Khang Luu

CS 32 Project 3

5/30/23

1. **a description of the design of your classes. We know what the public interfaces are, but what about your implementations: What are the major data structures that you use? What private member functions or helper non-member functions did you define for what purpose?**
   1. I use two vectors from the standard library that represent the North board and the South board. Because of the built-in functions, I don’t have to worry about memory leaks. For my North Player, the vector’s 0 index is the pot and holes are everything subsequent after the pot in a descending index order. For my South Player, however, the pot is still at index 0, but the holes are in descending order, meaning that if there are 3 holes, the last one would be at index 1, right next to the pot, and hole 1 would be at the last index, 3.
   2. In my Game Class, I have two private helper functions: GameOver and FinishingTheGame. GameOver informs my status if the Game is over, which is when both the sides have zero beans in play. I called on this function when one side cannot make a move, and so this really check that the game is over and beans are swept rather than a bug. My FinishingTheGame sweep the opponent beans into their pots and is called when a player attempted to make a move and fail, and this beans the game is over.
   3. In my SmartPlayer class, I have three helper functions. The first one is smartSow, which sows the copied boards and return true only when another turn is available (when the last bean landed in the pot). It also makes captures for the players. My smartMove essentially given a side, and tries to minimize the score that it received from recursive function of copied board. It will limits the recursion at depth greater than 2 and time limit above 5 seconds. Lastly, it calls on my last helper function evaluate, which given a side, it return how good it is for that player.

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

* The heuristics I used for my evaluation is the difference between the current player’s pot subtract the opponent’s pot: However, there are more scenarios I considered:
  + First checks cases where the winners are obvious, like when the number of beans in play + one pot is less than the beans in the opponent’s pot. For these I returned infinity if you are the winner and -infinity if you will lose. I also check the situation where one side has no more beans left to play, which means the game is basically over and I tally up the beans on each side.
  + If the above scenario wasn’t true, I then first checked the pot difference heuristics. If the both players have the same amount of beans in the pots, something that is frequent in the early rounds, I tailed up the number of beans on their sides and added that to the pots, using the difference between this new addition.
* Overall, this means that my evaluate is side dependent. When my recursive smartMove function reaches the leaf nodes, it would evaluate these nodes. Then, when it go back up the tree, my smartMove would choose the least desirable outcome, simulating the opponent playing at their optimum and always picking the worse move for me. Similarly, I will also trying to minimize my opponent’s odd. Finally, when the depth is zero, which the smartPlayer will now choose the outcome with the highest value. This means that though my opponent will always try to minimize my outcome, at depth 0, I can still choose the best outcome.

**3. pseudocode for non-trivial algorithms.**

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

check if I can sow there (ex. no pot or when there are no beans).

Side is North

Empty the chosen hole

While there are still beans left to sow

Visit each hole to the left in the North and drop a bean

While there are beans left to sow

Visit each hole in the South side in a counter-clockwise fashion and drop a bean, ignoring the pot

If there are no more beans, break.

Visit each hole in the North side and drop a beans, including the pot.

Side is South

Empty the chosen hole

While there are still beans left to sow

Visit each hole to the left in the South counter-clockwise and drop a bean

While there are beans left to sow

Visit each hole in the North side in a counter-clockwise fashion and drop a bean, ignoring the pot

If there are no more beans, break.

Visit each hole in the South side and drop a beans, including the pot.

}

bool Board::moveToPot(Side s, int hole, Side potOwner) {

if it’s a pot, return false

if its North and the potowner is North

move beans into correct pot.

If its North and the potowner is South  
 move beans into correct pot.

If its South and the potowner is North

move beans into correct pot.

If its South and the potowner is South

move beans into correct pot.

