Raymond Chen

David Smallberg

CS32 Spring

Project 3 Writeup

**General Design Overview**

My primary way of designing the Kalah game is using two vectors, one to represent the SOUTH player’s pot and holes and the other to represent the NORTH player’s pot and holes. I decided to use vectors as it is dynamic thereby allowing me to resize it to whatever size I wanted to depending on the number of holes the caller inputted. I also didn’t need to worry about any sort of dynamic memory, implementing my own copy constructors, or assignment operators as vectors are from the built in C++ library. I constructed each vector as 1 size greater than the number of holes for each side to account for the pot. Having the 0th index represent the POT also made things easier as I could use POT interchangeably, overall better the readability and allowing me to get a better understanding of what I had to implement. I created a private member variable “m\_turn” in my Game class which was extremely important for me to be able to keep track of the turns of players throughout the game. This allowed me to know when to end the game as a game is over only when it is Side s’s turn and s has no beans on their side. I also had three private helper functions in my Player class to help with the implementation of chooseMove. I will elaborate on this in the next section.

**SmartPlayer::chooseMove Implementation**

First I needed to create an evaluation function that would return an int type to represent my heuristic. int SmartPlayer::eval(const Board& b) const returned the difference between the number of beans in SOUTH’s pot and NORTH’s pot. Thus, a positive value is is a good heuristic for South player while the negative value is a good heuristic for North Player. I created another helper function called makeSmartMove, which is just the minimax algorithm. I started off by checking if the beansInPlay(s) == 0 which would indicate whether or not the game is over. If this condition is met, we will set bestHole to -1 and then call the eval function to determine which side won the game. If the eval function returns a positive value, then that means SOUTH won so we would set the heuristic to INT\_MAX if the SmartPlayer is South. If the eval function returns a negative value, then that means NORTH won so we would set the heuristic to INT\_MIN if the SmartPlayer is North. We then have a stopping condition to check if the depth hit 0. If none of those base cases are met, I run a for loop that finds the first hole with available beans to sow and store that value. This ensures that there is at least one legal move SmartPlayer will make so that -1 will never be chosen. We then iterate through each of the holes on the respective side and for the holes with beans, we will create a copy of the board and make moves on the copy of the board to see whether or not the move is good. We determine the smartness of a move by comparing the heuristics. If we find a better heuristic (remember a good heuristic depends on which side SmartPlayer is) then we assign it to our heuristic variable. I implemented a function called sowANDcap to perform a sow and a capture. The capture functionality is implemented in the actual game class which we don’t access in smartPlayer so I create a sowANDcap functions to also determine which moves can give us a capture. This allows the smartPlayer to have a better understanding of which moves can be more favorable.

**Pseudocode for NonTrivial Functions**

Board class

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

{  
Check if the chosen hole is invalid, return false if the hole is invalid

Check if the hole that we want to sow has any beans in it, if empty then we return false

Empty the hole that we selected and store it somewhere to keep track of the number of beans to sow

while beanstoSow > 0

Check for the current side and curenthole you are at, move the beans in counterclock wise direction

Skip over enemies pots

Account for cases where you sow from a hole to a pot or from one hole to another hole on the other side

Otherwise if the side is north, iterate through the vector backwards

Otherwise if the side is south, iterate through the vector forwards

When the loop is done running

Set the endSide and endHoles accordingly

Return true

Game class

bool Game::move(Side s)

{  
if the beans in play on the respective side is 0

Sweep the beans of the opponent into their pot

Return false

If s == SOUTH

Store the hole that the player wants to sow and pass it into the sow function

If the endHole is s’s pot then extra move can be made

Call move recursively

if the sow ended on the side of s and if the endHole only has 1 beans and if the beans in the hole directly across is 0

Make a capture

Change the turn to NORTH

If s == NORTH

Store the hole that the player wants to sow and pass it into the sow function

If the endHole is s’s pot then extra move can be made

Call move recursively

if the sow ended on the side of s and if the endHole only has 1 beans and if the beans in the hole directly across is 0

