**Suraj Shah Project 3 605707212**

**Description**

The combination of these classes simulates the game Kalah. In order to effectively dynamically allocate storage to represent the holes and the board, I used a vector of 2 vectors representing North and South side. I created some private member functions such as sowable, eval, and minmax to help with the smartPlayer chooseMove function. Sowable checks the ability of a turn being made and if a follow up turn is possible as well. Eval is what determines how much to reward the move based on the difference in quantity of beans in the pot. Minmax is a recursive function that create a tree of different possibilities and return the best possible move within 5 seconds.

**SmartPlayer Description**

My chooseMove function starts by checking if there are holes on the player’s side. If there are, then it starts looping through all the valid holes. It calls sowable (as mentioned above) to sow the beans and then it calls minmax. The result of minmax is saved to a variable and if that variable has the largest value, it saves to the variable that will be returned. The hole that corresponds to that value is also saved. Since minmax is a recursive function, this will keep repeating down to a max depth of 7 and fill out the branches. This leads to the highest value and path being followed. Minmax not only check possible moves for the player but also the opponent to predict how to win. Min max also calls eval which, once agains, returns a value the correlates to the difference between the player and opponent’s pot. The larger the difference in the player’s favor, the higher the reward.

**Board**

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

Check if side and hole are valid (if not, return false)

Save the beans to a temporary variable and set the hole to 0

While there are beans to sow:

If it is on the north side

Decrease the hole number and decrease the number of beans to sow

If the hole is 0 but it isn’t the correct side, then don’t sow a bean

If the hole number goes lower than 0, reset the hole and side to be 1 and South.

If it is on the south side

Increase the hole number and sow a bean

If it goes past the number of holes and south was the starting player

Set hole to pot and sow a bean, then set it to max hole number on north side

Return true if no complications

}

**SmartPlayer:Player**

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

If no beans on player’s side, return -1

For valid holes:

*sowable*

If it can sow, then maximizer is true and call *minmax*

If it can’t sow, then maximizer is false and call *minmax*

Save the larger returned value and it’s corresponding hole to variable

Return the best hole

}

SmartPlayer::sowable(Board& board, int move, Side side) const {

Call board.sow and if it cannot sow, return false

After sowing, check if can capture and do it if possible. Then return false

If the player can make another turn, return true

Otherwise return false

}

SmartPlayer::minmax(const Board& board, Side side, bool isMax, int depth, JumpyTimer& jt) const {

If either side has no beans, return *eval*

Exit if the depth has gone past 0

If player is maximizer

For each valid hole:

Copy the board and check if it can sow

If you can sow again

Call minmax on the player’s side

Otherwise

Call minmax on the opponents side

If minmax returns a value greater than the previous highest value, save it to the same variable

Otherwise

Repeat above but call minmax only for opponents side

Return the hole that corresponds to the highest value

}

SmartPlayer::eval(const Board& board, Side side) const {

If there are no more beans in play

If player’s pot has more beans, return with high value

Otherwise, return with large negative value

If pots are equal, return 0

Otherwise if a pot has more than have the beans

Player’s pot, then return high value

Opponent’s pot, return high neg value

Return the difference between the pots squared

}

**Game**

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

If there are no more beans on the player’s side, then the game is over

Check which pot has more beans between North and South and set the one with more beans as winner

If they are equal, then there is no winner

}

Game::move(Side s) {

while the player ends in the pot, keep doing the turn.

check if player’s side has no beans

if so, move other player’s beans to pot

return false

let player choose move and sow the beans accordingly

display the board

if player ends on their side and lands their last bean in an empty hole, move that bean and opponents beans in the corresponding hole to player’s pot

display board

Switch currentplayer and turn

Return true

}

Game::play() {

Check if both player are inactive

If they are, make user press enter between moves

Display starting board

While the game is not over:

Player makes a move

Check status of game

Check status of game again to verify winners

Display the winner’s name unless the game ends in a tie

}

**Bugs and Problems**

Had a lot of trouble getting the timer function to work. In the end I changed the max depth to finish with 5 seconds without needing the timer to mark it off. The sowing function was a bit tricky and required some serious consideration.

