Joshua Liu

105136031

May 18, 2019

CS 32

Project 3 - Kalah

Description of Design of Classes

The program consists of 8 files. These include 4 header files and 4 .cpp files, where declarations and simple implementations are in the header files and the implementations of non-trivial functions are in the .cpp files. We have three different classes, which are Board, Game, and Player. Player is a base class with three derived classes called BadPlayer, SmartPlayer, and HumanPlayer, where BadPlayer chooses a random move, SmartPlayer chooses the most intelligent move possible, and HumanPlayer asks the user for an input to play the game. The major data structures I used were two dynamically allocated arrays in the Board class, which are used to hold the beans that make up the board. Aside from this, all the other variables are not dynamically allocated. The Board class consists of an assignment operator and copy constructor as well in order to make a copy of the board when SmartPlayer is called and it has to generate and analyze a game tree. The only private member function I used was SmartPlayer::smartMove in the SmartPlayer class. This is the recursive function used to determine what the best move is given a board. The implementations of the function and classes will be described in the following sections.

Description of SmartPlayer::chooseMove

void SmartPlayer::smartMove(const Board& board, const Side player, const Side& originalPlayer, int& bestHole, int& value, int& counter) const

I designed the SmartPlayer::chooseMove function by referring to the given pseudocode in the project specifications. The function takes in the game Board, the current player’s side, the original player’s side, the best hole, the value of that best hole, and a counter for keeping track of how many layers the game tree has traversed. The heuristics I used to evaluate the board positions is the difference in the number of beans in the players’ pot. I attempted to add more and alter this heuristic to see if I could get a better result, such as taking into account the number of beans on a player’s side, but it always ended with a less stable result, where it would lose to my BadPlayer in certain cases. Using the heuristic of the difference in the pots, I successfully beat my BadPlayer with all the board configurations that I have tried, meaning that it works quite well. I have also tried to win against my SmartPlayer and I could not no matter how many times I tried. The program takes in the parameters shown above and evaluates the board. It does this by: first seeing if the “game” is over, where the players cannot make any more moves, and setting the value to either 999 if the player is going to win, or -999 if the player is going to lose, and second, looking at the criterion for searching the next node of the game tree, which in my case is the number of layers it has checked, and third, making a move on the board and running the smartMove function again to check for the next best move. In the end, the value of the best position is provided, as well as the hole number that corresponds to that position. The chooseMove function then returns this “best” value and uses it as its turn.

This function and heuristic works well because if the player can go again or perform a capture, the number of beans in their pot will be greater and thus it will choose that move to maximize the beans in its pot compared to the opponent. In our function call, the majority of the parameters are pass-by-reference to ensure that the program does not spend extra time creating copies of the same variables to pass into functions, maximizing efficiency and runtime.

Pseudocode

Board::Board(int nHoles, int nInitialBeansPerHole) – constructor for Board

if number of holes less than zero

set holes to 1

create dynamically allocated array for south and north

if initial bean parameter is less than zero

set initial beans per hole to zero

for all the holes on the board

set them to the given initial amount of beans for south and north

set the pots(position 0) of south and north to zero

set private data member to number of holes

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

if hole is less than 0 or hole is greater than number of holes

return -1

if side is north

return beans in north hole

if side is south

return beans in south hole

int Board::beansInPlay(Side s) const

if side is north

for all the holes

add up number of beans

return sum

if side is south

for all the holes

add up number of beans

return sum

int Board::totalBeans() const

for all the holes on south including pot

find the sum

for all the holes on north including pot

find the sum

add the previous two sums

return overall sum

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

if hole is less than 1 or larger than number of holes on board

return false

if side is north

if all holes in north are zero

return false

if side is south

if all holes in south are zero

return false

while amount of beans of chosen hole is not zero

if side is north

if hole is zero(pot position)

decrement hole for next position(counterclockwise)

if at position of own pot

add a bean in pot

if amount is zero

keep track of the ending side and ending hole

change the side if we reach the pot of north

else

add bean to hole

if amount is zero

keep track of ending side and ending hole

if side is south

if hole is (number of holes + 1)(pot position)

