Morgan Lim

105 168 668

a) To keep track of the holes and beans in Board, I used two dynamically allocated arrays, one for the north player, and one for the south player. There are several helper functions that I created in Player, Game, and Board. I created these helper functions to make the code easier to read and less messy. For example, in sow(), the function is really long and I found that while I was writing the function, I would often look back and forth between the beginning and middle of the if statement that I was writing to see what side I was writing for. As I was check my code, it was confusing to read NORTH and SOUTH, and I was often unsure of what a section of code was meant to be doing. By using the functions determineOppSide() and switchSide() my code was cleaner and easier to read. Additionally, I found that myself rewriting a lot of code, that was tweaked just a bit every time, so it seemed more efficient and less cluttered to just create function for the repeating bits of code.

b) For every hole that is not empty is chooseMove, the function creates a temporary Board, makes a move at the hole with the temporary board, and passes this board to miniMax. miniMax is called recursively, and assumes that the opponent player plays at their best. Then, miniMax returns the how “good” each move is based on the evaluate() function. Evaluate returns 0 if there is a tie, 10000 if the player wins, and -10000 is the players loses. If the game is not over, the board returns a middle value. If the player is winning but hasn’t won (the player has more beans in their pot than the opponent), the board is evaluated at 5000, if it’s the other way around, -5000 is returned.

c) pseudocode

Board.cpp

Board::Board(int nHoles, int nInitialBeansPerHole)

: member initialization for data members

if entered holes < 1 then treat it as if it is 1

if entered initial beans < 0 then treat it as if it is 0

dynamically allocate new memory for each side as an array

set each pot of each side to 0

set hole of each side to initial beans

~Board()

delete each array

Board::Board(const Board &other)

copy data members of other into this instance

create new memory for arrays

copy data into arrays

int beans(Side s, int hole)

if the entered amount of moves is greater than the amount of holes on the board

return -1

otherwise determine the array for the selected side

return the amount of the beans in that hole for that side

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

if the hole is empty, or invalid or is a pot

return false

otherwise

if the picked hole is on the edge of either side

add a bean to that side’s pot

switch sides

move to next hole

otherwise

move to next hole

while there are still beans left

if there is only one bean left

place bean in that hole

(for both north and south sides)

otherwise if the current hole is on the edge and is the player’s side

add one bean to hole

add one bean to the pot

switch sides

otherwise if the current hole is on the edge but on the opponent’s side

add one bean to the hole

switch sides

move hole to first hole of other side

otherwise if the hole is somewhere in the middle

add bean to hole

update the hole and side ended

bool moveToPot(Side s, int hole, Side potOwner)

if the hole is empty or the hole number is not valid

return false

otherwise

find the amount of beans in the opposite hole

add those beans to the potOwner’s pot

clear the hole of the pot owner

bool setBeans(Side s, int hole, int beans)

if the hole is empty of not valid

return false

otherwise

set the number of beans in that array

int\* determineSide(Side s)

return a pointer to the array of the side that was passed to the function

int\* determineOppSide(Side s)

return a pointer to the opposite side of the side that was passed

Game.cpp

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

check if either side is empty

if it is empty

if either side is empty

the game is over

if the game is not over

exit the function without changing anything else

if the game is a tie

there is no winner

exit the function

if there is a winner

set a winner

exit the function

bool move()

if the game is over

clean up the board

return false

otherwise

while the the last bean was placed in the player’s pot and the game is not over

let the player play by sowing

display the board(changes have been made)

if the last bean was placed in an empty hole on the player’s side and the hole across from it is not empty

take the bean and the beans from the opponents hole and move into the pot

display the board (changes have been made)

change the player to the other player

change the side to the other side

void play()

if the game is over

exit the function

otherwise while the game is not over

let the players mopve

check the status of the game

Player\* otherPlayer(Player\* p)

return a pointer to the player that is not player p

Side determineSide(Player\* p)

returns the side that the player p is on

Player\* determinePlayerBySide(Side s)

returns pointer to the player of side s

void cleanUp(bool& over, bool& hasWinner, Side & winner)

for every hole on the board

move any beans in that hole into that side’s pot

check the status of the game

Player.cpp

HumanPlayer

int chooseMove(const Board& b, Side s)

prompt for input until valid input is made

(valid being a a number between 1 and the number of holes on the board)

return input

BadPlayer

int chooseMove(const Board& b, Side s)

find the first non empty hole and return that one

if all holes are empty return -1

SmartPlayer

int chooseMove(const Board& b, Side s)

for every hole that is not empty

make a move at the hole by sowing and making captures as necessary

use a minimax algorithm to determine how “good” of a move that is

if the current move is better than the last move

set the current hole as the hole to pick

if no choice is better than the other

find any non empty hole and return that one

int miniMax()