**Test cases**

// board test

int nHoles = 6;

int nInitialBeansPerHole = 4;

Board a(nHoles, nInitialBeansPerHole);

assert(a.beansInPlay(NORTH) == 4\*6);

assert(a.beansInPlay(SOUTH) == 4\*6);

assert(a.totalBeans() == 4\*6\*2);

assert(a.setBeans(NORTH, 3, 0) == 1);

assert(a.setBeans(NORTH, -1, 0) == 0);

assert(a.setBeans(NORTH, 7, 0) == 0);

assert(a.totalBeans() == 4\*6\*2-4);

assert(a.beansInPlay(NORTH) == 4\*5);

assert(a.beansInPlay(SOUTH) == 4\*6);

assert(a.setBeans(NORTH, 3, 4) == 1);

Side endSide = SOUTH;

int endHole = 999;

assert(a.sow(NORTH, 3, endSide, endHole) == 1

&& endSide == SOUTH

&& endHole == 1);

// 5 5 0 4 4 4

//1 0

// 5 4 4 4 4 4

assert(a.totalBeans() == 4\*6\*2);

assert(a.beansInPlay(NORTH) == 4\*6-2);

assert(a.beansInPlay(SOUTH) == 4\*6+1);

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

assert(a.totalBeans() == 4\*6\*2);

Board board(6, 3);

//test beans

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

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

assert(board.beans(NORTH, 2) == 3);

assert(board.beans(NORTH, 3) == 3);

assert(board.beans(NORTH, 4) == 3);

assert(board.beans(NORTH, 5) == 3);

assert(board.beans(NORTH, 6) == 3);

assert(board.beans(NORTH, 7) == -1);

assert(board.beans(NORTH, -1) == -1);

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

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

assert(board.beans(SOUTH, 2) == 3);

assert(board.beans(SOUTH, 3) == 3);

assert(board.beans(SOUTH, 4) == 3);

assert(board.beans(SOUTH, 5) == 3);

assert(board.beans(SOUTH, 6) == 3);

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

assert(board.beans(SOUTH, 8) == -1);

assert(board.beans(SOUTH, 8) == -1);

//test totalbeans and set and beansInPlay

assert(board.beansInPlay(NORTH) == 18);

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

assert(board.totalBeans() == 2\*18);

assert(board.setBeans(NORTH, 1, 20));

assert(board.beansInPlay(NORTH) == 18 - 3 + 20);

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

assert(!board.setBeans(NORTH, -1, 20));

assert(!board.setBeans(NORTH, 9, 20));

assert(board.beans(NORTH, 1) == 20);

assert(board.totalBeans() == 18 \* 2 - 3 + 20);

assert(board.setBeans(NORTH, 0, 20));

assert(board.totalBeans() == 18\*2 - 3 + 20 \* 2);

//test movetopot

assert(board.moveToPot(NORTH, 2, NORTH));

assert(board.beans(NORTH, 0) == 23);

assert(board.moveToPot(NORTH, 1, NORTH));

assert(!board.moveToPot(NORTH, 0, NORTH));

assert(board.moveToPot(NORTH, 2, SOUTH));

assert(board.moveToPot(SOUTH, 2, SOUTH));

assert(!board.moveToPot(SOUTH, 0, SOUTH));

//test sow

Board c(6, 4);

// Side endSide;

int endhole;

//Test the Sow Function

assert(c.sow(NORTH, 3, endSide, endhole));

assert(endhole == 1);

assert(endSide == SOUTH);

assert(c.sow(NORTH, 1, endSide, endhole));

assert(endhole == 4);

assert(endSide == SOUTH);

assert(!c.sow(NORTH, 0, endSide, endhole));

assert(c.setBeans(SOUTH, 1, 20));

assert(c.sow(SOUTH, 1, endSide, endhole));

assert(endhole == 6);

assert(endSide == NORTH);

assert(c.sow(SOUTH, 1, endSide, endhole));

assert(endhole == 2);

assert(endSide == SOUTH);

cout << "board -- all passed\n";

//test player

//human player

HumanPlayer will("will");

assert(will.isInteractive());

//bad player

Board d(6, 4);

assert(d.setBeans(SOUTH, 1, 0));

assert(d.setBeans(SOUTH, 2, 0));

assert(d.setBeans(SOUTH, 3, 0));

assert(d.setBeans(SOUTH, 4, 0));

assert(d.setBeans(SOUTH, 5, 0));

assert(d.setBeans(SOUTH, 6, 0));

BadPlayer bad("BadComputer");

assert(!bad.isInteractive());

assert(bad.name() == "BadComputer");

