Madeleine Murphy

004991129

Project 3 Report

Due May 21, 2019

**Description of classes**

**Board**

Board.cpp uses two dynamically allocated arrays, m\_north and m\_south, to store the information about the beans in each hole on each side. It uses int variables for the number of beans in North’s pot and in south’s pot. I did not call any helper functions in Board.cpp, however I did add appropriate destructor, copy constructor, and assignment operator functions for housekeeping because I use dynamic memory. The rest of the private member functions just store integer information about the board, such as total number of holes, total number of beans, etc.

**Player**

The only private member variable of Player.h is m\_name, which stores the name of the player. HumanPlayer, BadPlayer, and SmartPlayer are all public subclasses of Player. They all have a constructor which just initializes the Player constructor, isInteractive, and chooseMove. Only SmartPlayer has helper functions, which are chooseMoveHelper, evaluate, and notSide. ChooseMoveHelper and evaluate take in more parameters which give them helpful information about how to pick a smart move.

**Game**

Game uses a Board m\_board and pointer to both players, north and south. In its constructor it copies the board passed in to m\_board and plays on it. I also use a pointer to a player m\_whoseTurn to keep track of whose current turn it is. I also have member variables m\_over, m\_hasWinner, m\_winner for the status function. I have 4 helper functions, turnSide which takes a player and returns the side it is, notSide which takes a player and returns the opposite side, playerside which takes in a side and returns a pointer to a player, and m\_play which is the helper for m\_play that takes in extra parameters so that we can periodically ask the user to press ENTER in the case that both players are not interactive.

**SmartPlayer::chooseMove**

SmartPlayer::chooseMove first calls chooseMoveHelper, which takes in extra parameters of bestHole and value. It will ultimately return the bestHole choice back to chooseMove. In chooseMoveHelper, we first make a copy of the board, and also keep track of the original board. Then check if there are no more moves to make, and if there are no more moves return -1. Then, we go through every possible move for the player and evaluate that move. We keep track of what evaluate returns for each move, and which ever hole returns the maximum, we choose that hole.

Evaluate is where most of the implementation actually takes place. Evaluate takes in a depth, hole, board, side of what move is being made, the value, and the side of the original player who is making the move from chooseMove. Both sides are required because we are using recursion back and forth between the moves. The overall algorithm of evaluate is: if depth is not 0 and the hole we inputed gives a valid move, sow that hole. If they land in the pot, evaluate every possible next move the player can take. If they make a capture, make that capture. If the game is over, sweep the holes and compare the pots to determine a winner. Return 1000, -1000, or 0 depending on who won/tied. If the game is still in play, use the heuristic of the difference between my pot and my opponents pot to measure the value. If we just made a turn, the next move is the opponents. The opponent wants to minimize the next value, so it’s going to run through all of its possible moves and find the minimum of those values and return that minimum. If it was just the opponents turn, we get to go, and we want to run through all of our possible moves and evaluate them and return the maximum of all of our possible moves. Thus, evaluate recursively calls itself whenever a player gets to make another move.

**Pseudocode**

**Board.cpp**

Board::Board(int nHoles, int nInitialBeansPerHole)

If nHoles not positive, treat as 1

If initial Beans not positive, treat as 0

Create new north array with nHoles

Create new south array with nHoles

Set total holes to nHoles\*2

Set holesOnSide to nHoles

Repeatedly

Put number of initial beans in each hole on each side

Set north and south pot to 0

Total beans = beans in each hole \* total holes

int Board::beans(Side s, int hole) const

if side is North

if we’re at a pot return the # in pot

else if we’re at a valid hole

return the # of beans in the hole

if side is South

if we’re at a pot return the # in pot

else if we’re at a valid hole

return the # of beans in the hole

return -1 if not a valid hole

int Board::beansInPlay(Side s) const

if side is North

count total beans in all of North’s holes

if side is South

count total beans in all of South’s holes

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

if the hole is invalid return false

if hole is a pot return false

for each side

if the hole is empty return false

store number of beans to sow

remove the beans from the hole

repeatedly

if we’re at out pot

add to our pot, switch sides

if we’re at a hole in our side

add a bean to that hole & move next

if we’re at opponent’s pot

switch to other side

if we’re at opponent’s pot

add a bean to that hole & move next

set endSide to currentSide

set EndHole to current hole (or pot)

return true

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

if the hole is invalid return false

if hole is a pot return false

for each side

store number of beans in hole

remove the beans

add the beans to the belonging potOwner

return true

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

if the hole is invalid return false

if hole is a pot return false

for each side

if we’re at side’s pot

set pot to beans value

add or subtract to the total beans

if we’re at one of side’s holes

set beans in hole to beans value

add or subtract to the total beans

Board::~Board()

