**Report**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**A description of my design:**

**[Design of type: Hole]**

For the implementation of the whole Kalah game, I first created a **new class of Holes**. For each hole, it has (as private members) **a side, a sequence number, a number of beans inside and a pointer that points to the next hole** (which links these holes together).

With the public-interface functions that can access these private members and return or modify them, other types (such as Board etc.) can easily make use of these holes for the implementation of the whole game.

**[Design of type: Board]**

For the implementation of the class Board, in addition from the integer private members that indicate the number of holes and initial beans inside, **I added two vectors of pointers pointing to type Hole to store the South holes and North holes**.

I chose to use the data structure (STL) of **vector**, because it can easily add new items, and access them just using brackets and indexes ([X-a number]).

However, because the elements in the vector are pointers to newly allocated Holes, I have to **delete them** before clearing up the vector in the **destructor**. Similarly, I have to modify the **copy constructor and the assignment operator** to deal with copying the contents of the pointers.

**[Design of type: Game]**

For the implementation of the class Game’s private field, I added **two Player pointers** that points two a South player and a North player (that are passed in as parameters in the constructor interface).

**[Design of type: Player]**

There are three kinds of Players in total: the Human Player, the Bad Player, and the Smart Player. These three classes are all derived (inherited) from the base class Player. For each of these derived Players, they have their own implementation of public functions: isInteractive and chooseMove, which fits their own characteristics.

thatFor the Smart Player class, I added and implemented these three additional private functions:

* int evaluate(const Board& b, Side s) const: this function takes in a board and the side that we want to maximize, and calculate a value for this stage of game. (a larger value means the stage is better for the maximizer, which is indicated by the Side s).
* int minimax(Board&b, int depth, Side s, Side mySide) const: this function takes in a board, a integer value depth (which is just a counter of how many recursions we go through), a side (the side that is playing this turn), and side mySide (the maximizer’s side). This function considers all the possible moves this stage of game can result it (with a depth level that limits it from going to far), and returns the value of the board.
* void makeCompleteMove(Board& board, Side side, int hole) const: this function takes in a board, the side that moves, and the hole it sows from. This function allows a complete move (including extra turns and captures), that helps the implementation of the chooseMove function of the Smart Player.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**A description of SmartPlayer::chooseMove**

For the implementation of the **chooseMe** function of the Smart Player class, I utilized the three private member functions that I added to the Smart Player class.

**Evaluation function**: this function, when given a board state and maximizer side, can evaluate how good this state is for the maximizer, giving a large value if the maximizer’s side is in advantage. For evaluation, I used these heuristics:

* **the number of beans in maximizer’s pot minus that in opponent’s pot**: the goal of this game is to gain as many beans in pot as possible, so we will want to find the state where we have the most beans in our pot while the opponent has the least in theirs (i.e. the difference of the number of beans in pots). Because this value is crucial for winning the game, **I emphasize its impact (weight) by timing this value by 10** for the value returned by the evaluation function.
* **the number of beans in maximizer’s holes minus that in opponent’s holes**: if we make sure that for each possible state, we can have the most beans on our side and the least on opponent’s side, then when the game ends and all beans are swept into corresponding pots, we will win. However, this is only a trivial goal as the number of beans on one side does not really decide who will win in the end, so **I did not give it any significant weighing**.
* **If the game is over, did the maximizer win, or the opponent wins**: if at a certain point when the simulated game is over, we need to check who wins. Since the maximizer winning is the best possible scenario, I give it the utmost weighing by setting the value to 9999; similarly, I will set the value to -9999 if the minimizer wins, as that is the worst case we can possibly encounter.

**Minimax function**: this function, when passed in a board, will consider all possible ways the game can go (without passing the depth set), evaluate each scenario, and return the final value of the board (basing on the maximizer).

**makeCompleteMove function:** this function, when passed in a board, a side that moves, and the hole to move from, makes a complete move. While game is not over, it sows from that hole, and considers whether it takes another additional turn or if there is a capture. If it should take another turn, this function simulates this additional turn by recursively calling itself, find the best additional move possible, and make that move.

