Report

1. For the Board class, I have two vectors to represent north and south side. The position 0 represents their pots. In addition, I also have an integer variable to keep track of the amount of holes. For the Player class, I have a string to store the player’s name. For the SmartPlayer class, I have two helper functions. One is minimax, which let me know what the best move is; the other one is eval, which evaluate a position’s value. For the Game class, I have two pointers point to south and north. In addition, the Game class also has a board as its private member. Furthermore, to keep track of the players’ turn, I use a Side variable.
2. In the SmartPlayer::chooseMove , first if there’s no possible move, or the game is over, return -1. Otherwise, it will call miniMax with the search depth of 4 and time limit 5 seconds to get the best move. In miniMax, a position’s value is determined by calling another function, eval. My strategy to determine a position’s value has five major factors. First, if the next move lets a player win, the position’s value is set to a very large number (10000). Yet if the next move lets a player lose, the position’s value is set to a very small number (-10000). Next, if a player’s next move is likely get it another turn (last bean is sown into its pot), the position’s value is set to a large number (1000, smaller than that when the player is going to win). In addition, if the player’s next move is likely to capture its opponent’s beans, set the position’s value to (100). In all other cases, the position’s value is set to the bean’s difference between South and North pots. In conclusion, if the position is most beneficial to the player, it will get a large value; otherwise, it will get a small value. Later on in minimax, it will call itself again to determine the player’s opponent’s value. Lastly, If the opponent' value is better for the player than best seen so far. South will choose the move with large value and set best hole to the current hole that is being tested. North will choose the move with small value and set best hole to the current hole that is being tested.
3. Pseudocode

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

If the hole indicated by (s,hole) is empty or invalid or a pot, this function returns false without changing anything.

Create a count variable to keep tracks of the amount of beans in the indicated hole.

set the hole being sown to 0 beans

Repeat until the amount of beans is decrement to 0

if we are in a pot, advance to the opponent's side

if we are at the end hole of the south, advance to south's pot

if we originally are on the opposite side, skip the hole.

if we are at the end hole of the north, advance to north's pot

if we originally are on the opposite side, skip the hole.

in all other cases, decrement if it were north, increment otherwise

increment the amount of beans in the hole sown.

Return true

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

if beans in its sides' holes are empty, return -1

assign a random valid hole. If the hole is empty, assign again

return hole

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

if beans in both sides' holes are empty, return -1

set search depth to 4

start timer

call minimax; from the minimax, we know the best hole

return the best hole

void SmartPlayer::minimax(int& bestHole, int& value, const Board& b, Side s, int searchDepth) const

if the side is south, set value to -100000000, while set value to 1000000000 if the side is south

if no move for player exists, set best hole to -1

set to 1000000000 if south wins

set to -1000000000 if north win

set value to 0 if there's a tie

if time elapses more than five seconds, return

if the depth is more than 4, set best hole to -1, call eval function to set value and return

for every hole a player can choose

set a temp Board: without moving for real

if it's sowable, make the move

If it was placed in one of the player's own holes that was empty just a moment before, and if the opponent's hole directly opposite from that hole is not empty, capture the beans.

if the bean ends at the the player's pot, it gets another turn

assign a random hole while it is not empty

if the next turn is not possible, break the loop

move the remaining beans to the pots when the game is over

call oponent's minimax to know its next move, decrement the depth

if opponent' value is better for the player than best seen so far. South will choose the move with large value and set best hole to current hole

if opponent’s value is better for the player than best seen so far. North will choose the move with small value and set best hole to current hole

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

create a temporary board to simulate the original board

if one move to win, set value to 10000

if one move to lose, set value to -10000

if one move to get another turn, set value to 1000

if one move to capture, set value to a large num

in regular case, set the value to be the difference between two pots

return value

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

if more moves are possible, set over to false

otherwise, set over to true

if both sides' pots have same amount of beans, they tie and there's no winner and set hasWinner to false

else, find which side's has more beans in its pot and set hasWiner to true

bool Game::move()

call status and return false if the game is over

decide whose turn it is and create a temporary pointer pointing to that player

choose a hole

sow beans from that hole

If it was placed in one of the player's own holes that was empty just a moment before, and if the opponent's hole directly opposite from that hole is not empty, capture the beans.

