Joanna Liu

405-957-891

**CS32 Spring 2023 - Project 3 Report**

1. **A description of the design of your classes.**

Board Class

The data structure I used to represent my Kahla board consisted of two vectors, one to represent the north side of the board and one to represent the south side of the board. For both of the vectors, the 0 index was designated to be the pot for that side. The number of holes per side on the board and the number of beans each hole started with I stored as integer variables.

An extra private member function in the Board class was added to streamline checking if an input is a valid hole. Given an integer that is supposed to represent a hole number, the function (named invalidHole) will return true if the hole is invalid, meaning that it is either a pot or out of the bounds of the board, or false if the hole is a valid hole on the current board.

Moving and sowing the beans functioned primarily through simple accesses of indexes in the north and south vectors, and for loops if necessary (such as in the sow function).

Player and SmartPlayer

The Player class had an additional three extra private functions that were used to implement SmartPlayer’s chooseMove function, as well as the private variables m\_name in the Player base class to store the player’s name, and max\_depth in the SmartPlayer class to store the maximum depth for the game tree to be searched.

The first private function is the evaluate function, which, given a board, will return an integer that represents how good or bad the current position is for a given side. The heuristic used was the difference in the number of beans in each pot, with the difference being calculated as (North pot’s beans) - (South pot’s beans). So, the more positive the return value, the better the current board position is for the North side and the worse the current position for the South side. Similarly, the more negative the return value, the better the current board position is for the South side and the worse the current position for the North side. If either side has the majority of beans, it means that even if the opponent is to obtain all beans currently still in play on the board, the former side would still win the game. In this case, the evaluate function either returns a very positive value if North has a majority of the beans (9999) or a very negative value if South has a majority of the beans (-9999).

The next private function is the turnFinished function, which sows the beans of the board passed through the parameter and then, depending on where the last bean ends, will return true if the turn has finished (bean didn’t end in a pot), or false if the current side can take another turn (bean ends in pot). This function is called in chooseMoveHelper (more information below) in order to “sow” and “unsow” the beans and get the values for the position of the board if it was sowed from a particular hole.

The last private function added to the SmartPlayer class is the chooseMoveHelper function, which is the main recursive function for creating and traversing the game tree to help the computer find the most ideal move. It takes in seven parameters: the current board, the current side whose turn it is, a variable that will hold the best hole choice (best move), a value that will hold the best value from the evaluate function, an integer that holds the current depth of the game tree, a double that holds the maximum amount of time given to the function to traverse the game tree, and a JumpyTimer object that is used to time how long the function is taking. This function calls turnFinished and checks if the turn has ended yet, then calls the appropriate recursive call by changing the parameters depending on if the turn has ended or not. The algorithm sort of creates a “tree” through recursion, searching down different paths of possible hole choices and evaluating the board after each recursive call to see if performing that move results in a better position for the current playing side than the previous best hole. After all recursive calls of the function have been completed, the best move to the maximum depth (stored as a private variable in the SmartPlayer class) or maximum time (five seconds) will be stored in the bestHole parameter.

Game Class

My implementation of the Game class had 4 private data members: the board, the current side, and two Player pointers pointing to the player object that represented the north side and the player object that represented the south side in the game. The reason why Player pointers were used was so that any type of player derived from the base Player class could be used as a player of the game.

Side/Main Files - Extra non-member functions

I added one global constant double, MAX\_TIME, which stores the maximum time that the SmartPlayer is allowed to take in order to make a move. Because the project specified the maximum time should be 5 seconds, I set MAX\_TIME to the recommended time of 4990 ms, which gives 10 ms of buffer time.

I also included the Timer and JumpyTimer declarations and implementations in these files, which primarily define a system for keeping track of the amount of time that passes for a certain number of function calls.

1. **A description of your design for SmartPlayer::chooseMove, including what heuristics you used to evaluate board positions.**

My SmartPlayer::chooseMove begins by creating a JumpyTimer that is used to make sure the SmartPlayer does not take more than five seconds to return a move choice. The function then calls a helper function, chooseMoveHelper, whose primary purpose is to use recursion to traverse a game tree to find the best hole to choose to result in the best position. Once the chooseMoveHelper finishes running, the best move that the function was able to find given the maximum depth allowed and the maximum time allowed would be saved in the variable bestHole, which was passed as an int reference parameter. This value is then returned by chooseMove. The position of the board was evaluated through the evaluate function, which is defined by the heuristic that the more beans you have in your pot compared to your opponent, the better. More information about the inner workings of the evaluate and chooseMoveHelper function is defined in part 1 of the report.

1. **Pseudocode for non-trivial algorithms.**

Board

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

If the hole given is invalid, return false.

If the North side is sowing:

Remove the beans from that hole on the north side, returning false if there are no beans

While you still have beans:

Starting from the hole passed in the parameter, drop beans counterclockwise   
 (leftwards) on the North side of the board.

If you run out of beans,

set endHole to the last hole you dropped a bean.

set endSide to North.

return true.

If you run out of holes on the North side and still have beans:

Drop a bean in the North pot.