Make a capture

Change the turn to SOUTH

void SmartPlayer::makeSmartMove(const Board& b, Side s, int depth, int& bestHole, int& heuristic) const

void makeSmartMove(in: player, board; out: bestHole, value):

if no move for player exists (i.e., game is over),

bestHole = -1

If side is SOUTH and difference in beans in positive

value of the position is high

If side is NORTH and difference in beans in negative

value of the position is high

If the difference in beans is 0

There is a tie

return

if the depth has reached 0

bestHole = -1

value = value of this position

Return

for every hole available

Find the first hole that has beans and store it as the firstPossible value

(prevents smartPlayer from choosing -1 if we cannot find a favorable move)

for every hole h the player can choose

"make" the move h with a copy of the board

If an additional move can be made

makeSmartMove(boardCopy, s, depth -1, h2, v2)

Otherwise

makeSmartMove(boardCopy, opponent(s), depth -1, h2, v2)

if v2 is better for the player than best seen so far,

bestHole = h

value = v2

Return

**Bugs and Problems**

One major problem I had was that my smartPlayer would sometimes return -1 as a move when it got deeper into the game, despite there being available holes to sow. I fixed this by iterating through all the holes on the respective side of the smartPlayer and saving the index of the first hole that had beans in it. We later assign it to bestHole, and if the heuristics being compared are better, then the bestHole is overridden. In the worst case scenario where the heuristics being compared are not better, we have the index of the first hole with available beans to act as a safety/next best move available. This way, smartPlayer never chooses -1 which is illegal.

Another problem I had was that my SmartPlayer would not chooses the hole that gave them an additional turn on the first move. I fixed this by setting bestHole to whatever index it was that gave an additional move when I “made” a move on the copy of the board. Then I would check the heuristic of the additional move that was made and if that had a good enough heuristic then that meant choosing the hole that gives another move is a good move.

**Test Cases**

I just created more test cases in the same layout provided by the spec.

Board b1(4, 3);

assert(b1.holes() == 4 && b1.totalBeans() == 24 &&

b1.beans(SOUTH, POT) == 0 && b1.beansInPlay(SOUTH) == 12);

b1.setBeans(SOUTH, 2, 5);

b1.moveToPot(SOUTH, 3, SOUTH);

assert(b1.totalBeans() == 23 && b1.beans(SOUTH, 2) == 5 &&

b1.beans(SOUTH, 3) == 0 && b1.beans(SOUTH, POT) == 3 &&

b1.beansInPlay(SOUTH) == 9);

Side es1;

int eh1;

b1.sow(SOUTH, 4, es1, eh1);

assert(es1 == NORTH && eh1 == 4 && b1.beans(SOUTH, 4) == 0 &&

b1.beans(NORTH, 4) == 4 && b1.beans(SOUTH, POT) == 4 &&

b1.beansInPlay(SOUTH) == 5 && b1.beansInPlay(NORTH) == 8);

Board b2(2, 4);

assert(b2.holes() == 2 && b2.totalBeans() == 12 &&

b2.beans(SOUTH, POT) == 0 && b2.beansInPlay(SOUTH) == 6);

b2.setBeans(SOUTH, 1, 2);

b2.moveToPot(SOUTH, 2, SOUTH);

assert(b2.totalBeans() == 11 && b2.beans(SOUTH, 1) == 2 &&

b2.beans(SOUTH, 2) == 0 && b2.beans(SOUTH, POT) == 4 &&

b2.beansInPlay(SOUTH) == 2);

Side es2;

int eh2;

b2.sow(SOUTH, 1, es2, eh2);

assert(es2 == NORTH && eh2 == 2 && b2.beans(SOUTH, 1) == 0 &&

b2.beans(NORTH, 2) == 3 && b2.beans(SOUTH, POT) == 4 &&

b2.beansInPlay(SOUTH) == 1 && b2.beansInPlay(NORTH) == 7);

These are just some tests I ran to ensure that the sow function did the right thing and would the getter functions for the beans in a particular hole, pot, or entire side returned the right values. For example it was important to make sure that sow would skip over the opponents pot.