if the bean ends at the the player's pot, it gets another turn, display the result.

move the remaining beans to the pots when the game is over

set the turn to the other player's

return true

void Game::play()

while the game is not over (do while loop)

display the game

if the players are not interactive, ask user to press enter to continue

call Game::move()

call Game::status to decide if the game is over

if there’s a winner

find which side wins and output who wins

if there’s no winner

output that there’s a tie

4. bugs, serious inefficiencies, or notable problems

1. When two smart players play, the north side in a board(3, 2) and a board(6, 4) always win. When one smart player play with a bad player, it does not always win but at least get a tie.
2. I am not sure how to thoroughly test a smart player’s chooseMove, so it might not play a perfect game.
3. Although it has complete information, Game::display()does always not appear normal. Players’ name might be a bit off the middle. The south pot might be a bit off in a larger board.
4. My smart player always move in five seconds, but I am not sure if it has any effect on my code

5. Test cases

void doBoardTests()

{

Board b(3,2);

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

b.beans(SOUTH, 0) == 0 && b.beansInPlay(SOUTH) == 6);

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, 0) == 2 &&

b.beansInPlay(SOUTH) == 3);

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, 0) == 3 &&

b.beansInPlay(SOUTH) == 1 && b.beansInPlay(NORTH) == 7);

b.setBeans(NORTH, 0, 2);

b.setBeans(NORTH, 1, 3);

b.setBeans(NORTH, 2, 3);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 0, 1);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 3);

b.setBeans(SOUTH, 3, 0);

assert(!b.sow(SOUTH, 0, es, eh) && !b.sow(NORTH, 0, es, eh));//can't sow hole

assert(!b.sow(SOUTH, 1, es, eh) && !b.sow(SOUTH, 3, es, eh) && !b.sow(NORTH, 3, es, eh));//can't sow empty hole

assert(b.sow(SOUTH, 2, es, eh));

assert(es == NORTH && eh == 3);

assert(b.beans(SOUTH, 2) == 0 && b.beans(SOUTH, 3) == 1 && b.beans(SOUTH, 0) == 2 && b.beans(NORTH, 3) == 1); //regular sow on south

assert(b.sow(NORTH, 2, es, eh));

assert(es == SOUTH && eh == 1);

assert(b.beans(NORTH, 1) == 4 && b.beans(NORTH, 0) == 3 && b.beans(SOUTH, 1) == 1 && b.beans(NORTH, 2) == 0); //regular sow on north

assert(b.beansInPlay(SOUTH) == 2 && b.beansInPlay(NORTH) == 5); //test beansInPlay

assert(!b.moveToPot(SOUTH, 0, SOUTH) && !b.moveToPot(SOUTH, -1, SOUTH) && !b.moveToPot(SOUTH, b.holes()+1, SOUTH)); //can't move with an invalid hole

for(int i = 1; i <= b.holes(); i++)

{

b.moveToPot(SOUTH, i, SOUTH);

b.moveToPot(opponent(SOUTH), i, opponent(SOUTH));

}//move everything to pots

assert(b.beansInPlay(SOUTH) == 0 && b.beansInPlay(NORTH) == 0); //test beansInPlay

}

void doPlayerTests()

{

HumanPlayer hp("Marge");