}

bool Game::move(Side s) {

if side is NORTH

if there are beans left on North’s side

North Player choose a move

If the move is hole -1

Then game is over

Return false

Board sow that move

While the last hole landed is a pot and on the same side

If there are no more beans left to play

Then the game is over

Return false

If the hole is not the pot

Break

North Player choose a move

Board sow that move

If the last bean landed in such a way that capturing the opponent’s beans is available

move the last bean in the North’s pot

move the opponent’s bean from the hole directly across into the North’s pot

return true;

else

the game is over

return false

if the side is SOUTH

if there are beans left on the South’s side

South Player choose a move

If the move is hole -1

Then game is over

Return false

Board sow that move

While the last hole landed is a pot and on the same side

If there are no more beans left to play

Then the game is over

Return false

If the hole is not the pot

Break

South Player choose a move

Board sow that move

If the last bean landed in such a way that capturing the opponent’s beans is available

move the last bean in the South’s pot

move the opponent’s bean from the hole directly across into the South’s pot

Return true;

Else

The game is over

Return false

}

void Game::play() {

if both players are un-interactive

ask user to press enter

while the game is not over

if South can’t make a move

then check if game is over

update status

brake

display

if North can’t make a move

then check if game is over

update status

break

display

if the players are un-interactive

ask user to press enter

display

if there is a winner

announce the winner

if there is not a winner

announce the game is tied

}

bool Game::GameOver() const {

if there are beans in play on both side

return false

return true

}

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

while the randomized number for hole is zero, is larger than the number of hole, and if there are no beans for that hole

generate a new random number

if the number is not zero, less than or equal to the number of hole, and the number of beans at the hole is not zero

break.

Return the random number

}

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

If the pot is empty on both sides, return 0

If there are no more beans left in play

And the pots on both sides equal each other, return 0

Compare the pot and return infinity to the player with a bigger pot, -infinity for the other

If one side has no more beans in play and the other has no more beans left

Tally add up the beans in the pot and the sides for each player

Return infinity for player with more beans, -infinity for the other

If a player has more beans in its pot than the beans in play on both sides added to the opponent’s pots

Return infinity for this player, -infinity for the other

If both players have more than 1 bean in play

If both players have equal beans in their pot

Tally up their pots with the number of beans on their side and return the difference

Else

Return the difference between their pot and the opponent pot

}

bool SmartPlayer::smartSow(Board& b, Side s, int hole) const {

check if sowing the copied board using the input parameter was successful

if the sowing was unsuccessful, return false

if an opportunity to capture from the opponent is available, then make the capture

return false

if the last bean landed in the player’s own pot

return true

return false

}

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

if there are no beans in play for the player

return -1

if the opponent has no more beans in play

for each hole on the player’s side

check for the hole with the smallest amount of beans

return that hole

bestValue = -inf;

for each hole on the player's side

if there are no beans at the hole

continue

make a copy of the board

sow the copied board

if the turn is not over

tempvalue = call smartMove on the copied board while increasing depth

else

temp value = call smartMove for the opponent on the copied board while increasing depth

if tempvalue is better than bestValue

bestValue = tempvalue

return bestValue

}

int SmartPlayer::smartMove(const Board& b, Side currSide, int depth, TimeKeeper& timer, double timeLimit) const {

if there is no more beans in play for either side, if the depth is greater than 3, or if the time limit is zero

evaluate the leaf nodes and return this value

bestValue = inf

newValue = 0

for every hole that the current player can sow

start the timer

make a copy of the board

if the player’s turn is not over

newValue = call smartMove on the copied board while increasing depth

else

newValue = call smartMove for the opponent on the copied board while increasing depth

