Winona Kraus

205-094-361

Project 3

1. The Board class uses two dynamically allocated arrays to hold the values of the beans for each hole. There is two separate data members for the pots of the board. I created a copy contractor and assignment operator for this class for use in other classes.

The Player, HumanPlayer, and BadPlayer classes are simple with the BadPlayer randomly selecting a valid hole number when the chooseMove function is called. The SmartPlayer class, on the other hand, uses three different private member functions. The first one is legalMoves which returns an array of the legal moves meaning an array of hole numbers. The second function is the calculateValue function which returns the difference between the values of the pots of the two players in favor of the smartPlayer. It will return -999 if the player has no chance of winning, and 999 if the player has the majority of the beans and will win. The last function, chooseBestMove, does most of the work, running through each possible legal move and thinking 3 moves ahead. It uses the alarm clock to decide if should stop running early.

The Game class has four private data members: two players, one side for representing whose turn it is, and a board. The display function displays the board similar to who it is shown in the spec.

1. The SmartPlayer::chooseMove returns another function called chooseBestMove. As mentioned earlier, how good a board position is decided by the number of beans in the pots of the two players compared in favor of the SmartPlayer. The implementation currently only thinks three moves ahead. The chooseBestMove function also takes into account multiple turns and captures. For the multiple turns, it will find the move that has the best value and move without looking further ahead.
2. For the Smart::chooseBestMove implementation starting at the for loop

for all possible legal moves

creates a copy of the board

makes a move on copy of board

while the last bean lands in the player’s pot

the player takes another turn

it will make the move with the highest value

then it will check if a capture was made before looping again

checks for a capture

sets value2 to the best move for the opponent

if the value2 is greater than the previous value and it’s the south’s move

bestHole is set to the current legal move

value is set to value2

if the value2 is less than the previous value and it’s the north’s move

bestHole is set to the current legal move

value is set to value2

if it’s come to the beginning of the game tree

then it returns the bestHole

otherwise, it returns value

1. Sometimes, the program will be stuck on running and will not finish running. Even if it works at first, sometimes it will pause and not continue running if though it says it’s running.
2. Test cases:

**Board class:**

Board b(3, 2);

assert(b.holes() == 3 && b.totalBeans() == 12 &&

b.beans(SOUTH, POT) == 0 && b.beansInPlay(SOUTH) == 6); *(checks that the holes.(), totalBeans(), beansInPlay.() return the correct values and checks that the pot is empty since no beans should be in it)*

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); *(checks that the setBeans and moveToPot functions work properly reducing the totalBeans to 11, adding two beans to the south’s pot from the 2nd hole, and adding only one bean to the 1st hole)*

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); *(checks that the sow function returns the correct endSide and endHole, makes sure the function removed the beans from the hole it took them from, and makes sure the beans were put in their proper places)*

assert(b.beans(SOUTH, 4) == -1 && b.sow(NORTH, -2, es, eh) == -1 && moveToPot(NORTH, -1, SOUTH) == -1 && b.setBeans(SOUTH, -2, 5) == -1 && b.setBeans(SOUTH, 4, -2) == -1) *(checks that no invalid hole or bean values can be passed to these functions without them returning -1)*

**Player class:**

HumanPlayer hp("Marge");

assert(hp.name() == "Marge" && hp.isInteractive()); *(checks that the name() function for HumanPlayer works and that isInteractive() returns true)*

BadPlayer bp("Homer");

assert(bp.name() == "Homer" && !bp.isInteractive()); *(checks that the name() function for BadPlayer works and that isInteractive() returns false)*

SmartPlayer sp("Lisa");

assert(sp.name() == "Lisa" && !sp.isInteractive()); *(checks that the name() function for SmartPlayer works and that isInteractive() returns false)*

Board b(3, 2);

b.setBeans(SOUTH, 2, 0);

cout << "=========" << endl;

int n = hp.chooseMove(b, SOUTH);

cout << "=========" << endl;

assert(n == 1 || n == 3); *(checks that the player chooses a valid hole number)*

n = bp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3); *(checks that the player chooses a valid hole number)*

n = sp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3); *(checks that the player chooses a valid hole number)*

**Game class:**

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;

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, POT) == 0 && g.beans(NORTH, -1) 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); *(checks that the status and beans function works properly with the bean function not accepting invalid hole numbers, it is also checking that the game properly copied over the board)*

g.move();

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); *(makes sure the move function works properly. There is only one possible move, so all of the values of the beans should be the values listed. It also checks that there was a successful capture)*

g.move();

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); *(makes sure the move function works properly with a regular move without a capture)*

g.move();

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); *(makes sure the move function works properly)*

g.move();

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); *(makes sure the move() function properly ends the game and that the status function knows the game is over, if there is a winner, and who the winner is)*