1. I designed my Board class by a circular single linked list. First I constructed a Hole class in which each object has member variables of its index, its number of beans, its pointer to next Hole and its side. Then, I dynamically allocated two arrays of Holes, one for north and one for south. The board object has its two member variables (both pointers) to monitor each element of the array of Holes. Then since the game executes in counterclockwise direction, the direction of the south array is just what we want. However, we need to reversely assign the indices for north array. After that, construct two Holes for pots on both sides. Assign m\_next pointer of the last element of north array (with index 1) to point to the north pot, m\_next pointer of the north pot to point to the first element of south array, m\_next pointer of the last element of south array to point to the south pot, and m\_next pointer of the south pot to point to the first element of the north array. Then, a circular single linked list forms as intended.

For the Game class, I declared its member variables of a Board, two pointers to two players, one bool variable to monitor whether the game ends and one Side variable to monitor whose turn it is.

1. The heuristics that I intended to use is the difference between numbers of beans in two pots. I designed SmartPlayer::chooseMove() in a tree where each node has a board member and 20 pointers pointing to children nodes. If no moves are possible, return -1 and set the value (heuristics) to positive infinity, negative infinity or 0. However, if the heuristics disallows me to search nodes below, I would just return -1 and set value to the value of this position. For each hole, if the value (heuristics) that we get after making this move is a better value than not making this move, we just return the number of this position and set value to that value.
2. bool Sow(side, index, endSide, endHole)

If index is not a valid number, return false without changing anything

If index is valid, find the Hole whose HoleSide is side and HoleIndex is index by a for-loop.

Copy the number of beans in that Hole to another variable and assign the number of beans in that Hole to 0 to evacuate the Hole.

Add one to the number of beans of each Hole next to the previous one in counterclockwise direction until all beans have been distributed

Omit the pot of the other side if encountering.

For the Hole into which the last bean is put, assign its HoleSide to endSide and HoleIndex to endIndex.

Return

bool move()

If the game is over, return

If it’s south turn, choose a move according to respective algorithms for different kinds of players and execute sow() for south and the move, which is index for the argument of sow().

If after sowing no beans are in south side, end the game, return

If the last been is put into holes of north, then next turn will be north turn, return

If the number of bean in endHole is 1 and the opposite hole in the north is not empty, a capture happens, return

If the last bean is put into the south pot, additional turn happens. Additional turn could happen continuously.

After additional turns stop, retest whether it’s end of game or the last bean is put into north holes or a capture happens. The next turn must be north turn.

If it’s north turn, choose a move according to respective algorithms for different kinds of players and execute sow() for north and the move, which is index for the argument of sow().

If after sowing no beans are in north side, end the game, return

If the last been is put into holes of south, then next turn will be south turn, return

If the number of bean in endHole is 1 and the opposite hole in the south is not empty, a capture happens, return

If the last bean is put into the north pot, additional turn happens. Additional turn could happen continuously.

After additional turns stop, retest whether it’s end of game or the last bean is put into south holes or a capture happens. The next turn must be south turn.

1. When I implemented move() function for the first time, I failed to consider the situation that after sowing the beans of one side, there are no beans in play for that side so that the game ends. This makes move() function go into an infinite loop so that the game won’t ever end. Another problem is that I failed to distinguish the situations between one human player with one computer player and two computer players. If two players are both not interactive, I need to output some texts like “Press ENTER to continue”, and if one of them is interactive, I need to output some texts like “Select a move:”. This is the place where isInteractive() works. With the help of this function, I can do different things for various situations. The third problem is that I considered the capture situation before the additional turns situation so that if the last additional turn is a capture, I just omit it and it will be the turn of the other side. Therefore, I fixed this bug by considering the capture situation again after the while-loop of additional turns. Then, I will make a capture even though it’s not the first turn of certain side.
2. Test Cases:

#include "Game.h"

#include "Player.h"

#include "Board.h"

#include "Side.h"

#include <iostream>

#include <cassert>

using namespace std;

void Tests()

{

//Test the message printing and situations like combinations of additional turns and capture

HumanPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board a(3,2);

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

g.play();

//Test whether the program executes step by step instead of generating the result immediately

BadPlayer bp3("Tracy");

BadPlayer bp4("Xiaopei");

Board p(3, 2);

Game g(p, &bp3, &bp4);

g.play();

//Test the member functions and constructor of Board class

Board b(3,2);

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

b.beans(SOUTH, 0) == 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, 0) == 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, 0) == 3 &&

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

//Test the member functions and constructor of Player class

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 d(3,2);

d.setBeans(SOUTH, 2, 0);

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

int n = hp.chooseMove(d, SOUTH);

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

assert(n == 1 || n == 3);

n = bp.chooseMove(d, SOUTH);

assert(n == 1 || n == 3);

n = sp.chooseMove(d, SOUTH);

assert(n == 1 || n == 3);

//Test the member functions, especially display() and constructor of Game class

BadPlayer bp5("Kobe");

BadPlayer bp6("Paul");

Board h(3,0);

b.setBeans(SOUTH, 1, 2);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 2);

Game g(h, &bp5, &bp6);

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

}

int main()

{

Tests();

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

}