if bestValue is greater than or equal to newValue

bestValue = newValue

return bestValue

}

**4. a note about any known bugs, serious inefficiencies, or notable problems you had.**

* I had an issue with my SmartPlayer function where it kept having an infinite loop problem and never go deeper than depth 1.
  + I realized my problem what that I passed into my index by reference into the recursive function, which changes the value and so my loop was busted.
  + I fixed this issue by redoing my smartMove function so that it doesn’t rely on passing by reference but rather it will return an int of the best value.
* Another problem I had is that when the SmartPlayer’s opponent’s has no more beans left to play, the SmartPlayer’s choosemove would try to create copies and sow the opponent’s holes. As all of the holes are empty, it had an infinite loop problem.
  + I overcame this challenge by adding an if statement in chooseMove that recognizes this situation and choose the hole with the smallest amount of beans so that the beans do not spill over into the opponent. Hence, the game will end.
* Another issue I encountered what that my play function would keep asking SmartPlayer to make a move on an empty board. Normally in my BadPlayer function and HumanPlayer function, I had a base case that return false when a move can’t be make, but because the SmartPlayer is still making hypothetical board, it ran into an infinite loop trying to sow an invalid board.
  + I overcame this challenge by adding an if statement that return -1 if the player has no more beans left to sow. Then, in my play function, I checked if the move chosen was -1, in that case the game is declared over.
* Lastly, overall my SmartPlayer still dominates the BadPlayer every time, I cannot understand how if it is the North Player (going second), that it only beats the BadPlayer about 55% of the time, whereas if SmartPlayer go first, it always beat the BadPlayer and even another SmartPlayer. More importantly, I realized through numerous tests and observing SmartPlayers playing each other that my algorithm’s strength is in “attacking,” such as making strategic moves to make big captures. However, it is weaker in defending its holes from the opponents.

**5. 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."**

**Test functions in Board.cpp**

Board b(6, 3);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 6, 7);

b.setBeans(NORTH, 3, 11);

b.setBeans(SOUTH, 1, 1);

b.setBeans(SOUTH, 2, 15);

b.setBeans(SOUTH, 4, 5);

b.setBeans(SOUTH, 5, 1);

b.sow(SOUTH, 2 , endside, endHole);

assert(b.beansInPlay(SOUTH) == 21);

assert(b.beansInPlay(NORTH) == 34);

assert(b.totalBeans() == 56);

assert(endside == SOUTH, endHole == 4);

cout << "Tests Passed!" << endl;

Board b(6, 3);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 6, 7);

b.setBeans(NORTH, 3, 11);

b.setBeans(SOUTH, 1, 1);

b.setBeans(SOUTH, 2, 15);

b.setBeans(SOUTH, 4, 5);

b.setBeans(SOUTH, 5, 1);

b.sow(NORTH, 3, endside, endHole);

assert(endside == NORTH, endHole == 5);

cout << "Tests Passed!" << endl;

Board b(3, 2);

Side endside = SOUTH;

int endhole = -1;

b.setBeans(NORTH, 0, 4);

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 2, 0);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 0, 6);

b.setBeans(SOUTH, 1, 4);

b.setBeans(SOUTH, 2, 2);

b.setBeans(SOUTH, 3, 0);

b.sow(NORTH, 1, endside, endhole);

assert(endside == NORTH && endhole == 0);

assert(b.sow(NORTH, endhole, endside, endhole) == false); // pots can’t be sowed

cout << "Test passed" << endl;

Board b(6, 3);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 6, 7);

b.setBeans(NORTH, 3, 11);

b.setBeans(SOUTH, 1, 1);

b.setBeans(SOUTH, 2, 15);

b.setBeans(SOUTH, 4, 5);

b.setBeans(SOUTH, 5, 1);

assert(b.moveToPot(SOUTH, 2, NORTH) == true);

assert(b.moveToPot(SOUTH, 0, NORTH) == false);

assert(b.moveToPot(NORTH, 3, SOUTH) == true);

assert(b.beans(NORTH, 0) == 15);

assert(b.beans(SOUTH, 0) == 11);

cout << "Tests Passed!" << endl;

**Test Evaluate**

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(6, 0);

b.setBeans(NORTH, 0, 22);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 1);

b.setBeans(NORTH, 3, 1);

b.setBeans(SOUTH, 0, 20);

b.setBeans(SOUTH, 5, 3);

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

g.display();

cout << g.evaluate(SOUTH) << endl; // p121: -2

Board b(6, 0);

b.setBeans(NORTH, 0, 22);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 3, 1);

b.setBeans(NORTH, 3, 1);

b.setBeans(SOUTH, 0, 22);

b.setBeans(SOUTH, 5, 1);

g.display();

cout << g.evaluate(SOUTH) << endl; // p123: 0

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(6, 0);

b.setBeans(NORTH, 0, 22);

b.setBeans(SOUTH, 0, 20);