decrement hole for next position(counterclockwise)

if at position of own pot

add a bean in pot

if amount is zero

keep track of the ending side and ending hole

change the side if we reach the pot of south

else

add bean to hole

if amount is zero

keep track of ending side and ending hole

return true

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

if hole is not valid

return false

if side is north

if pot owner is north

put it in north

else

put it in south

return true

if side is south

if pot owner is north

put it in north

else

put it in south

return true

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

if hole is invalid

return false

if side is north

set beans in hole to beans given

return true

if side is south

set beans in hole to beans given

return true

Board::~Board()

delete north array with []

delete south array with []

Board::Board(const Board& other)

dynamically allocate array for north and south side

set number of holes to current data member

for every hole

copy over the beans for south and north

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

if pointer this is not the same as address of other

delete north and south dynamically allocated arrays

dynamically allocated arrays

copy over data members and beans in every hole

return \*this

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

if no beans on side s

return -1;

get user input

keep asking if input is invalid

return input

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

if no beans on side s

return -1

starting from side closest to pot

if side is north

while bean in hole at position closest to pot is zero

move to next hole

return first hole with beans

if side is south

while bean in hole at position closest to pot is zero

move to next hole

return first hole with beans

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

if no beans on side s

return -1

create variables for smartMove

call smartMove

return best hole

void SmartPlayer::smartMove(const Board& board, const Side player, const Side& originalPlayer, int& bestHole, int& value, int& counter) const

if no more moves are possible

create endgame board

move all beans from holes to respective pots

if original player’s pot has more than opponent’s pot

besthole = -1

value = 999 // because we win

else

bestHole = -1

value = -999 // because we didn’t win

return

criterion: if layer in game tree is greater than 4(counter > 4)

bestHole = -1

evaluation: value = difference between original player’s pot and opponent’s pot

return

create tempBoard

for all the holes i

if no beans at this hole

continue

create variables for recursive smartMove

make player sow on temp board

if player’s last bean land in own pot

run smartMove again

else

if capture

perform capture

run smartMove again, incrementing counter(number of layers searched)

set tempBoard to original board to “unmake” move

bestHole has not been initialized(we say -2)

set value to temp value from recursive call

if original player is player in function call

if temp value is greater or equal to value

bestHole = -1

value = v2

else

if temp value is less than or equal to value

bestHole = -2

value = v2

return

void Game::display() const

new line

display player’s name in middle

new line

print number of beans in every hole(north)

new line

print number of beans in pots, north on the very left, south on the very right

new line

print number of beans in every hole(south)

new line

print player’s name in middle

new line

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

if game isn’t over

over = false

return false

else

over = true

if no winner

hasWinner = false, return

else

hasWinner = true,

set winner to winning side

bool Game::move()

if game is over

move beans to respective pots, return false

data member m\_turn 🡪 even means south’s turn, odd means north’s turn

do

if game is over, move beans to respective pot and return false

let user know if player gets to go again

if south’s turn

chooseMove from Player and sow the move

if last bean in own pot

update display, make sure to run loop again

else if can perform caption

move beans respectively

if north’s turn

chooseMove from Player and sow the move

if last bean in own pot

update display, make sure to run loop again

else if can perform caption

move beans respectively

while player needs to go again, rerun do-while

update display

return true

void Game::play()

send welcome message to user

if south has no beans initially

end game after moving beans to north’s pot

return status and say who won

prompt user to press enter to continue

while player can move, move

let user know who’s turn it is

prompt for enter press

update display

get status of game

if there is winner

return winner

else

tie

return

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

call beans(Side s, int hole)

Note on bugs, inefficiencies, or notable problems

While implementing the game, I had the most trouble with SmartPlayer. Tracing through the recursive function and writing out the game tree was a difficult task that took many hours. The implementation of the game itself was not too difficult but did take some time. There are no known bugs to my program and overall the program is efficient. We can change the number of layers the smart player decides to analyze in the smartMove function to see how much time we want the smartMove player to take when deciding what move to make.

