CS32 Project 3 Report

#705120273

Melody Chen

Description of design of my classes:

The major data structure I use to construct my holes and ports is a dynamic array. I have two dynamic arrays one for North and one for South. I use the index zero of each array to represent the pots of each player. This makes it easy for me to reference each hole without indexing. For the Board class, I have two private member functions that help me implement the other functions. One helps me add any number of beans to a hole and the other one helps me with my sow function. For the game class, I made one helper function that helps me move all the beans on the board to their corresponding pot. For the other classes necessary for the game, I did not need any significant data structures or helper function except for the SmartPlayer which will be explained later.

Description of design for SmartPlayer:

My design for SmartPlayer follows professor Smallberg’s pseudocode. I used the minimax algorithm to decide on which move to choose through my recursion until either the heuristic is met or the smart player has found a hole that would lead to winner or defeat. For my heuristic, I simply chose the difference between the number of beans in the pots because that seems to work best. For the second criterion, how deep the tree should go, I set it to ~4 layers deep because that too seems to give me the best results.

Pseudocode for non-trivial algorithms:

Sow function:

If hole is an invalid input

return false

while helper function returns false

keep calling the helper function

return true

Sow helper function:

If special condition

take care of special conditions

takes care of skipping opponents pot and shifting indexes

for all holes on one side until no more holes or no more beans

add a bean in each hole

if beans moves = beans\_to\_move

return true

subtract beans moved from beans\_to\_move

correctly index end hole

return false

Human player choose move:

while response from player is invalid

ask for response

check for proper response

return response

Bad player choose move:

for every hole on one side

if the hole is not empty

return hole

Smart player choose move:

create alarm clock

call helper function

return best hole

Smart player choose move helper function:

if either side has no more beans

create a temp board which is a copy of current board

move all beans to pot since there’s a winner

if there is a tie

value = 0

if beans in original pot greater than opponent pot

value = 9999

else

value = -9999

else if deeper than 4 levels

obtain heuristic(difference between beans in pot)

break

for all available hole on current player’s side with beans

test sowing on temp board at hole

if end up sowing in to your own pot

don’t increment deepness

call helper function with current player and updated values

else

if there is a capture

move beans to pot

call helper function with opposite player and updated values

if current player is original player and temp value is greater than value

updates best hole and value

if current player is opponent and temp value is less than value

updates best hole and value

Game status function:

if both sides of the board are empty

game is over

if beans in pots of both sides have same number

there is no winner

else

there is a winner

determine which side is the winner

else

game is not over

Game move function:

if either side of the board is empty

move the beans to their respective sides

obtain move from current player

sow with current player

loop while you end on your own side

if you fulfill capture condition

moved captured beans into your pot

switch current player to opponent

end turn

if you ended in hole 0

obtain chosen move from current player

sow

else

break

switch current player

end turn

Game play function:

while game is not over

keep doing moves

check status of game

if there is a winner

print winner

else

print tie

Test cases:

*//test cases for board*

Board h(5,3); *//board with 5 holes each with 3 beans*

assert(h.holes()==5); *//test if holes work correctly*

assert(h.beans(Side::NORTH, 0)==0); *//check for initial number of beans in pot to be zero*

assert(h.beans(Side::SOUTH, 0)==0);

assert(h.beans(Side::NORTH, 1)==3); *//check holes for initial bean number to be 3*

assert(h.beans(Side::NORTH, 3)==3);

assert(h.beans(Side::NORTH, 5)==3);

assert(h.beans(Side::NORTH, 6)==-1); *//check when looking for beans in an invalid hole*

assert(h.beansInPlay(Side::NORTH)==15); *//correct beansInPlay number*

assert(h.beansInPlay(Side::SOUTH)==15);

assert(h.totalBeans()==30); *//correct total beans number*

Side s;

**int** end;

Board z(6,15);

z.sow(Side::NORTH, 1, s, end); *//check for sowing*

assert(z.beans(Side::SOUTH, 0)==0); *//test for if sowed correctly*

assert(z.beans(Side::NORTH, 0)==2);

assert(end==1); *//test for the right end spot*

assert(s==Side::SOUTH); *//test for the right end side*

assert(z.beans(Side::NORTH, 1)==1); *//test for correct number of beans sowed*

assert(z.beans(Side::NORTH, 4)==16);

assert(z.beans(Side::SOUTH, 1)==17);