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 5, 2);

b.setBeans(SOUTH, 5, 3);

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

g.display();

cout << g.evaluate(SOUTH) << endl; // p111: -2

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(6, 0);

b.setBeans(NORTH, 0, 22);

b.setBeans(SOUTH, 0, 24);

b.setBeans(NORTH, 1, 1);

b.setBeans(SOUTH, 6,1);

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

g.display();

cout << g.evaluate(SOUTH) << endl; // p112: 2

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(6, 0);

b.setBeans(NORTH, 0, 22);

b.setBeans(SOUTH, 0, 25);

b.setBeans(NORTH, 1, 1);

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

g.display();

cout << g.evaluate(SOUTH) << endl; // p113: inf

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(6, 0);

b.setBeans(NORTH, 0, 22);

b.setBeans(SOUTH, 0, 20);

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 5, 2);

b.setBeans(SOUTH, 6,1);

b.setBeans(SOUTH, 3, 2);

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

g.display();

cout << g.evaluate(SOUTH) << endl; // P21: 2

BadPlayer bp1("Bart");

BadPlayer bp2("Homer");

Board b(6, 0);

b.setBeans(NORTH, 0, 25);

b.setBeans(SOUTH, 0, 20);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 4, 1);

b.setBeans(SOUTH, 6,1);

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

g.display();

cout << g.evaluate(SOUTH) << endl; // P22: -inf

**Test SmartPlayer**

Board b(6, 3);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 6, 7);

b.setBeans(NORTH, 2, 2);

b.setBeans(NORTH, 3, 1);

b.setBeans(SOUTH, 4, 3);

b.setBeans(SOUTH, 2, 15);

b.setBeans(SOUTH, 6, 1);

b.setBeans(SOUTH, 5, 2);

SmartPlayer sp("Smart");

BadPlayer bp("Bad");

b.dump();

assert(sp.chooseMove(b, NORTH) == 2); // choosing this will lead to landing 1 bean in the pot and a capture of 15.

assert(sp.chooseMove(b, SOUTH) == 6); // choosing this allowed a bean to land in a pot and grant player another turn

cout << "Tests Passed!" << endl;

Board b(6, 0);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 0, 22);

b.setBeans(SOUTH, 0, 20);

b.setBeans(NORTH, 2, 1);

b.setBeans(NORTH, 5, 2);

b.setBeans(SOUTH, 3, 2);

b.setBeans(SOUTH, 5, 1);

SmartPlayer sp("Smart");

assert(sp.chooseMove(b, SOUTH) == 3); // move to avoid a capture

assert(sp.chooseMove(b, NORTH) == 5); // allowing a capture

Board b(6, 0);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 0, 8);

b.setBeans(SOUTH, 0, 7);

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 3, 5);

b.setBeans(NORTH, 6, 3);

b.setBeans(SOUTH, 5, 1);

b.setBeans(SOUTH, 2, 3);

b.setBeans(SOUTH, 3, 1);

b.dump();

SmartPlayer sp("Smart");

assert(sp.chooseMove(b, SOUTH) == 5); // to capture the opponent’s 3 beans

assert(sp.chooseMove(b, NORTH) == 6); // to avoid being captured

Board b(6, 0);

Side endside;

int endHole = 0;

b.setBeans(NORTH, 0, 8);

b.setBeans(NORTH, 1, 1);

b.setBeans(NORTH, 3, 5);

b.setBeans(NORTH, 6, 3);

b.dump();

SmartPlayer sp("Smart");

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

// assert than when there is no beans left for the player, it would update play that the game is over by choosing -1

assert(sp.chooseMove(b, NORTH) == 1);

// assert that when the opponent has zero bean, the player tries to maximizes winning chance by picking the hole with the smallest amount of beans to prevent a spillover