HumanPlayer hp3("RAY");

assert(hp3.name() == "RAY" && hp3.isInteractive());

BadPlayer bp3("greg");

assert(bp3.name() == "greg" && !bp3.isInteractive());

SmartPlayer sp3("AI");

assert(sp3.name() == "AI" && !sp3.isInteractive());

Board b3(5, 6);

b3.setBeans(SOUTH, 3, 3);

int n7 = hp3.chooseMove(b3, SOUTH);

assert(n7 == 2 || n7 == 4);

int n8 = bp3.chooseMove(b3, SOUTH);

assert(n8 == 2 || n8 == 4);

int n9 = sp3.chooseMove(b3, SOUTH);

assert(n9 == 2 || n9 == 4);

This is to check that the the players are choosing the expected moves and to ensure that the interactive function returns the proper bools.

void doGameTests()

{

BadPlayer player1("Alice");

BadPlayer player2("Bob");

Board board(4, 0);

board.setBeans(SOUTH, 1, 3);

board.setBeans(NORTH, 2, 2);

board.setBeans(NORTH, 3, 1);

board.setBeans(NORTH, 4, 4);

Game game(board, &player1, &player2);

bool over;

bool hasWinner;

Side winner;

// Bob

// 0 1 2 3

// 0 0

// 3 0 1 4

// Alice

game.status(over, hasWinner, winner);

assert(!over && game.beans(NORTH, POT) == 0 && game.beans(SOUTH, POT) == 0 &&

game.beans(NORTH, 1) == 0 && game.beans(NORTH, 2) == 2 && game.beans(NORTH, 3) == 1 && game.beans(NORTH, 4) == 4 &&

game.beans(SOUTH, 1) == 3 && game.beans(SOUTH, 2) == 0 && game.beans(SOUTH, 3) == 0 && game.beans(SOUTH, 4) == 0);

game.move(SOUTH);

// 0 1 2 3

// 0 0

// 0 1 2 5

// Alice

game.status(over, hasWinner, winner);

assert(!over && game.beans(NORTH, POT) == 0 && game.beans(SOUTH, POT) == 0 &&

game.beans(NORTH, 1) == 0 && game.beans(NORTH, 2) == 2 && game.beans(NORTH, 3) == 1 && game.beans(NORTH, 4) == 5 &&

game.beans(SOUTH, 1) == 0 && game.beans(SOUTH, 2) == 1 && game.beans(SOUTH, 3) == 2 && game.beans(SOUTH, 4) == 0);

game.move(NORTH);

// 1 0 2 3

// 0 0

// 0 1 2 5

// Alice

game.status(over, hasWinner, winner);

assert(!over && game.beans(NORTH, POT) == 0 && game.beans(SOUTH, POT) == 0 &&

game.beans(NORTH, 1) == 1 && game.beans(NORTH, 2) == 0 && game.beans(NORTH, 3) == 2 && game.beans(NORTH, 4) == 5 &&

game.beans(SOUTH, 1) == 0 && game.beans(SOUTH, 2) == 1 && game.beans(SOUTH, 3) == 2 && game.beans(SOUTH, 4) == 0);

game.move(SOUTH);

// 1 0 2 3

// 0 0

// 0 0 3 6

// Alice

game.status(over, hasWinner, winner);

assert(!over && game.beans(NORTH, POT) == 0 && game.beans(SOUTH, POT) == 0

These are tests I ran to make sure that invalid moves are rejected by the game and proper moves are being made.

Testing smartPlayer:

To ensure that smart player did not make illegal moves: I played the game many times with smartPlayer and checked to see if -1 was chosen as as move at any point. I played smartPlayer against myself to see how many times it would win. I also played smartPlayer against a badPlayer to ensure that it would win every time. I printed each round of the game and looked to see if smartPlayer actually made a favorable move (making moves that get them an additional turn or gives a capture). South also consistently wins when two smartPlayers go against each other.