//The bad player should choose the the lowest possible hole that is availiable

assert(bad.chooseMove(d, SOUTH) == -1);

assert(bad.chooseMove(d, NORTH) == 1);

d.sow(NORTH, 1, endSide, endhole);

assert(bad.chooseMove(d, NORTH) == 2);

d.sow(NORTH, 2, endSide, endhole);

assert(bad.chooseMove(d, NORTH) == 1);

cout << "player -- all passed\n";

//test game class

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board f(3, 0);

f.setBeans(SOUTH, 1, 2);

f.setBeans(NORTH, 2, 1);

f.setBeans(NORTH, 3, 2);

Game g(f, &bp1, &bp2);

bool over;

bool hasWinner;

Side winner;

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();

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();

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();

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();

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);

Board alpha(1, 1);

alpha.setBeans(NORTH, 0, 9);

alpha.setBeans(NORTH, 1, 2);

alpha.setBeans(SOUTH, 1, 0);

alpha.setBeans(SOUTH, 0, 10);

BadPlayer cert("billy");

BadPlayer dad("jef");

Game beta(alpha, &cert, &dad);

bool ov = false;

bool haW = false;

Side wen = SOUTH;

beta.status(ov, haW, wen);

assert(ov && haW && wen == NORTH);

alpha.setBeans(NORTH, 1, 1);

Game gamma(alpha, &cert, &dad);

gamma.status(ov, haW, wen);

assert(ov && !haW);

cout << "game -- all passed\n";

//smart player tests

SmartPlayer smartP("smart");

HumanPlayer humanP("caro");

// do a capture

Board board0(4, 0);

board0.setBeans(NORTH, 2, 1);

board0.setBeans(SOUTH, 1, 1);

board0.setBeans(SOUTH, 3, 1);

Game game(board0,&smartP,&humanP);

assert(smartP.chooseMove(board0, SOUTH) == 1);

// do a capture

Board board1(4, 0);

board1.setBeans(NORTH, 2, 1);

board1.setBeans(SOUTH, 1, 1);

board1.setBeans(SOUTH, 3, 1);

Game game1(board1,&smartP,&humanP);

assert(smartP.chooseMove(board1, SOUTH) == 1);

// protect

Board board2(4, 0);

board2.setBeans(NORTH, 2, 1);

board2.setBeans(SOUTH, 1, 1);

board2.setBeans(SOUTH, 3, 1);

board2.setBeans(SOUTH, 2, 1);

Game game2(board2,&smartP,&humanP);

// game2.display();

// game2.move();

// game2.display();

// assert(smartP.chooseMove(board2, SOUTH) == 1);

// protect from capture

Board board3(5, 0);

board3.setBeans(NORTH, 3, 1);

board3.setBeans(SOUTH, 1, 1);

board3.setBeans(SOUTH, 4, 1);

Game game3(board3,&smartP,&humanP);

assert(smartP.chooseMove(board3, SOUTH) == 4);

// do a capture

Board board4(6, 1);

board4.setBeans(NORTH, 3, 50);

board4.setBeans(SOUTH, 3, 0);

board4.setBeans(SOUTH, 6, 0);

Game game4(board4,&smartP,&humanP);

assert(smartP.chooseMove(board4, SOUTH) == 2);

// put a bean in the pot then capture better player

Board board5(6, 1);

board5.setBeans(SOUTH, 2, 3);

board5.setBeans(SOUTH, 4, 0);

board5.setBeans(SOUTH, 5, 0);

board5.setBeans(SOUTH, 5, 0);

board5.setBeans(NORTH, 5, 19);

board5.setBeans(SOUTH, 6, 0);

Game game5(board5,&smartP,&humanP);

assert(smartP.chooseMove(board5, SOUTH) == 2);

// do a hard capture

Board board6(6, 1);

board6.setBeans(SOUTH, 2, 3);

board6.setBeans(SOUTH, 4, 0);

board6.setBeans(SOUTH, 5, 0);

board6.setBeans(SOUTH, 5, 0);

board6.setBeans(NORTH, 5, 19);

Game game6(board6,&smartP,&humanP);

assert(smartP.chooseMove(board6, SOUTH) == 6);

Board board7(6, 1);

board7.setBeans(NORTH, 0, 3);

board7.setBeans(NORTH, 1, 0);

board7.setBeans(NORTH, 2, 0);

board7.setBeans(NORTH, 3, 1);

board7.setBeans(NORTH, 4, 1);

board7.setBeans(NORTH, 5, 1);

board7.setBeans(NORTH, 6, 1);

Game game7(board7,&smartP,&humanP);

assert(smartP.chooseMove(board7, SOUTH) == 2);