Board c(6,20); *//check for sowing with large number*

c.sow(Side::SOUTH, 2, s, end);

assert(c.beans(Side::NORTH, 0)==0); *//no beans should be sowed into north's pot*

assert(end==5); *//check for correct sowing*

assert(s==Side::NORTH);

assert(c.beans(Side::NORTH, 3)==21);

assert(c.beans(Side::SOUTH, 2)==1);

Board d(5, 4);

assert(d.moveToPot(Side::NORTH, 1, Side::SOUTH)==**true**); *//test for moveToPot*

assert(d.beans(Side::NORTH, 1)==0);

assert(d.beans(Side::SOUTH,0)==4);

assert(d.moveToPot(Side::NORTH, 0, Side::SOUTH)==**false**); *//test for moveToPot faulty*

assert(d.beans(Side::NORTH, 1)==0); *//nothing should be changed*

assert(d.beans(Side::SOUTH,0)==4);

assert(d.moveToPot(Side::NORTH, 1, Side::SOUTH)==**true**); *//test for moveToPot*

assert(d.beans(Side::NORTH, 1)==0); *//no beans should be moved*

assert(d.beans(Side::SOUTH,0)==4);

assert(d.setBeans(Side::SOUTH, 0, -10)==**false**); *//test invalid num for set beans*

assert(d.setBeans(Side::SOUTH, 0, 0)==**true**); *//valid set bean number*

assert(d.beans(Side::SOUTH,0)==0);

assert(d.setBeans(Side::SOUTH, 6, 5)==**false**); *//test invalid num for set beans*

d.display();

Board aa(d); *//check if copy constructor works correctly*

assert(aa.totalBeans()==36); *//check if number of beans corresponds to board d*

assert(aa.beansInPlay(Side::SOUTH)==20);

assert(aa.beansInPlay(Side::NORTH)==16);

assert(aa.beans(Side::SOUTH, 0)==0);

assert(aa.moveToPot(Side::SOUTH, 1, Side::NORTH));

assert(aa.beans(Side::NORTH, 0)==4); *//should be changed*

assert(d.beans(Side::NORTH, 0)==0); *//should not be changed*

c = aa; *//test if assignment operator works properly*

c.display();

assert(c.beans(Side::NORTH, 0)==4); *//should be the same as aa*

assert(c.setBeans(Side::NORTH, 1, 10)==**true**);

assert(c.beans(Side::NORTH, 1)==10); *//c should be updated*

assert(aa.beans(Side::NORTH, 1)==0); *//aa should not change*

*//test cases for player*

HumanPlayer e("Josh");

assert(e.isInteractive()==**true**); *//human player should be interactive*

assert(e.name()=="Josh"); *//test if name works correctly*

BadPlayer f("Josh");

assert(f.isInteractive()==**false**); *//bad player should not be interactive*

assert(f.name()=="Josh"); *//test if name works correctly*

assert(f.chooseMove(d, Side::NORTH)>1&&f.chooseMove(d, Side::NORTH)<=5); *//choose move needs to choose move within range*

SmartPlayer g("Josh");

assert(g.isInteractive()==**false**); *//smart player should not be interactive*

assert(g.name()=="Josh"); *//test if name works correctly*

assert(g.chooseMove(d, Side::NORTH)>1&&g.chooseMove(d, Side::NORTH)<=5); *//choose move needs to choose move within range*

*//test cases for Game*

Game i(c, &f, &g);

**bool** over, haswinner;

Side winner;

i.status(over, haswinner, winner); *//test for status*

assert(over==**false**);

assert(haswinner==**false**);

c.setBeans(Side::SOUTH, 0, 4); *//set up for a tied scenario*

**for**(**int** k=0; k<c.holes(); k++){

c.setBeans(Side::NORTH, k+1, 0);

c.setBeans(Side::SOUTH, k+1, 0);

}

Game j(c, &f, &g);

j.status(over, haswinner, winner); *//check status on tied scenario*

assert(over==**true**);

assert(haswinner==**false**); *//should be no winners*

c.setBeans(Side::SOUTH, 0, 5); *//set up for scenario where there is a winner*

c.display();

Game k(c, &f, &g);

k.status(over, haswinner, winner); *//check status on no tied scenario*

assert(over==**true**);

assert(haswinner==**true**); *//should be a winner*

assert(winner==Side::SOUTH);

assert(k.move()==**false**); *//test move method*

c.setBeans(Side::NORTH, 1, 5);

Game l(c, &f, &g);

assert(l.move()==**false**); *//should not be able to move*

cerr<<"Passed all test cases!"<<endl;