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);

n = sp.chooseMove(b, SOUTH);

assert(n == 1 || n == 3);

b.setBeans(NORTH, 0, 2);

b.setBeans(NORTH, 1, 3);

b.setBeans(NORTH, 2, 3);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 0, 1);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 3);

b.setBeans(SOUTH, 3, 0);

n = sp.chooseMove(b, SOUTH);

assert(n == 2); //can only move legally

n = bp.chooseMove(b, SOUTH);

assert(n == 2); //can only move legally

b.setBeans(NORTH, 1, 0);

b.setBeans(NORTH, 2, 0);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 0);

b.setBeans(NORTH, 0, 2);

b.setBeans(SOUTH, 0, 2);

n = sp.chooseMove(b, SOUTH);

assert(n == -1); //while can't move, return -1

n = bp.chooseMove(b, SOUTH);

assert(n == -1); //while can't move, return -1

}

void doGameTests()

{

SmartPlayer 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;

// Homer

// 0 1 2

// 0 0

// 2 0 0

// Bart

g.display();

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, 0) == 0 && g.beans(SOUTH, 0) == 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();

// 0 1 0

// 0 3

// 0 1 0

g.display();

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, 0) == 0 && g.beans(SOUTH, 0) == 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();

// 1 0 0

// 0 3

// 0 1 0

g.display();

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, 0) == 0 && g.beans(SOUTH, 0) == 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();

// 1 0 0

// 0 3

// 0 0 1

g.display();

g.status(over, hasWinner, winner);

assert(!over && g.beans(NORTH, 0) == 0 && g.beans(SOUTH, 0) == 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();

// 0 0 0

// 1 4

// 0 0 0

g.display();

g.status(over, hasWinner, winner);

assert(over && g.beans(NORTH, 0) == 1 && g.beans(SOUTH, 0) == 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);

//////////////////////////////////

b.setBeans(NORTH, 0, 2);

b.setBeans(NORTH, 1, 3);

b.setBeans(NORTH, 2, 3);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 0, 1);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 3);

b.setBeans(SOUTH, 3, 0);

HumanPlayer bp3("Human");

Game g1(b, &bp1, &bp3);

// Homer

// 3 3 0

// 2 1

// 0 3 0

// Bart

g1.display();

g1.move();

// 3 3 1

// 2 2

// 0 0 1

g1.display();

g1.status(over, hasWinner, winner);

assert(!over && g1.beans(NORTH, 0) == 2 && g1.beans(SOUTH, 0) == 2 &&

g1.beans(NORTH, 1) == 3 && g1.beans(NORTH, 2) == 3 && g1.beans(NORTH, 3) == 1 &&

g1.beans(SOUTH, 1) == 0 && g1.beans(SOUTH, 2) == 0 && g1.beans(SOUTH, 3) == 1);

g1.move();

// 3 4 0

// 9 2

// 0 0 1

g1.display();

g1.move();

// 0 0 0

// 9 3

// 0 0 0

g1.display();

g1.status(over, hasWinner, winner);

assert(over && g1.beans(NORTH, 0) == 9 && g1.beans(SOUTH, 0) == 3 &&

g1.beans(NORTH, 1) == 0 && g1.beans(NORTH, 2) == 0 && g1.beans(NORTH, 3) == 0 &&

g1.beans(SOUTH, 1) == 0 && g1.beans(SOUTH, 2) == 0 && g1.beans(SOUTH, 3) == 0);

assert(hasWinner && winner == NORTH);

assert(!g1.move()); //can't move when the g1ame is over

b.setBeans(NORTH, 1, 0);

b.setBeans(NORTH, 2, 0);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 0);

b.setBeans(NORTH, 0, 2);

b.setBeans(SOUTH, 0, 2);

Game g2(b, &bp1, &bp3);

g2.status(over, hasWinner, winner);

assert(over && g2.beans(NORTH, 0) == 2 && g2.beans(SOUTH, 0) == 2 &&

g2.beans(NORTH, 1) == 0 && g2.beans(NORTH, 2) == 0 && g2.beans(NORTH, 3) == 0 &&

g2.beans(SOUTH, 1) == 0 && g2.beans(SOUTH, 2) == 0 && g2.beans(SOUTH, 3) == 0);

assert(!hasWinner);

}