Test Cases

The test cases below were used to test my program as I was making it. The Game class was mainly tested using the interface created by Game::display(). There are also test cases that make sure the SmartPlayer can beat the BadPlayer for a couple different board configurations. Each purpose of the tests is explained by the comments following it. All the test cases work and the program should run without any errors.

//=====TEST CASES FOR CLASS BOARD========================================================

Board testingA(3, 0);

Board testingB(100, 30);

Board testingC(-1, -1);

Board testingD(3, 3);

Board testingE(3, 50);

Board testingF(3, 3);

Board testingG(6, 20);

Board testingH(4, 4);

Board testingI(2, 2);

assert(testingA.holes() == 3); // test number of holes

assert(testingC.holes() == 1); // test number of holes when negative

assert(testingC.beans(NORTH, 1) == 0); // test number of beans when negative

assert(testingA.beans(NORTH, 2) == 0); // test number of beans when creating board

assert(testingA.beans(NORTH, 0) == 0); // test number of beans at pot

assert(testingB.beans(SOUTH, 100) == 30); // test beans regular inputs

assert(testingB.beans(SOUTH, 101) == -1); // invalid spot, greater than number of holes

assert(testingB.beans(SOUTH, -1) == -1); // invalid spot, negative holes

assert(testingB.beansInPlay(SOUTH) == 3000); // test number of beans with south

assert(testingB.beansInPlay(NORTH) == 3000); // test number of beans with north

assert(testingA.beansInPlay(SOUTH) == 0); // test number of beans in north

assert(testingA.beansInPlay(NORTH) == 0); // test number of beans in south

Side whichSideItsOn;

int endingPositionOfSow;

assert(testingD.sow(NORTH, 1, whichSideItsOn, endingPositionOfSow) == true

&& whichSideItsOn == Side::SOUTH && endingPositionOfSow == 2

&& testingD.beans(NORTH, 0) == 1);// sow regularly

assert(testingE.sow(NORTH, 1, whichSideItsOn, endingPositionOfSow) == true

&& whichSideItsOn == Side::NORTH && endingPositionOfSow == 0

&& testingE.beans(NORTH, 0) == 8 && testingE.beans(SOUTH, 0) == 0); // sow regularly

assert(testingF.sow(NORTH, -1, whichSideItsOn, endingPositionOfSow) == false

&& whichSideItsOn == Side::NORTH && endingPositionOfSow == 0

&& testingF.beans(NORTH, 0) == 0 && testingF.beans(SOUTH, 0) == 0); // cannot sow with negative position, do nothing and return false

assert(testingG.sow(SOUTH, 5, whichSideItsOn, endingPositionOfSow) == true

&& whichSideItsOn == Side::NORTH && endingPositionOfSow == 2

&& testingG.beans(NORTH, 0) == 0 && testingG.beans(SOUTH, 0) == 2

&& testingG.beans(SOUTH, 5) == 1); // sow regularly from south side

assert(testingH.moveToPot(NORTH, 3, NORTH) == true

&& testingH.moveToPot(SOUTH, 3, NORTH) == true

&& testingH.beans(NORTH, 3) == 0 && testingH.beans(SOUTH, 3) == 0

&& testingH.beans(NORTH, 0) == 8 && testingH.beans(SOUTH, 0) == 0); // moveToPot tested twice, and values add up correctly

assert(testingH.moveToPot(NORTH, -1, NORTH) == false

&& testingH.moveToPot(SOUTH, 5, NORTH) == false

&& testingH.beans(NORTH, 3) == 0 && testingH.beans(SOUTH, 3) == 0

&& testingH.beans(NORTH, 0) == 8 && testingH.beans(SOUTH, 0) == 0); // moveToPot tested with negative numbers, do nothing and return false

assert(testingI.setBeans(NORTH, 2, 100) == true

&& testingI.beans(NORTH, 2) == 100

&& testingI.setBeans(SOUTH, 1, -4) == false

&& testingI.beans(SOUTH, 2) == 2); // test setBeans, normally as well as setting negative number (where it should do nothing and return false)

std::cerr << "Passed all of Joshua's Test Cases for class Board." << std::endl;

