Wolfe Pickett

CS 32 Project 3

Report

Description of the design of classes:

**Board** class uses two integer vectors, one for each side (North and South). I also have a private member function, m\_holes, which stores the number of holes on a side (not counting the pot).

**Player** class has only one private member, m\_name, which has information about the player. HumanPlayer, BadPlayer, and SmartPlayer are subclasses of Player. They all have a constructor and chooseMove. SmartPlayer has private functions eval (a function that given a position, returns a number that measures how good that position is for the player) and minimax (to search the game tree so that it finds what it considers to be the best play available).

**Game** class uses a Board m\_board, pointers to both players, North and South, and a Side m\_turn that indicates a current turn.

SmartPlayer::chooseMove:

The chooseMove() method of the SmartPlayer class is designed to find the best move for the SmartPlayer. It uses the minimax algorithm with alpha-beta pruning to evaluate and select the best move for the player. The overall design is as follows:

The player first checks if the game is over. If it is, the player returns -1. The player then chooses a search depth. The search depth is the number of steps ahead that the player will look. The player then calls the minimax function to recursively search all possible board positions. The minimax function returns the best possible move for the player. The player then returns the best possible move.

The eval function is used within the minimax function to evaluate board positions. It incorporates several heuristics to assign a value to each board state: the number of beans in the player's pot, the number of beans in the opponent's pot, the number of beans that the player can capture from the opponent, the number of beans that the opponent can capture from the player, the number of beans that the player can sow in the opponent's holes, and the number of beans that the opponent can sow in the player's holes. The player chooses the move that maximizes the value of the board position. The minimax function recursively evaluates all possible moves and their subsequent outcomes up to the specified search depth, considering both the player's and the opponent's positions, and updates the best move and value accordingly.

Pseudocode for non-trivial algorithms:

Board.cpp:

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

if hole is invalid

return false

count = number of beans from the specified hole

while count > 0

if endHole is at the starting position

move to the last hole on the board

switch to the opponent's side

else if endHole is at the last hole and on the SOUTH side

if the current player is SOUTH

move to the first hole on the board (the pot of SOUTH)

else

move to the last hole on the board

else if endHole is at the first hole and on the NORTH side

if the current player is NORTH

move to the first hole on the board (the pot of NORTH)

else

move to the second hole on the board

else

if the current side is NORTH

move to the previous hole

else

move to the next hole

increment the number of beans in the current hole by 1

decrease the count by 1

return true

Player.cpp:

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

if there are no beans in any of the sides' holes -> return -1

start from the first hole

while i <= board.holes()

ff there are beans in the current hole

return the index of the first non-empty hole found

move to the next hole

if no non-empty hole is found -> return -1

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

initiliaze value

check if one move can lead to a win or loss

value is large -> Winning position

value is small -> Losing position

value is 0 -> Draw

calculate the difference between player's pot and opponent's pot

check if one move can grant an extra turn

extra turn possibility

check if one move can capture opponent's seeds

capture possibility

return value

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

set initial value to infinity for NORTH and negative infinity for SOUTH

check if the game is over

set value to positive infinity if SOUTH wins

set value to negative infinity if NORTH wins

set value to 0 if there's a tie

check if reached the maximum search depth

evaluate the board state using the eval function

iterate through each hole

create a temporary board to simulate moves

sow the seeds and get the end hole and side

skip if the move is invalid

handle capturing opponent's seeds if conditions are met

handle repeated turns until endHole is 0 and side is the same

check if the game is over after the moves

move remaining seeds to the pot

recursive call to minimax for the opponent's turn

update bestHole and value based on the opponent's value

update alpha value, update beta value

perform alpha-beta pruning if the cutoff condition is met

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

initialize the best hole to an invalid value

initialize the initial value based on the player's side

return -1 if the game is over

initialize maximum search depth for the minimax algorithm

initialize the alpha value to negative infinity

initialize the beta value to positive infinity

create a timer to measure the elapsed time

set the time limit to 5 seconds in milliseconds

check if the hole is not empty

create a temporary board to simulate moves

skip if the move is invalid

capture opponent's seeds if conditions are met

apply minimax algorithm to evaluate the board state

if SOUTH

update the best hole if a better move is found

update the value with the opponent's value

update the alpha value

if NORTH

update the best hole if a better move is found

update the value with the opponent's value

update the beta value

return the best hole found

Game.cpp:

**bool Game::move(Side s)**

check the status of the game

determine the player based on the current turn

player selects a move

perform the selected move

check if the move results in a capture

continue with additional turns as long as the conditions are met

display the current state of the game

switch the turn to the opponent

check if the game is over since all beans being collected by one player

move remaining beans to the south/north player's pot

return false to indicate the game is over

return true to continue the game

Any known bugs, serious inefficiencies, or notable problems you had:

I had lot of difficulty determining how I should implement the board class and its private members. I could have implemented a data structure (like an array or vector) to store all the holes and have several helper functions. However, this task would require more inefficient code, so I opted to store them into a vector. I also had a few bugs in my Board class (that were related to indexing) that I later resolved. The minimax and associated chooseMove algorithms for SmartPlayer were also slow, so I decided to add alpha–beta pruning to decrease the number of nodes that are evaluated by the minimax algorithm to reduce the amount of searching.

A list of the test cases that would thoroughly test the functions:

**void** test()

{

// Test Case 1: Empty Board

Board b1(0, 0);

assert(b1.beansInPlay(NORTH) == 0);

assert(b1.beansInPlay(SOUTH) == 0);

assert(b1.totalBeans() == 0);

// Test Case 2: One Hole

Board b2(1, 3);

assert(b2.beansInPlay(NORTH) == 3);

assert(b2.beansInPlay(SOUTH) == 3);

assert(b2.totalBeans() == 6);

// Test Case 3: Negative beans

Board b3(2, -1);

assert(b3.beansInPlay(NORTH) == 0);

assert(b3.beansInPlay(SOUTH) == 0);

assert(b3.totalBeans() == 0);

}

// Test Board class

**void** testBoard() {

Board board(6, 4);

// Test beansInPlay()

assert(board.beansInPlay(NORTH) == 24);

assert(board.beansInPlay(SOUTH) == 24);

// Test totalBeans()

assert(board.totalBeans() == 48);

// Test setBeans()

board.setBeans(NORTH, 2, 3);

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

// Test beans()

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

// Test moveToPot()

assert(board.moveToPot(NORTH, 4, NORTH));

assert(board.beans(NORTH, 4) == 0);

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

// Test sow()

Side endSide;

**int** endHole;

assert(board.sow(SOUTH, 1, endSide, endHole));

assert(endSide == NORTH);

assert(endHole == 3);

assert(board.beans(SOUTH, 1) == 0);

assert(board.beans(NORTH, 2) == 5);

}

**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);

}

**void** doGameTests()

{

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;

// Homer

// 0 1 2

// 0 0

// 2 0 0

// Bart

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(SOUTH);

// 0 1 0

// 0 3

// 0 1 0

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(NORTH);

// 1 0 0

// 0 3

// 0 1 0

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(SOUTH);

// 1 0 0

// 0 3

// 0 0 1

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);

// 0 0 0

// 1 4

// 0 0 0

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);

}

**void** testSow()

{

//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);

}