its corresponding side

If the number of beans in pot add with total number of beans in holes is the same for both side

Set hasWinner to false;

Return

Else

Set winner to the side with greater number of beans,

Set hasWinner to true

Return

Int cmove(Game\* g,Board& b, Side s,const Player\* p,int d)

Create two dynamically allocated array , used to check capture

While we have to make a movement

Set each item in those two arrays to the number of beans in corresponding side

If the function is called by tra function and the difference between number of beans in two pots is greater than half of the total number of beans at the beginning of each game

Return 0;

If the function is called by tra function

Use the value passed in to be the hole number

Else

Call player’s chooseMove function

Call Board’s sow function

If the player satisfies the condition for an additional move

If the function is called by tra function

Delete the dynamic array

Return 1

Else

Tell the player that an additional move is made

Else

Set the Boolean value used for additional move to false

If the player satisfies the condition for capture

Tell the player that a capture is made

Move beans to pot according the rule

Call sstatus function to check the status of this game

If the game is over

Move the remaining beans into their corresponding pot

Delete the dynamic arrays.

d. serious inefficiencies: the depth can only be 6 when it is a big board. If we can store each possibilities in a tree, then we need not to make a decision every time, just follow that tree will be enough.

Notable problems: figuring out how to use recursion without creating an actual game tree to store my decisions.

e. test cases:

int main()

{

/\*HumanPlayer hp("Marge");

SmartPlayer bp("Homer");

Board b(6, 2);

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

g.play();\*/

Board b(1, 1);

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

Board b1(0, 0);

assert(b1.holes() == 1);

assert(b1.beans(SOUTH, 1) == 0);

Board b2(-1, -1);

assert(b2.holes() == 1);

assert(b2.beans(NORTH, 1) == 0);

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

assert(b.beans(NORTH, 2) == -1);

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

Board b3(4, 10);

// 10 10 10 10

// 0 0

// 10 10 10 10

assert(b3.beansInPlay(NORTH) == b3.beansInPlay(SOUTH) && b3.beansInPlay(NORTH) == 40);

assert(b.totalBeans() == 2);

assert(b1.totalBeans() == 0);

assert(b2.totalBeans() == 0);

int eh = 0;

Side es = NORTH;

assert(!b.sow(NORTH, 0, es, eh));

assert(b.sow(NORTH, 1, es, eh) && eh == 0 && es == NORTH);

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

assert(!b1.sow(NORTH, 1, es, eh));

assert(!b2.sow(NORTH, 1, es, eh));

assert(!b3.sow(NORTH, 5, es, eh));

assert(!b1.sow(NORTH, -1, es, eh));

// 11 11 12 1

// 1 0

// 11 11 11 11

assert(b3.sow(NORTH, 4, es, eh) && es == NORTH && eh == 3);

// 12 12 13 3

//1 2

// 12 12 12 1

assert(b3.sow(SOUTH, 4, es, eh) && es == NORTH && eh == 4);

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

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

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

assert(b3.moveToPot(SOUTH, 3, NORTH) && b3.beans(NORTH, 0) == 13 && b3.beans(SOUTH, 3) == 0);

assert(!b3.moveToPot(SOUTH, -1, SOUTH));

// 12 12 13 3

//13 2

// 12 12 0 1

assert(!b.setBeans(NORTH, 2, 10));

assert(b.setBeans(NORTH, 1, 10) && b.beans(NORTH, 1) == 10);

assert(!b.setBeans(NORTH, -1, 0));

assert(!b.setBeans(SOUTH, 0, -1));

assert(b3.setBeans(NORTH, 0, 2) && b3.beans(NORTH, 0) == 2);

// 12 12 13 3

//2 2

// 12 12 0 1

HumanPlayer h("a");

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

assert(h.isInteractive());

b = Board(2, 3);

b.setBeans(SOUTH, 1, 0);

// 3 3

//0 0

// 0 3

cout << "only 2 works" << endl;

assert(h.chooseMove(b, SOUTH) == 2);

BadPlayer h1("b");

assert(h1.name() == "b");

assert(!h1.isInteractive());

assert(h1.chooseMove(b, SOUTH) == 2 );

b.setBeans(SOUTH, 2, 4);

Game gg(b, &h1, &h);

bool isover = false;

bool hw = false;

Side s = NORTH;

gg.status(isover, hw, s);

assert(!isover && !hw && s == NORTH);

assert(gg.move());

gg.status(isover, hw, s);

assert(!gg.move() && isover&&hw &&s == SOUTH && gg.beans(SOUTH,0) == 6 && gg.beans(NORTH, 0) == 4);

// 0 0

//4 6

// 0 0

Board b4(1, 1);

gg = Game(b4, &h1, &h);

assert(gg.move());

gg.status(isover, hw, s);

assert(!gg.move() && isover && !hw&&s == SOUTH);

assert(gg.beans(NORTH, 2) == -1);

}