Kelly Tran

UID: 405808641

Project 3 CS 32

**Design of Classes:**

Board: The Board class represents the state of the game board and provides functions to interact with and manipulate the board. It maintains the number of beans in each hole and pot for both sides. The design allows for easy access to hole and bean information, as well as updating the board state based on game rules, such as sowing and capturing beans. In addition to the provided public and private members, I added a vector to hold the data values that would be represented by the board for Kala. I found that in comparison to doubly linked lists and arrays, it was easier to dynamically allocate storage and change the size of the vector in accordance to the set-values for number of holes provided by the player.

Player: The major data structure of this class was the implementation of Smart Player. The chooseMove function takes the current state of the board (b) and the side of the player (s) as input and returns the bestHole by using the minimax algorithm with alpha-beta pruning. I used an evaluate function to assign a value that determines valuability to a board position by calculating the difference between the number of beans in the player's pot and the opponent's pot. The minimax function uses recursion to determine all possible moves for the player and evaluates the resulting board positions by using alpha-beta pruning to eliminate unnecessary steps. It returns the evaluation value for each position to decide the best move for the player. chooseMove iterates over all valid moves and calls minimax for each move. It keeps track of the move that results in the highest evaluation value for the maximizing player (SOUTH) or the lowest evaluation value for the minimizing player (NORTH). Chrono library is implemented to chooMove so that there is a time limit of 5 seconds so that if the time limit is exceeded, it returns a default move or handles the situation as appropriate.

Game: I added a promptToContinue function to call the buffer statements in functions move and play for efficiency. The main thing about Game.cpp is the move function, which allows a player to make a move in the game. If it's the player's turn and there are beans in play, the method lets the player choose a move. It then sows the beans on the board, updates the state, and checks for special conditions like extra turns (if the player lands on their pot) or captures. After the move, it checks if the game is over. If it is, it displays the final board, sweeps remaining beans into the pots, and returns false. Otherwise, it switches to the opponent's turn and returns true.

**Description of Smart Player:**

The main heuristic is the evaluation function, which calculates the difference between the number of beans in the player's pot and the opponent's pot. A positive value indicates a good position for the maximizing player, while a negative value suggests a bad position. The heuristic assumes that maximizing the number of beans in the player's pot relative to the opponent's pot leads to better gameplay. Evaluation uses the minimax function to explore the game tree and recursively evaluate the possible moves and the resulting values and alternate between maximizing and minimizing players. The minimax algorithm chooses the move that leads to the highest evaluation value when it is the maximizing player's turn and the move that leads to the lowest evaluation value when it is the minimizing player's turn. This allows Smart Player to make intelligent decisions by considering the current board state, anticipating future moves, and evaluating the potential outcomes. The depth parameter in the minimax() function determines how many moves ahead the player considers, allowing the SmartPlayer to make more informed and strategic decisions.

**(Important) Pseudocode:**

Board.cpp:

copy constructor:

Board(other: Board)

Copy the number of holes

Resize the board to match the number of holes

Copy the values of side from the other board

Resize each side of the board to match the number of holes

Copy the value of the pot from the other board

Copy the values of each hole from the other board with a for loop

Sow function:

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

Check if the hole is valid for sowing, return false if not

Initialize variables:

beansToSow = m\_board[s][hole]

m\_board[s][hole] = 0

currentSide = s

currentHole = hole

The actual sowing:

While beans the sow is greater than 0

Check if currentside is NORTH

Move to the previous hole by decrementing

Check if reached the pot

Increment pot and decrement beans to sow

currentSide = opponent(currentSide)

currentHole = 0

else

Increment the hole and decrement beans to sow

else if currentSide == SOUTH

Move to the next hole

Increment currentHole

Check if reached the pot

Increment pot and decrement beans to sow

currentSide = opponent(currentSide)

currentHole = m\_nHoles + 1

else

Increment the hole and decrement beans to sow

Adjust the current hole if necessary

if currentHole == 0 or currentHole == m\_nHoles + 1

currentHole = 0

Assign the end side and end hole

endSide = currentSide

endHole = currentHole

return true

Player.cpp:

SmartPlayer ChooseMove:

chooseMove(b, s) -> int

startTime = current time

depth = 1

maximizingPlayer = (s == SOUTH)

bestHole = -1

if maximizingPlayer

bestValue = -infinity

for hole = 1 to b.holes()

if b.beans(s, hole) > 0

bCopy = copy of b

if bCopy.sow(s, hole, endSide, endHole)

value = minimax(bCopy, opponent(s), depth, -infinity, +infinity, false)

if value > bestValue

bestValue = value

bestHole = hole

else

bestValue = +infinity

for hole = 1 to b.holes()

if b.beans(s, hole) > 0

bCopy = copy of b

if bCopy.sow(s, hole, endSide, endHole)