Delete elements in north array

Delete elements in south array

Board::Board(const Board&other)

Set all integer values (total holes, pot values, etc) to other’s values

Make new north array with # of holes

Make new south array with # of holes

Copy each element from other’s north and south arrays into our north and south arrays

Board& Board::operator=(const Board& rhs)

If this != rhs

Delete elements of current north and south arrays

Set all integer values (total holes, pot values, etc) to other’s values

Make new north array with # of holes

Make new south array with # of holes

Copy each element from other’s north and south arrays into our north and south arrays

Return \*this

**Player.cpp**

Player constructor, player name(), player isInteractive(), player chooseMove() all trivial

humanPlayer isInteractive is trivial, returns true unlike other isInteractives

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

Prompt player to cin a move

If the move is invalid

If the hole exists but there are no beans in that hole, tell user

If the hole is an invalid number, tell the user

Call chooseMove(b, s) to go back to the beginning of the function

Return the move

BadPlayer name trivial

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

for each side

repeatedly

check a hole to see if it’s a valid move

if valid, return the move

increment to next hole otherwise

if no valid moves found, return -1

Smartplayer isInteractive, constructor trivial

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

bestHole = 1

value = 0

call chooseMoveHelper(b, s, bestHole, value)

int SmartPlayer::chooseMoveHelper(const Board& b, Side s, int& bestHole, int& value) const

make a copy of the board

keep track of the original board

if there are no more moves to play

bestHole = -1

the value of the game = evaluate of this board

return bestHole

max = -1000

repeatedly for every hole on player’s side

set our board to the original board to unmake any moves

reset value, depth

if the hole is not empty (valid move)

evaluate that move

if that move’s value is greater than max

max = that move

bestHole = current hole

reset board

if next move is invalid

start loop over

if next hole’s value is better than current max

bestHole = next hole

max = next hole’s value

else

bestHole = current hole

evaluate bestHole to make sure value is set correctly (arbitrary but helpful)

return bestHole

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

if depth = 0 //base case, hit bottom of our function

return value

if it’s not possible to sow //this will also do the sowing for us

return value

if they land in their own pot

for every possible next move they can make

evaluate that move

track which move returns a maximum value

return the max value

if they made a capture

move to pot the captured beans

if the game is over

sweep every bean on both sides to respective pots

for original side

if our pot is bigger

value = 1000 //we won

if our pot is smaller

value = -1000 //we lost

else

value = 0 //we tied

return value

//the game is still in play otherwise

value = difference between our pot and opponents pot

if side == original //we just went, now opponent goes

reduce depth by 1

currentmin = 1000

repeatedly for each of opponent’s possible moves

evaluate that move

if that move is less than current min (will be for anything

other than our player winning)

current min = move

return min

else side != original //our turn again

reduce depth by 1

currentMax = -1000

repeatedly for each of opponent’s possible moves

evaluate that move

if that move is greater than current max

current max = move

return max

Side SmartPlayer::notSide(Side s) const

Return opposite of side s

**Game.cpp**

Constructor, beans trivial functions

Constructor just initializes all the variables and copies board

void Game::display() const also trivial, just couts name of players on each side, their holes, and their pots

TurnSide trivial, takes a player and returns their side

notSide trivial, takes a player and returns opponent’s side

playerSide trivial, takes a side and returns player on that side

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

if both sides has more than 0 beans in play

over = false

return

else //one or both sides has 0 beans in play

over = true

if north’s pot > south’s pot

hasWinner = true

winner = north

if south’s pot > north’s pot

has winner = true

winner = south

else

has winner = false //tie

bool Game::move()

display the board

check the status of the game

if the game is over

sweep beans into pots

update the status post sweeping to get the correct winner

display the board again

return false

set hole to the move the current player picks

display what hole they chose

if they can successfully sow that hole

check the status of the game

if that move ended the game

return move() //call move again to sweep

if they landed in their own pot

display board

state they get another turn

return move() //go again

if they made a capture

move beans into pot, making the capture

switch turns

return true

void Game::play()