**chooseMove function:** this is SmartPlayer’s public member function. This function first makes a copy of the board passed in (so that it won’t alter the original one), and then considers each valid hole. For each valid hole, it simulates a move, and then calls the minimax function to go through all possible scenarios for this move, which returns an evaluated number. After the evaluation, the function “unmakes” the move to prepare for the next hole it considers. Based on this number value, chooseMove function will find the best hole which gives the best number (thus is the best scenario), and choose to return this hole.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Pseudocode for non-trivial algorithms:**

**[Major Algorithms for type: Board]**

Board::Board(int nHoles, int nInitialBeansPerHole) // Constructor

Set number of holes and initial beans per hole to the parameters (validate them)

Create and allocate new South and North holes in corresponding vectors

Set up the next pointers for South pot and North pot

For all South holes

Set up their next pointers according to the numbering paradigm

For all North holes

Set up their next pointers according to the numbering paradigm

Board::Board(const Board& other) // Copy Constructor

Set number of holes and initial beans per hole to the same as the other Board (validate them)

Create and allocate new South and North holes in corresponding vectors

Set up the next pointers for South pot and North pot

For all South holes

Set up their next pointers according to the numbering paradigm

Give these holes the same number of beans as the corresponding hole in other Board

For all North holes

Set up their next pointers according to the numbering paradigm

Give these holes the same number of beans as the corresponding hole in other Board

Board& Board::operator = (const Board& other) // Assignment Operator

for all South holes and pot

delete it

for all North holes and pot

delete it

clear the South hole vector

clear the North hole vector

**[The rest is basically the same as the Copy Constructor]**

Board::~Board() // Destructor

for all South holes and pot

delete it

for all North holes and pot

delete it

clear the South hole vector

clear the North hole vector

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

if the hole input is invalid or a pot or empty

return false

store the number of beans removed

remove the beans from this hole

Set a Hole pointer that points to the hole to be sowed (to be added a bean)

For each bean removed

If the hole pointer points to the opponent’s pot

Move the pointer to the next hole

Continue

Add one bean to the hole the pointer points to

Move the pointer to the next hole

If the hole pointer points to the opponent’s pot

Move the pointer to the next hole

Add one bean to the hole the pointer points to

Record and return the endSide

Record and return the endHole

Return true

**[Major Algorithms for type: Game]**

bool Game::move()

if the game is over

return false

the player chooses a move

while the last bean is placed in own hole (an extra turn) and game is not over

display the board

the player chooses a move

if the last bean is placed in own hole and is previously empty (capture)

if the hole opposite to this hole is not empty

move this last bean and all beans in opposite hole to this player’s pot

exchange the player who should go next turn

display the board

if the game is over

sweep all beans in holes to their corresponding side’s pot

display the board

void Game::play()

check whether the two players are interactive

if two computer players playing

display the board and information

while game is not over

prompt for ENTER input

player make its move

check the game’s status

display winning information

else

display the board and information

while game is not over

player make its move

check the game’s status

display winning information

**[Major Algorithms for type: Player]**

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

if no move possible

return -1

make a copy of the board passed in so we won’t alter the original one

for each valid hole

“make” a move on this hole

Compute the value of the board state (using evaluation function)

“unmake” the move

If the value is the best value possible

Update the best value

Update the best hole possible

Return the best hole possible

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

int returnValue = 0

returnValue += 10\*(number of beans in s’s pot – number of beans in opponent’s hole)

returnValue += number of beans in all s’s holes

returnValue -= number of beans in all opponent’s holes

return returnValue

int SmartPlayer::minimax(Board&b, int depth, Side s, Side mySide) const

make a copy of the board

calculate the score of this board stage (using evaluation function)