// Handle special moves

// ...

value = minimax(bCopy, opponent(s), depth, -infinity, +infinity, true)

if value < bestValue

bestValue = value

bestHole = hole

endTime = current time

duration = endTime - startTime

if duration >= 5 seconds

return -1

return bestHole

Smart Player Evaluation:

evaluate(b, s) -> int

playerScore = number of beans in player's pot

opponentScore = number of beans in opponent's pot

scoreDifference = playerScore - opponentScore

return scoreDifference

Smart Player Minimax:

minimax(board, side, depth, alpha, beta, maximizingPlayer):

if depth is 0 or no beans are left for either player:

return evaluate(board, side)

if maximizingPlayer:

maxEval = -infinity

for each hole in board:

if board.beans(side, hole) > 0:

create a copy of the board called boardCopy

sow beans from side's hole on boardCopy

determine endSide and endHole based on the sow operation

if endHole is POT:

if there are beans remaining for side:

eval = minimax(boardCopy, side, depth, alpha, beta, true)

else:

sweep remaining beans into respective player's pot

eval = evaluate(boardCopy, side)

else if endSide is side and boardCopy.beans(endSide, endHole) is 1 and boardCopy.beans(opponent(side), endHole) > 0:

capture beans in the empty hole and the opponent's hole

eval = minimax(boardCopy, side, depth - 1, alpha, beta, false)

maxEval = max(maxEval, eval)

alpha = max(alpha, eval)

if beta is less than or equal to alpha:

break

return maxEval

else:

minEval = +infinity

for each hole in board:

if board.beans(side, hole) > 0:

create a copy of the board called boardCopy

sow beans from side's hole on boardCopy

determine endSide and endHole based on the sow operation

if endHole is POT:

if there are beans remaining for side:

eval = minimax(boardCopy, side, depth, alpha, beta, false)

else:

sweep remaining beans into respective player's pot

eval = evaluate(boardCopy, side)

else if endSide is side and boardCopy.beans(endSide, endHole) is 1 and boardCopy.beans(opponent(side), endHole) > 0:

capture beans in the empty hole and the opponent's hole

eval = minimax(boardCopy, side, depth - 1, alpha, beta, true)

minEval = min(minEval, eval)

beta = min(beta, eval)

if beta is less than or equal to alpha:

break

return minEval

Game.cpp:

Move:

move(Side s):

display the current state of the game board

if it is not the current player's turn or there are no beans in play for the current player:

return false

let the current player choose a move

if the chosen move is valid:

sow beans and update the board state

if the last sow ends in the current player's pot:

display the updated board

if the current player still has beans in play:

display a message indicating they get another turn

recursively call move(s) for the current player

else if the last sow ends on the current player's side and captures opponent's beans:

capture the beans and update the board state

display the updated board

check if the game is over for either player

if the game is over:

sweep remaining beans into the respective player's pot

return false

switch the current player to the opponent

return true

Play:

play:

while True:

call move(s) for the current player

if the current player has beans in play:

if the current player is interactive:

prompt the user to continue

call move(s) for the current player again

if move(s) returns false:

sweep beans into the opponent's pot if no valid move is available

switch the current player to the opponent

check if the game is over

if the game is over:

display the winner or a tie message

break

**Bugs, Inefficiencies, Problems:**

I had a really hard time figuring out how to add buffer statements in Game.cpp because I didn’t want the code to run through the game too quickly without making sure the player was interactive with the terminal statements. I also kept on having to tweak my sow function in Board.cpp because it would sow into the wrong holes. Eventually I figured it out after looking through the breakpoints.

The functions I definitely had the most trouble with were the Smart Player functions. I had to look towards other videos explaining how to use the minimax functions, and even then, it took a lot of time having to write out each possibility to generate the game tree. After debugging for like 5 hours, I found out how to run Smart Player correctly so long as the depth is 0 or 1… A current problem is still that whenever I increase the depth, the Smart Player function, in particular, minimax, runs incorrectly and returns the wrong hole for bestHole when chooseMove is called. I’m also not sure how effective the chrono library is, because I don’t think there’s been a scenario where the time duration took over 5 seconds.