evaluate the function to how “good” of a state it is in and to see if the game is over

if the game is over or time is out

return the the evaluation of the board

if looking for the maximum

for every hole that is not empty

make a move

use minimax algorithm to determine how good of a move that was

if that move was better for the player than before set best as that move

otherwise if looking for the minimum

for every hole that is not empty

make a move

use minimax algorithm to determine how good that move was

if move was worse for the player (therefore better for the opponent) than before set it as the best move

return best

int evaluate(const Board b, Side s, bool& over)

if the game is over

sweep beans into their respective pots

if the game is a tie

return 0

if the game was lost

return -10000

if the game was won

return 10000

if the game is not yet over return some intermediate value that isnt toal loss or win

if the game is currently a tie

return 0

if the player has more beans in the their pot

return 5000

if the player has fewer beans in the their pot

return -5000

bool isOver(Board b, Side s)

check to to see if either side is empty

if either side is empty

return true

otherwise return false

d) Some bugs and inefficiencies that I found were that SmartPlayer’s chooseMove would sometimes pick holes that were empty, but I could not figure out a way to debug or trace through the function, so instead I just put in a safe guard: I sent the best value as a very random value (-88 )that, if there was no best value found, would not be changed. If control reached the end of chooseMove and bestValue was still -88, the function would simply pick the first hole that is not empty as the move. While this was not the best move, pick a less than optimal move is better than picking an invalid hole that causes a runtime error. Another issue I ran into is that chooseMove for HumanPlayer cannot handle input that is not an integer. Additionally, I found that, even using the alarm clock provided in the FAQ, SmartPlayer’s choose move was consistently taking longer than 5 seconds to choose a move with 4900 as the parameter for the AlarmClock’s parameter. I ended up having to shorten it in order to get it to make a move in a shorter amount of time. One final bug: every so often when I played the game against SmartPlayer, there would be some kind of runtime error where the board would be printed repeatedly until the program was terminated. I wasn’t able to debug or figure out what was causing it because it happened so rarely.

e) test cases

Board b(6, 3);

Side endSide(SOUTH);

int endHole(0);

bool over, hasWinner(true);

HumanPlayer p1("p1");

BadPlayer p2("p2");

SmartPlayer p3("p3");

Player\* pt1 = &p1;

Player\* pt2 = &p2;

Game g(b, pt2, pt1);

//board test cases

//to check holes()

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

//to check that the assigned number of beans for

//the holes and pots are correct

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

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

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

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

//to test that beans() is working correctly

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

assert(b.beans(SOUTH, -3) == -1);

//to test beansInPlay()

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

assert(b.totalBeans() == 36);

//to check that sow is working correctly

assert(b.sow(SOUTH, 6, endSide, endHole) == true);

assert(endHole == 5);

assert(endSide == NORTH);

assert(b.sow(SOUTH, 6, endSide, endHole) == false);

assert(b.beansInPlay(SOUTH) == 15);

assert(b.sow(SOUTH, -1, endSide, endHole) == false);

//to check moveToPot

assert(b.moveToPot(SOUTH, 1, SOUTH) == true);

assert(b.moveToPot(SOUTH, 7, SOUTH) == false);

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

assert(b.beans(SOUTH, 0) == 4);

//to test player

//to test name()

assert(p1.name() == "p1");

assert(p2.name() == "p2");

assert(p3.name() == "p3");

//to test isInteractive

assert(p1.isInteractive() == true);

assert(p2.isInteractive() == false);

assert(p3.isInteractive() == false);

//to test chooseMove

assert(p1.chooseMove(b, SOUTH) != 1 && p1.chooseMove(b, SOUTH) != 6);

assert(p2.chooseMove(b, SOUTH) != 1 && p2.chooseMove(b, SOUTH) != 6);

assert(p3.chooseMove(b, SOUTH) != 1 && p3.chooseMove(b, SOUTH) != 6);

//to test Game

g.display();

g.status(over, hasWinner, endSide);

assert(over == false && hasWinner == true);

//^ hasWinner is true because if the game is not

//over, status() is only supposed to set over to

//false and nothing else

cerr << "passed all tests!";

//tests for 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;

// homer

// 0 1 2

// 0 0

// 2 0 0

// bart

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();

// 0 1 0

// 0 3

// 0 1 0

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();

// 1 0 0

// 0 3

// 0 1 0

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();

// 1 0 0

// 0 3

// 0 0 1

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();

// 0 0 0

// 1 4

// 0 0 0

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);

}

int main()

{

dogametests();

cout << "passed all tests" << endl;

}

//tests for 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);

}

int main()

{

doPlayerTests();

cout << "Passed all tests" << endl;

}

//tests for 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);

}

int main()

{

doBoardTests();

cout << "Passed all tests" << endl;

}