if depth > 3

return score

if the game is over

if maximizer wins

score += 9999

if minimizer wins

score -= 9999

return score

if the side of player’s turn is my side

for each valid hole on the side

“make” a complete move

Call minimax recursively and choose the maximum value

“un-make” the move

Return the max value

Else

for each valid hole on the side

“make” a complete move

Call minimax recursively and choose the minimum value

“un-make” the move

Return the min value

void SmartPlayer::makeCompleteMove(Board& board, Side side, int hole) const

if the game is over

return

do

if this is an extra turn

for each hole on the side of the board

“make the move”: makeCompleteMove (using recursion)

get the value of this move (using evaluate function)

if this value is the largest possible, record the hole of this move

“unmake” the move

Sow this hole

While game is not over and the last bean is placed in own pot

if the last bean is placed in own hole and is previously empty (capture)

if the hole opposite to this hole is not empty

move this last bean and all beans in opposite hole to this player’s pot

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**bugs, serious inefficiencies, or notable problems**

* **[Major Difficulty**]: I initially got stuck on how to organize the holes --- I know that each hole contains multiple elements, thus it should be represented as a data type (class). However, I cannot choose to use arrays (because I don’t know its size), so **I should choose a STL to use**. In the end, I chose the vector STL to store the pointers to newly allocated hole objects.
* **[Bug]:** When compiling the Game tests, the program crashed due to **repeatedly deleting the same object**. (Solved it by modifying the class Board’s copy constructor).
* **[Bug]:** After I finished the whole program, I run it on g++, but resulted in a lot of memory leaks. It turned out, when using the assignment operator to alter an existing Board, **I have to clean its content first**, or else the memory will leak.
* **[Major Difficulty]: One of the most significant difficulty is to organize the code implementation:** I know that I need a function that evaluates a board condition, and a minimax function that considers all possible moves with the previous function, and a choose move function that combines both previous two functions. However, how to organize them? With some thinking, I decided to divide these three functions and implement them in the private domain of the SmartPlayer class. Then, utilizing recursion and pass by reference, I can link them all together.
* **[Major Difficulty]:** For this game, we need to consider the scenario when a player gains an extra turn for putting the last bean into its own pot. I didn’t know how to evaluate for this scenario, since it’s the same side taking another full turn and evaluating from the start. Later, I put the implementation of making a full move in a separate function and recursively calls it with the evaluation function. In this way, the problem is solved.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**A list of test cases:**

**[Test Cases for type: Board]**

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doBoardTests()

{

/// Test Case: for a Normal Board//////////////////////////////////

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

Board a(5,4); // 5 Holes each side, 4 beans in each

assert(a.holes() == 5 && a.totalBeans() == 40 && // Check the number of holes on one side, and total beans in game

// Test: holes and totalBeans functions

a.beans(SOUTH, POT) == 0 && a.beans(NORTH, POT) == 0 // Check the pots for both sides are initialized empty

// Test: beans function

&& a.beansInPlay(SOUTH) == a.beansInPlay(NORTH)); // Test: beansInPlay function

// Test: setBeans function

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

assert(a.setBeans(NORTH, 2, 2));

assert(a.setBeans(SOUTH, 1, 7));

assert(!a.setBeans(NORTH, -1, 3)); // Invalid case: hole is invalid (too small)

assert(!a.setBeans(NORTH, 55, 3)); // Invalid case: hole is invalid (too big)

assert(!a.setBeans(NORTH, 3, -9)); // Invalid case: bean is negative

assert(a.setBeans(SOUTH, 0, 4));// Special case: setting a pot

for (int i = 3; i < 6; i++)

{

a.setBeans(NORTH, i, 3);

a.setBeans(SOUTH, i, 1);

}

assert(a.beans(NORTH, 3) == 3 && a.beans(NORTH, 1) == 4 && a.beans(NORTH, 0) == 0 && a.beans(NORTH, 5) == 3);

assert(a.beans(SOUTH, 3) == 1 && a.beans(SOUTH, 1) == 7 && a.beans(SOUTH, 0) == 4 && a.beans(SOUTH, 5) == 1);