**Test Cases:**

Board.cpp:

Board b(3, 2);

assert(b.holes() == 3 && b.totalBeans() == 12 && b.beans(SOUTH, POT) == 0 && b.beansInPlay(SOUTH) == 6); // checks to see that initial beans in hole are implemented correctly

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 to see that setBeans and moveToPot are functional

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 Board can sow properly and still report back the correct beansInPlay

Player.cpp without functionality of Smart.cpp:

HumanPlayer hp("Marge");

assert(hp.name() == "Marge" && hp.isInteractive()); //checks that HumanPlayer is in fact a human player :)

BadPlayer bp("Homer");

assert(bp.name() == "Homer" && !bp.isInteractive()); //checks that BadPlayer is a computer

SmartPlayer sp("Lisa");

assert(sp.name() == "Lisa" && !sp.isInteractive()); //checks that SmartPlayer is a computer

Board b(3, 2);

b.setBeans(SOUTH, 3, 0);

int n = hp.chooseMove(b, SOUTH); //makes sure that chooseMove is functional

assert(n == 1 || n == 2); //checks that the players chose 1 or 3

n = bp.chooseMove(b, SOUTH);

assert(n == 1 || n == 2);

n = sp.chooseMove(b, SOUTH);

assert(n == 1 || n == 2);

Game.cpp:

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(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 the beans were set correctly

g.move(SOUTH);

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);

//checks that move for South works correctly

g.move(NORTH);

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);

//checks that move for North works correctly

g.move(SOUTH);

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(NORTH);

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);

//checks that play and status work correctly

}

Tests everything without SmartPlayer:

HumanPlayer hp("Marge");

BadPlayer bp("Homer");

Board b(3, 2);

Game g(b, &hp, &bp);

g.play();

//game play works ok, display is accurate, sow is accurate, winner is declared correctly

SmartPlayer:

Board board(0, 0);

SmartPlayer player("Alice");

int move = player.chooseMove(board, SOUTH);

assert(move == -1);

//tests an empty board

{

Board board(0, 0);

board.setBeans(SOUTH, 1, 1);

SmartPlayer player("Alice");

int move = player.chooseMove(board, SOUTH);

assert(move == 1);

}

//tests that an empty board if set with a bean, there will be a hole so there should be a valid move

Board board(0, 0);

board.setBeans(SOUTH, 1, 4);

board.setBeans(SOUTH, 3, 6);

board.setBeans(SOUTH, 6, 3);

board.setBeans(NORTH, 3, 3);

board.setBeans(NORTH, 4, 1);

board.setBeans(NORTH, 6, 1);

SmartPlayer player("Alice");

int move = player.chooseMove(board, SOUTH);

assert(move == 1 || move == 2 || move == 4);

//tests capturing

Board board(0, 0);

board.setBeans(SOUTH, 1, 4);

board.setBeans(SOUTH, 6, 3);

board.setBeans(NORTH, 3, 3);

board.setBeans(NORTH, 4, 1);

board.setBeans(NORTH, 6, 1);

SmartPlayer player("Alice");

int move = player.chooseMove(board, SOUTH);

assert(move == 1 || move == 2 || move == 3 || move == 5);

//tests additional turns

Board board(0, 0);

board.setBeans(SOUTH, 1, 4);

board.setBeans(SOUTH, 3, 6);

board.setBeans(SOUTH, 6, 3);

board.setBeans(NORTH, 3, 3);

board.setBeans(NORTH, 4, 1);

board.setBeans(NORTH, 6, 1);

SmartPlayer player("Alice");

int move = player.chooseMove(board, SOUTH);

assert(move >= 1 && move <= 6);

//tests depth of Smart Player

Tests Time:

Board board(1, 1);

SmartPlayer smartPlayer("SmartPlayer");

auto startTime = std::chrono::steady\_clock::now();

smartPlayer.chooseMove(board, SOUTH);

auto endTime = std::chrono::steady\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime).count();

double baselineTime = static\_cast<double>(duration) / 1000.0;

assert(baselineTime <= 5.0);

//makes sure that time is less than 5 seconds

Board board(4, 4); // 4 holes per side, 4 initial beans per hole

SmartPlayer smartPlayer("SmartPlayer");

auto startTime = std::chrono::steady\_clock::now();

smartPlayer.chooseMove(board, SOUTH);

auto endTime = std::chrono::steady\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(endTime - startTime).count();

double executionTime = static\_cast<double>(duration) / 1000.0;

assert(executionTime <= 5.0);

//tests for when there’s a larger board