**Project 3 Report**

* My member classes are almost the same as the declarations given in the spec sheet. The additional helper functions that I declared are: a minimax algorithm function (determines the best move for the player) and evaluatw function (determines how good or bad a state of a board is for the player) for SmartPlayer::chooseMove(); a swap function in Board that helps me write my the assignment operator function. The function basically swaps the pointer to the start of the linked lists (or in this case the pointers for the pots) and swaps the other data members like integers for number of holes and number of beans per hole.
* The major data structure that I used in my project was linked lists. I used nodes of a linked list as the pots and holes on the board. Each linked list had a beans integer that stores the number of beans the hole contained, a holenumber integer that stored the holenumber of the holes and pots, the side that the hole is at, and a next pointer that points to the next hole. I had two Hole pointers that pointed to the north and south pot. This allowed me to construct and traverse the linked lists for certain implementations.
* My SmartPlayer::chooseMove itself does not do any computation to determine the best hole. It calls a minimax function that I declared, which computes the best hole for the SmartPlayer to play. The minimax function after its recursive calls, calls another function called evaluate that evaluates the current state of the board and returns how good or bad the current state is. A positive return value means that the board is good for the player and greater that positive value, the better the state is. Similarly, a negative value means that the state of the board is not good and more the negative the value is, the worse the board is.

In terms of heuristics, my rule of thumb is “The greater the difference between the number of beans in my pot and the number in my opponent's, the better". That the only heuristics that is reflected in my evaluation function.

* Pseudocode

**Board::sow**

Return false if there are no beans in the indicated hole

For (until reach the hole indicated)

Traverse the linked list;

Set an integer to the number of beans in the hole and set the hole’s beans to 0.

Until the beans are not zero

From the hole indicated, add one bean at a time to the adjacent holes, but skipping the opponents pot.

When we are down to the last bean, and the bean lands in a hole that has 0 beans and the opponent.

Set the endside and endhole to the temporary pointer’s side and holenumber.

If the beans in on the starting side are 0 then sweep all the beans from the opposite side to its hole.

**BadPlayer::chooseMove**

If the beans in play on that side are 0 then return -1

Else

By default, choose the hole next to the pot if the beans are not 0

Else choose the next hole that has beans

**SmartPlayer::chooseMove**

Call the minimax function

Return the besthole that minimax function determined

**Minimax**

If beans in play are zero on the indicated side then determine the state of the board, put besthole as -1 and set value as the calculated value

If timedOut

Determine the state of the board and set value equal to the state of the board

For all the holes on the side of the board that the player can choose

Create a temporary board to make the moves

Call the evaluate function on the temp board

If the value is the best seen so far,

then set value to that value

set besthole to that hole

return

**Evaluate**

evaluate the state of the board based on the difference in the number holes in the player’s pot and the opponents pot

if it’s the player’s side,

evaluate the value based on the perspective of the player.

i.e. the more beans that the player has than the opponent, return a greater value

if it’s the opponent

evaluate based on the player’s perspective

i.e. the more beans that the opponent has than the player, then return a more negative value

**Game::move**

If the game is over

Return false

Else

If its south’s move

Play south’s move

If the last bean ended in pot

Call move without changing the turn

Set turn to north’s turn

If its north’s move

Play north’s move

If the last bean ended in pot

Call move without changing the turn

Set turn to south’s turn

Return true

**Game::play**

While the game is not over

Call the move function

Call the display function to display the current state of the board

If the game is over

Determine the winner and print out the winner’s name

* The most notable bug in my program is that the minimax function is sometimes caught in an infinite recursive call when the player gets another turn (i.e. when the last bean ends in the player’s pot). Another bug is that the minimax function does not return a hole within 5 secs with bigger boards – boards with around 8-10 holes.
* **Test Cases**

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void doBoardTests()

{

Board b(3, 2);

assert(b.holes() == 3 && b.totalBeans() == 12 &&

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

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);

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);

}

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();

// 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();

// 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();

// 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();

// 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);

}