{

call m\_play with bool false and turn 0

}

void Game::m\_play(bool s, int k)

{

if both players are interactive

if the move is valid

if s is true

ask user to press enter (first time will use getline)

call m\_play again with (!s, k+1)

else

if hasWinner = false

cout the game resulted in a tie

else

cout the winner of the game

else //one player is interactive at least

if the move is valid

call play again with (!s, k+1)

else

if hasWinner = false

cout the game resulted in a tie

else

cout the winner of the game

return

}

**Any known bugs, problems**

Overall creating the smartPlayer::chooseMove function was obviously difficult. I tried my best to think through the algorithm logically so that it would work as a minmax algorithm. So far, my chooseMove function has been working for basic test cases on boards I set up where the player had an obvious move it could make to win the game. However, it is possible that the function has bugs that I don’t know about because of the nature of the function. Testing it and tracing through it on small boards is doable, but on a board with 6 holes and 4 beans in each hole, it was very difficult for me to trace through and check that smartPlayer was choosing the best move. So, while it works for moves with lower depths, it may not work for moves that require going all the way down to my depth of 5 (although the theory of recursion is that it should work all the way down). It may be possible that in some cases it does not accurately return the right value for the tree. I also tried playing against my smartPlayer, which was somewhat helpful but obviously is not a thorough test. So at the moment I cannot state exactly what the bugs are, but it is definitely possible that for certain cases smartPlayer doesn’t work 100% accurately. For example, it was hard for me to confirm that the values being returned for each possible move were the correct values because of how big the trees could get, so that was one of the problems I encountered. Overall, however, smartPlayer seems to be able to choose between very bad and very good cases for its possible moves.

**Test Code**

int main()

{

    //Testing Board Constructor//

    //Board(int nHoles, int nInitialBeansPerHole)

    Board h(3, 4);

    Board p(-1, -1);

    assert(h.holes() == 3);

    assert(p.holes() == 1);

    //Testing Board beans(Side s, int hole) const

    assert(h.beans(NORTH, 1) == 4);

    assert(h.beans(SOUTH, 1) == 4);

    assert(h.beans(NORTH, 0) == 0);

    assert(h.beans(SOUTH, 0) == 0);

    assert(p.beans(SOUTH, 0) == 0);

    assert(p.beans(NORTH, 0) == 0);

    assert(p.beans(NORTH, 1) == 0);

    assert(p.beans(SOUTH, 1) == 0);

    assert(p.beans(NORTH, 2) == -1);

    assert(p.beans(SOUTH, 2) == -1);

    //Testing beansInPlay

    assert(h.beansInPlay(NORTH) == 12);

    assert(h.beansInPlay(SOUTH) == 12);

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

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

    //Testing total beans

    assert(h.totalBeans() == 24);

    assert(p.totalBeans() == 0);

    //Testing Sow

    //sow(Side s, int hole, Side& endSide, int& endHole)

    Side es;

    int eh;

    assert(h.sow(NORTH, 1, es, eh) == true);

    assert(es == SOUTH);

    assert(eh == 3);

    assert(h.sow(NORTH, 1, es, eh) == false);

    assert(h.sow(NORTH, 2, es, eh) == true);

    assert(es == SOUTH);

    assert(eh == 2);

    assert(h.sow(NORTH, 1, es, eh) == true);

    assert(es == NORTH);

    assert(eh == 0);

    assert(h.sow(SOUTH, 3, es, eh) == true);

    assert(es == SOUTH);

    assert(eh == 1);

    assert(h.sow(SOUTH, 3, es, eh) == false);

    assert(h.sow(NORTH, 2, es, eh) == true);

    assert(es == NORTH);

    assert(eh == 1);

    assert(h.sow(NORTH, 1, es, eh) == true);

    assert(es == SOUTH);

    assert(eh == 1);

    //Testing beans in play at the moment

    assert(h.beansInPlay(NORTH) == 5);

    assert(h.beansInPlay(SOUTH) == 14);

    assert(h.sow(SOUTH, 1, es, eh) == true);

    assert(es == SOUTH);

    assert(eh == 2);

    //Testing total beans to make sure it hasn't changed

    assert(h.totalBeans() == 24);

    //Testing beansInPlay atm

    assert(h.beansInPlay(NORTH) == 8);

    assert(h.beansInPlay(SOUTH) == 10);

    assert(h.sow(SOUTH, 3, es, eh) == true);

    assert(es == SOUTH);

    assert(eh == 0);