// Test: moveToPot function

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

assert(a.moveToPot(NORTH, 2, NORTH));

assert(a.moveToPot(SOUTH, 3, NORTH));

assert(a.moveToPot(NORTH, 4, SOUTH));

assert(!a.moveToPot(NORTH, -3, SOUTH));// Invalid case: hole is invalid (too small)

assert(!a.moveToPot(NORTH, 53, SOUTH));// Invalid case: hole is invalid (too big)

assert(!a.moveToPot(NORTH, 0, SOUTH)); // Invalid case: hole is a pot

assert(a.beans(NORTH, 3) == 3 && a.beans(NORTH, 1) == 4 && a.beans(NORTH, 0) == 3 && a.beans(NORTH, 2) == 0);

assert(a.beans(SOUTH, 3) == 0 && a.beans(SOUTH, 1) == 7 && a.beans(SOUTH, 0) == 7 && a.beans(SOUTH, 2) == 4);

assert(a.beansInPlay(NORTH) == 10 && a.beansInPlay(SOUTH) == 13);// Test: beansInPlay function

// Test: sow function

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

Side endSide;

int endHole;

assert(!a.sow(NORTH, 2, endSide, endHole));// Invalid case: hole is empty

assert(!a.sow(NORTH, -5, endSide, endHole));// Invalid case: hole is invalid

assert(!a.sow(NORTH, 65, endSide, endHole));// Invalid case: hole is invalid

assert(!a.sow(NORTH, 0, endSide, endHole));// Invalid case: hole is pot

assert(a.sow(NORTH,1,endSide,endHole)); // Normal sowing

// 0 0 3 0 3

//4 7

// 8 5 1 1 1

assert(endSide == SOUTH && endHole == 3);

assert(a.beans(NORTH, 1) == 0 && a.beans(NORTH, 0) == 4 && a.beans(SOUTH, 3) == 1);

assert(a.beansInPlay(NORTH) == 6 && a.beansInPlay(SOUTH) == 16);

assert(a.setBeans(SOUTH, 5, 8));

assert(a.sow(SOUTH, 5, endSide, endHole)); // Special sowing: passing the opponent's pot

// 1 1 4 1 4

//4 8

// 9 6 1 1 0

assert(endSide == SOUTH && endHole == 2);

assert(a.beans(NORTH, 1) == 1 && a.beans(NORTH, 0) == 4 && a.beans(SOUTH, 3) == 1);

assert(a.beans(NORTH, 3) == 4 && a.beans(NORTH, 5) == 4 && a.beans(SOUTH, 1) == 9);

assert(a.beansInPlay(NORTH) == 11 && a.beansInPlay(SOUTH) == 17);

assert(a.totalBeans() == 40);// Test: totalBeans function

/// Test Case: for a Special Board//////////////////////////////////

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

Board b(0, -9);//If nHoles is not positive, act as if it were 1; if nInitialBeansPerHole is negative, act as if it were 0.

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

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

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

assert(b.beans(NORTH, 0) == 0 && b.beans(SOUTH, 0) == 0);

}

int main()

{

doBoardTests();

cout << "Passed all tests!" << endl;

}

**[Test Cases for type: Player]**

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doPlayerTests()

{

//// Player test cases in the spec

cout << "Test cases in the spec:" << endl;

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);

cout << "Newly added test cases:" << endl;

////////// Some test cases I added

HumanPlayer human("Brandon"); // Test: constructor

assert(human.name() == "Brandon" && human.isInteractive());// Test: name and isInteractive functions

HumanPlayer human2(""); // Test: constructor (special case: name is empty)

assert(human2.name() == "" && human2.isInteractive());// Test: name and isInteractive functions

BadPlayer Bad("");// Test: constructor (special case: name is empty)

assert(Bad.name() == "" && !Bad.isInteractive());

SmartPlayer Smart("");// Test: constructor (special case: name is empty)

assert(Smart.name() == "" && !Smart.isInteractive());