If that was your last bean:

set endHole to the pot.

set endSide to North.

return true.

If you still have beans:

Starting from the leftmost hole on the south’s side, drop beans counterclockwise

(rightwards) on the South side of the board   
 If you run out of beans,

set endHole to the last hole you dropped a bean.

set endSide to North.

return true.

If you still have beans, start again from the north side, but start dropping at the

rightmost hole instead of the hole passed in the parameter.

Else if it’s the south side sowing:

Remove the beans from that hole on the south side, returning false if there are no beans

While you still have beans:

Starting from the hole passed in the parameter, drop beans counterclockwise   
 (rightwards) on the South side of the board.

If you run out of beans,

set endHole to the last hole you dropped a bean.

set endSide to South.

return true.

If you run out of holes on the South side and still have beans:

Drop a bean in the South pot.

If that was your last bean:

set endHole to the pot.

set endSide to South.

return true.

If you still have beans:

Starting from the rightmost hole on the north’s side, drop beans   
 counterclockwise (rightwards) on the north side of the board   
 If you run out of beans,

set endHole to the last hole you dropped a bean.

set endSide to North.

return true.

If you still have beans, start again from the leftmost hole on South’s side.

}

Player

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

Create a JumpyTimer that checks the time one thousandth times as frequently as a normal Timer

Set a variable timeLimit to the constant MAX\_TIME.

Declare the variables bestHole and value to track results.

Initialize the variable depth to 0, our starting depth.

Call the helping function chooseMoveHelper with the board and side passed as parameters,

And the rest of the arguments being the corresponding variables we just declared.

Return the variable bestHole, which stores the result of the best choice from chooseMoveHelper

}

SmartPlayer::chooseMoveHelper(const Board& b, Side s, int& bestHole, int& value, int currDepth, double limit, JumpyTimer& timer) const {

If there are no beans in play, we’ve reached the max depth, or we’ve reached the max time:

Set the bestHole to -1

Call the evaluate function on the current board and set it’s result to value

Return;

For each of the holes on the board side,

Start the timer to see how much time has passed for the current branch

Split the allotted time limit by how many holes we need to explore so they each get

a fairly even amount of time of exploration

If there are no beans in the current hole, move to the next hole.

Once you reach a hole with beans, keep track of what hole it is.

Create a copy of the current board

Attempt to sow the current board, using the turnFinished function to see if we ended in

the pot (get another turn) or not

If we do get another turn:

Perform a recursive call with the new board

If we do not get another turn:

Perform a recursive call with the new board, opponent, and increase the

current depth

If the hole we are currently looking at is the first we’ve looked at,

Set bestHole to our current hole and value to the newValue stored after our

recursive calls for the first hole have finished.

Compare if the newValue we obtained from the recursive calls is better than our

previous best value. If so, assign bestHole to the current hole and value

to the current new value.

Adjust the timer by setting the time limit to (current time limit) - (how much time was elapsed).

If the our time limit is less than or equal to zero,

Set the time limit to zero.

}

SmartPlayer::turnFinished(Side s, int hole, Board& b, Side& endSide, int& endHole) const {

Sow the beans of the board passed as a parameter.

If we end in the pot, return false.

If we are able to perform a capture, perform the capture and return true.

Otherwise, return true.

}

SmartPlayer::evaluate(const Board& b) const {

Calculate the bean difference by subtracting the number of beans in the north pot by the

number of beans in the south pot.

If there is a tie (bean difference is zero), return 0.

If either side has the majority:

If the difference is positive, return 9999 because it means north has the majority.

If the difference is negative, return -9999 because it means south has the majority.

Return the bean difference.

}

Game

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

If there are no beans in play for both sides,

Set over to true.

If the north pot has the same number of beans as the south pot, set hasWinner to false.

Else, set hasWinner to true.

If there are more beans in the north pot,

Set the winner to north.

Else (south pot has more beans)

Set the winner to south.

Else, set over to false.

}

Game::move(Side s) {

If the game is over (no more beans in play on either side), return false.

Set the variables endside to the current side and endhole to the pot.

While the endside is our current side and the endhole is the pot,

If the current side has no beans in play,

Sweep all beans into the opponent’s pot.

Return false.

Have the current side’s player select a move using chooseMove.

Print this move onto the screen.

Sow the board with the selected move as the hole choice.

Display the current board.

If a capture is possible:

Complete the capture and display the resulting board.

Set the current side to the opposite side.

Return true.

}

Game::play() {

While the game is not over:

Call the move function. If it returns false:

Set over to true.

Call the status function to get the stats of the game.

If both players are not interactive,

Wait until enter is pressed to continue playing the game.

Call the status function to check if the game is over yet.

When the game is over,

Print out the total number of beans in each pot.

If there is a winner:

Print out the winner of the game using the hasWinner and winner variables.

Else, print out that it is a tie.