    //Testing beans in hole are correct

    assert(h.beans(SOUTH, 0) == 3);

    assert(h.beans(NORTH, 0) == 4);

    assert(h.beans(NORTH, 1) == 1);

    assert(h.beans(NORTH, 2) == 1);

    assert(h.beans(NORTH, 3) == 6);

    assert(h.beans(SOUTH, 1) == 1);

    assert(h.beans(SOUTH, 2) == 8);

    assert(h.beans(SOUTH, 3) == 0);

    //Testing moveToPot(Side s, int hole, Side potOwner)

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

    assert(h.beans(SOUTH, 0) == 4);

    assert(h.beans(NORTH, 0) == 4);

    assert(h.moveToPot(NORTH, 1, NORTH) == true);

    assert(h.beans(NORTH, 0) == 5);

    assert(h.moveToPot(NORTH, 2, SOUTH) == true);

    assert(h.beans(SOUTH, 0) == 5);

    //Testing setBeans(Side s, int hole, int beans)

            //first test total and beans at play for reference

    assert(h.beansInPlay(NORTH) == 6);

    assert(h.beansInPlay(SOUTH) == 8);

    assert(h.totalBeans() == 24);

    assert(h.setBeans(SOUTH, 1, 1) == true);

    assert(h.totalBeans() == 25);

    assert(h.beansInPlay(SOUTH) == 9);

    assert(h.setBeans(SOUTH, 2, 4) == true);

    assert(h.totalBeans() == 21);

    assert(h.beansInPlay(SOUTH) == 5);

    doBoardTests();

    cerr << "Passed all Board Tests" << endl;

    //////Early Player Tests!!//////

  Board yo(3, 0);

    HumanPlayer hehe("hehe");

    assert(hehe.name() == "hehe");          //testing name works with constructor inheritance

    assert(hehe.isInteractive());           //testing isInteractive

    BadPlayer oof("oof");

    assert(oof.name() == "oof");

    assert(!oof.isInteractive());

    assert(oof.chooseMove(yo, NORTH) == -1 && oof.chooseMove(yo, SOUTH) == -1); //testing badplayer has no moves

    yo.setBeans(NORTH, 1, 1);

    assert(hehe.chooseMove(yo, NORTH) == 1);    //testing prompting until valid move

    assert(oof.chooseMove(yo, NORTH) == 1 && oof.chooseMove(yo, SOUTH) == -1);      //bad player now has one move it can make

    HumanPlayer hp("Marge");                //some of smallberg's player tests (used in later tests so kept in)

    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);

    doPlayerTests();//smallberg's player tests

    cerr << "Passed all Player Tests" << endl;

    /////Game Tests/////

    Game li(b, &hp, &bp);

    Board c(6, 2);

    Game e(c, &hp, &bp);

    Board a(2, 2);

    Game o(a, &hp, &bp);

    //g.display();

    e.display();                    //testing that the display is working correctly

    o.display();

    li.play();

    BadPlayer bp2("Dan");               //two non interactive players against each other

    assert(bp2.name() == "Dan"  &&  !bp2.isInteractive());

    Game k(b, &bp, &bp2);

    k.play();

    HumanPlayer niko("Niko");       //played against some friends to see if it was working

    HumanPlayer mad("Maddie");

    HumanPlayer oli("Oliver");

    Board j(6, 3);

    Game yay(j, &niko, &mad);

    yay.play();

    Game dank(j, &oli, &mad);

    dank.play();

    doGameTests();      //smallberg's tests

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

    /// SMART PLAYER TESTS!!! /////

    //Set up games where smartie has one obvious move that will cause it to win

    SmartPlayer smartie("smartie");

    Board w(3, 1);

    w.setBeans(SOUTH, 3, 0);

    w.setBeans(NORTH, 1, 0);

    w.setBeans(NORTH, 2, 0);

    /\*

    0 0 1

   0     0

    1 1 0

     \*/

    Game goodluck(w, &smartie, &mad);

    assert(smartie.chooseMove(w, SOUTH) == 2);

    Board sm(3, 0);

    sm.setBeans(SOUTH, 1, 1);

    sm.setBeans(NORTH, 2, 2);

    /\*

     0 2 0

    0     0

     1 0 0

     \*/

    Game test1(sm, &smartie, &mad);

    assert(smartie.chooseMove(sm, SOUTH) == 1);