// Test: setBeans and constructor for Board type

Board a(4, 0);

a.setBeans(SOUTH, 1, 3);

a.setBeans(SOUTH, 4, 4);

a.setBeans(NORTH, 2, 3);

// Test: chooseMove function for human type

cout << "=========" << endl;

cout << "Only 1 and 4 should work. Test this." << endl;

int n1 = human2.chooseMove(a, SOUTH); // Test: chooseMove function

cout << "=========" << endl;

assert(n1 == 1 || n1 == 4);

// Test: chooseMove function for non-human types

n1 = Bad.chooseMove(a, NORTH);

assert(n1 == 2);

n1 = Smart.chooseMove(a, NORTH);

assert(n1 == 2);

n1 = Smart.chooseMove(a, SOUTH);

assert(n1 == 1 || n1 == 4);

n1 = Bad.chooseMove(a, SOUTH);

assert(n1 == 1 || n1 == 4);

}

int main()

{

doPlayerTests();

cout << "Passed all tests" << endl;

}

**[Test Cases for type: Game]**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doGameTests()

{

// Test: Player and Board constructors and setBeans function

BadPlayer bp1("Bad Player 1");

BadPlayer bp2("Bad Player 2");

Board b1(4, 2);

b1.setBeans(SOUTH, 1, 3);

b1.setBeans(SOUTH, 3, 0);

b1.setBeans(NORTH, 2, 0);

b1.setBeans(NORTH, 3, 4);

Game g(b1, &bp1, &bp2); // Test: Game constructor

bool over;

bool hasWinner;

Side winner;

// Bad Player 1

// 2 0 4 2

// 0 0

// 3 2 0 2

// Bad Player 2

g.display();// Test: display function (should have reasonable display of information)

g.status(over, hasWinner, winner); // Test: status function

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

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

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

g.move(); // Test: move function (Bad Player 2 makes move)

// Bad Player 1

// 2 0 4 2

// 0 0

// 0 3 1 3

// Bad Player 2

g.status(over, hasWinner, winner); // Test: status function

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

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

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

g.move(); // Test: move function (Bad Player 1 makes move)

// Bad Player 1

// 0 0 4 2

// 1 0

// 1 3 1 3

// Bad Player 2

g.status(over, hasWinner, winner); // Test: status function

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

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

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

// Test: beans function (with invalid case included that should return -1)

assert(g.beans(NORTH, 0) == 1 && g.beans(NORTH, 5) == -1 && g.beans(SOUTH, 0) == 0 && g.beans(SOUTH, 2) == 3);

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

g.play(); // Test: function play

// Let the game finish on its own (should have reasonable output that waits for the input ENTER of the user)

assert(g.beans(NORTH, 0) == 12 && g.beans(SOUTH, 0) == 3 && g.beans(SOUTH, 2) == 0);

}

int main()

{

doGameTests();

cout << "Passed all tests" << endl;

}

**[Additional Test Case: SmartPlayer]**

// This is an additional test case that specifically tests the Smart Player. Since the Smart Player has the same public interfaces as any other players, this test case does not repeatedly test the same functions. Instead, this test case is only to run the smart player and make sure that it is doing all things correctly: at least it must not choose any invalid holes; ideally, it will behave cleverly and try to win the game.

int main()

{

HumanPlayer hp("You");

SmartPlayer sp("Smart Computer"); // Test the SmartPlayer

Board board(5, 5);

Game game(board, &hp, &sp);

game.play(); // Simulate a game

cerr << "Passed all tests" << endl; // If all things work out correctly without

// any bugs and errors, test case succeed.

}

**[Additional Test Case: Game]**

// If when this function is called, South has no beans in play, so can't make the first move, sweep any beans on the North side into North's pot and act as if the game is thus over.

int main()

{

Board board(3, 3);

board.setBeans(SOUTH, 1, 0);

board.setBeans(SOUTH, 2, 0);

board.setBeans(SOUTH, 3, 0);

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Game g(board, &bp1, &bp2);

g.play();

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_