}

1. **A note about any known bugs, serious inefficiencies, or notable problems you had.**

The only main notable inefficiency my program has as far as I am concerned is the repetitiveness of the maximum depth and maximum time as limits to SmartPlayer::chooseMove. The maximum depth is set as a decently large number (in comparison to a 6-hole board) and limited by the maximum time, but it would probably be a little more efficient for the function to continue to run past the maximum depth if there is extra time in order to ensure the best move possible is returned.

1. **A list of the test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests. Even if you do not correctly implement all the functions, you can still list test cases that would test them. Don't lose points by thinking "Well, I didn't implement this function, so I won't bother saying how I would have tested it if I had implemented it."\**

To test my code, I used the testing functions provided by the project’s spec, as well as a few that I wrote on my own. I also heavily playtested my code, which I also made functions for so I could test the different combinations of players. In these functions, I also manipulated the board size and max\_depth variable of the SmartPlayer class to test for different types of games. Below is the testing code from my Main.cpp, with comments indicating the purposes of the tests.

void boardTests()

{

Board c(4, 2); // creating a new board

// testing getter functions

assert(c.totalBeans() == 16);

assert(c.beansInPlay(NORTH) == 8);

assert(c.beansInPlay(SOUTH) == 8);

assert(c.beans(NORTH, 1) == 2);

// testing setter functions

assert(c.setBeans(NORTH, 3, 0));

assert(c.setBeans(NORTH, 12, 4) == false);

assert(c.setBeans(SOUTH, 3, -23) == false);

// making sure beans changed after being set

assert(c.totalBeans() == 16 - 2);

assert(c.beansInPlay(NORTH) == 8 - 2);

assert(c.beans(NORTH, 3) == 0);

// testing sow function

Side endSide = NORTH;

int endHole = -999;

assert(c.sow(NORTH, 3, endSide, endHole) == false);

assert(endHole == -999 && endSide == Side::NORTH);

assert(c.sow(SOUTH, 4, endSide, endHole));

assert(endHole == 4 && endSide == NORTH);

// testing moveToPot

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

assert(c.beans(NORTH, POT) == 2);

assert(c.beans(NORTH, 1) == 0);

cout << "board tests passed" << endl;

}

void playerTests()

{

// creating players

HumanPlayer john("John");

SmartPlayer faust("Faust");

BadPlayer noob("Noob");

Player\* player1 = &john;

Player\* player2 = &faust;

Player\* player3 = &noob;

// checking name()

assert(player1->name() == "John");

assert(player2->name() == "Faust");

assert(player3->name() == "Noob");

// checking interactivity

assert(john.isInteractive());

assert(noob.isInteractive() == false && faust.isInteractive() == false);

Board b(3, 4);

// testing BadPlayer

assert(noob.chooseMove(b, NORTH) == 1); // chooses hole with largest num of beans

b.setBeans(NORTH, 2, 5); // change hole to have more beans

assert(noob.chooseMove(b, NORTH) == 2); // checking it picks the hole with more beans

// checking it picks the sooner hole if there are equal values

b.setBeans(NORTH, 3, 5);

assert(noob.chooseMove(b, NORTH) == 2);

b.setBeans(NORTH, 3, 7);

assert(noob.chooseMove(b, NORTH) == 3);

// checking that if it cannot pick any of the holes anymore that it doesn't do invalid moves

b.setBeans(SOUTH, 1, 0);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 3, 0);

assert(noob.chooseMove(b, SOUTH) == -1);

// testing HumanPlayer

b.setBeans(SOUTH, 1, 3);

assert(john.chooseMove(b, SOUTH));

// For input testing for the human player:

// selecting 2 or 3 should result in a display that there are no more beans

// selecting 4 or 0 should result in a display that the hole is out of bounds

// selecting 1 should work properly.

}

// playing between a BadPlayer and Human

void BadHumanPlayerTest()

{

HumanPlayer john("John");

BadPlayer noob("Noob");

Board b2(4, 3);

Game game1(b2, &john, &noob);

game1.play();

}

// Playing between a bad and smart player. SmartPlayer should always win

// Also tests to make sure the enter to continue implementation works

void BadSmartPlayerTest()

{

SmartPlayer smartie("Bot");

BadPlayer noob("Noob");

Board b(6, 4);

Game g(b, &noob, &smartie);

g.play();

}

// Playing with a human and SmartPlayer. It should be very difficult to beat the computer

void SmartHumanPlayerTest()

{

SmartPlayer smartie("Smart Bot");

HumanPlayer player("Player");

Board b(3, 4);

Game g(b, &smartie, &player);

g.play();

}

// Two human players

void TwoHumanPlayers()

{

HumanPlayer player1("P1");

HumanPlayer player2("P2");

Board b(3, 4);

Game g(b, &player1, &player2);

g.play();

}

// Game between two smart players!

void TwoSmartPlayers()

{

SmartPlayer smartie("Smart Bot");

SmartPlayer player("Radiant");

Board b(3, 4);

Game g(b, &smartie, &player);

g.play();

}

// between two bad players

void TwoBadPlayers()

{

BadPlayer noob1("Iron");

BadPlayer noob2("Noob");

Board b(3, 4);

Game g(b, &noob1, &noob2);

g.play();

}