    Board sm2(3, 0);

    sm2.setBeans(NORTH, 2, 2);

    sm2.setBeans(SOUTH, 1, 1);

    sm2.setBeans(SOUTH, 3, 1);

    /\*

     0 2 0

    0     0

     1 0 1

     \*/

    Game test2(sm2, &smartie, &mad);

    assert(smartie.chooseMove(sm2, SOUTH) == 3 || smartie.chooseMove(sm2, SOUTH) == 1);

    test2.play();

    bool hasWinner;

    bool isOver;

    Side s;

    test2.status(isOver, hasWinner, s);

    assert(isOver == true && hasWinner == true && s == SOUTH);

    Board sm3(4, 0);

    sm3.setBeans(NORTH, 2, 1);

    sm3.setBeans(SOUTH, 3, 2);

    sm3.setBeans(SOUTH, 4, 1);

    /\*

     0 1 0 0

    0       0

     0 0 2 1

     \*/

    Game test3(sm3, &smartie, &mad);

    assert(smartie.chooseMove(sm3, SOUTH) == 4);

    test3.play();

    test3.status(isOver, hasWinner, s);

    assert(isOver == true && hasWinner == true && s == SOUTH);

    //This tests that smartie will recognize other player's countermove as well

    Board sm4(4, 0);

    sm4.setBeans(NORTH, 3, 1);

    sm4.setBeans(SOUTH, 1, 1);

    sm4.setBeans(SOUTH, 3, 1);

    /\*

     0 0 1 0

    0       0

     1 0 1 0

     \*/

    Game test4(sm4, &smartie, &mad);

    assert(smartie.chooseMove(sm4, SOUTH) == 3);

    test4.play();

    test4.status(isOver, hasWinner, s);

    assert(isOver == true && hasWinner == true && s == SOUTH);

    Board sm5(6, 0);

    sm5.setBeans(NORTH, 4, 2);

    sm5.setBeans(SOUTH, 3, 1);

    sm5.setBeans(SOUTH, 5, 2);

    /\*

     0 0 0 2 0 0

    0           0

     0 0 1 0 2 0

     \*/

    Game test5(sm5, &smartie, &mad);

    assert(smartie.chooseMove(sm5, SOUTH) == 5 || smartie.chooseMove(sm5, SOUTH) == 3);

    test5.play();

    test5.status(isOver, hasWinner, s);

    assert(isOver == true && hasWinner == true && s == SOUTH);

    /\*

    0 2 0 0

   0       0

    1 0 1 0

     \*/

    Board sm7(4, 0);

    sm7.setBeans(NORTH, 2, 2);

    sm7.setBeans(SOUTH, 1, 1);

    sm7.setBeans(SOUTH, 3, 1);

    Game test7(sm7, &smartie, &mad);

    assert(smartie.chooseMove(sm7, SOUTH) == 1);

    /\*

    1 0 1

   0     0

    3 2 1

     \*/

    //smartie chooses 3, then gets other moves til it wins

    Board sm9(3, 0);

    sm9.setBeans(NORTH, 1, 1);

    sm9.setBeans(NORTH, 3, 1);

    sm9.setBeans(SOUTH, 1, 3);

    sm9.setBeans(SOUTH, 2, 2);

    sm9.setBeans(SOUTH, 3, 1);

    Game test9(sm9, &smartie, &mad);

    cout << smartie.chooseMove(sm9, SOUTH) << endl;

    assert(smartie.chooseMove(sm9, SOUTH) == 3);

    /\*

    0 0 1 1

   0       0

    1 0 0 0

     \*/

    Board hi(4, 0);

    hi.setBeans(NORTH, 3, 1);

    hi.setBeans(NORTH, 4, 1);

    hi.setBeans(SOUTH, 1, 1);

    Game hitest(hi, &mad, &smartie);

    hitest.play();      //smartie must choose 3 after maddie chooses 1, which it does

    Board sm6(6, 3);

    Game test6(sm6, &smartie, &mad);

    test6.play();

    Board sm8(6, 4);

    Game test8(sm8, &smartie, &mad);

    test8.play();

    Board sm10(6, 4);

    SmartPlayer smarter("smarter");

    Game test10(sm10, &smarter, &smartie);

    test10.play();

    cout << "Passed all Smart Player Tests" << endl;

}