//=============TEST CASES FOR PLAYER=====================================================

std::cerr << std::endl;

HumanPlayer Human\_Player\_A("humanA");

Board playerBoard\_A(4, 4);

assert(Human\_Player\_A.isInteractive() == true);

//std::cout << Human\_Player\_A.chooseMove(playerBoard\_A, NORTH) << std::endl;

BadPlayer Bad\_Player\_A("badA");

assert(Bad\_Player\_A.isInteractive() == false);

//std::cout << Bad\_Player\_A.chooseMove(playerBoard\_A, NORTH) << std::endl;

//std::cout << Bad\_Player\_A.chooseMove(playerBoard\_A, SOUTH) << std::endl;

std::cerr << "Passed all of Joshua's test for class Player." << std::endl;

//============TEST CASES FOR GAME========================================================

// test status function with unfinished game

Board gameBoardA(6, 2);

BadPlayer gameBP\_A("GBP\_A");

BadPlayer gameBP\_B("GBP\_B");

Game gameInstanceA(gameBoardA, &gameBP\_A, &gameBP\_B);

gameInstanceA.display();

bool gameInstanceA\_over = false;

bool gameInstanceA\_hasWinner = false;

Side gameInstanceA\_winner = NORTH;

gameInstanceA.status(gameInstanceA\_over,

gameInstanceA\_hasWinner, gameInstanceA\_winner);

assert(gameInstanceA\_over == false && gameInstanceA\_hasWinner == false

&& gameInstanceA\_winner == NORTH);

// test status function with finished game, no winner

Board gameBoardB(6, 0);

BadPlayer gameBP\_C("GBP\_C");

BadPlayer gameBP\_D("GBP\_D");

Game gameInstanceB(gameBoardB, &gameBP\_C, &gameBP\_D);

gameInstanceB.display();

bool gameInstanceB\_over = false;

bool gameInstanceB\_hasWinner = false;

Side gameInstanceB\_winner = NORTH;

gameInstanceB.status(gameInstanceB\_over,

gameInstanceB\_hasWinner, gameInstanceB\_winner);

assert(gameInstanceB\_over == true && gameInstanceB\_hasWinner == false

&& gameInstanceB\_winner == NORTH);

// test status with winner

Board gameBoardC(6, 0);

gameBoardC.setBeans(SOUTH, 0, 15);

BadPlayer gameBP\_E("GBP\_E");

BadPlayer gameBP\_F("GBP\_F");

Game gameInstanceC(gameBoardC, &gameBP\_E, &gameBP\_F);

gameInstanceC.display();

bool gameInstanceC\_over = false;

bool gameInstanceC\_hasWinner = false;

Side gameInstanceC\_winner = NORTH;

gameInstanceC.status(gameInstanceC\_over,

gameInstanceC\_hasWinner, gameInstanceC\_winner);

assert(gameInstanceC\_over == true && gameInstanceC\_hasWinner == true

&& gameInstanceC\_winner == SOUTH);

//==========TESTING MOVE=================================================================

doGameTests();

doPlayerTests();

doBoardTests();

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

// Testing SmartPlayer

SmartPlayer SmartPlayer\_1("Josh");

BadPlayer BadPlayer\_1("Melody");

Board PlayerBoard\_1(6, 4);

Game g(PlayerBoard\_1, &BadPlayer\_1, &SmartPlayer\_1);

g.play();

bool over, hasWinner;

Side winner;

g.status(over, hasWinner, winner);

assert(over == true && hasWinner == true && winner == NORTH); // test to see if SmartPlayer can beat BadPlayer on 6 hole board with 4 beans in each hole

Board PlayerBoard\_2(4, 4);

Game g2(PlayerBoard\_2, &BadPlayer\_1, &SmartPlayer\_1);

g2.play();

g2.status(over, hasWinner, winner);

assert(over == true && hasWinner == true && winner == NORTH); // test to see if SmartPlayer can beat BadPlayer on 2 hole board with 2 beans in each hole

std::cerr << "Passed all asserts!" << std::endl;