# Timothy Liu CS32 Proj 3

# Board

## 

## Board Constructor

For the board, I have a 2D vector named mBoard, which contains 2 vectors and within those 2 vectors are ints that represent each of the holes on the board. The mBoard vector is initialized with 2 vectors, and the 2 vectors are initialized to have nHoles+1 elements and each int is initialized to have nInitialBeansPerHole. The private members mHoles and mInitialBeansPerHole are also initialized to whatever nHoles and nInitialBeansPerHole is in the member initialization list.

## 

## beansInPlay(Side s)

There's an int that keeps track of how many beans have been counted up. The function then goes through each of the holes on the given side (except the pot) with a for loop and adds the # beans in that hole to the running total. It will then return the # of beans counted after going through the holes.

## totalBeans()

There's an int that keeps track of how many beans have been counted up. The function then goes through each of the holes on one side of the board with a for loop and adds the # beans in each hole to the running total, and repeats for the other side. The # of beans counted is returned.

## sow(Side s, int hole, Side& endSide, int& endHole)

There is a bounds check for whether a given position is valid or whether there are enough beans to sow. The # of beans at a given hole is then assigned to a variable to represent the # of beans able to be sown and the beans at the given hole is set to 0.

Pseudocode:

| if (hole is POT or hole > mHoles or no beans to sow)  return false  # beans to sow = # beans at given position  Erase the beans at given position  while still have beans to sow:  if you just switched sides  if switching from NORTH:  Set position to leftmost hole on SOUTH side  if switching from SOUTH:  Set position to rightmost hole on NORTH side  if you're on SOUTH side and you didn't just switch sides:  Increment hole # by 1  If you're on NORTH side and you didn't just switch sides:  Decrement hole # by 1  If you reach SOUTH pot:  if you have no beans to sow and you're NORTH:  End your turn at NORTH's rightmost hole  Otherwise, player must be SOUTH side:  Decrement # beans to sow  Increment South pot  Indicate that you're about to switch sides  If you reach NORTH pot:  If you have no beans to sow and you're SOUTH  End your turn at SOUTH's leftmost hole  Otherwise, player must be NORTH side:  Decrement # beans to sow  Increment NORTH pot  Indicate that you're about to switch sides  If you didn't reach a pot:  Add beans to the hole you landed on  Decrement # beans able to sow  Indicate that you didn't switch sides  Set endSide and endHole to the position you end at  Return true |
| --- |

## moveToPot

There's a bounds check for whether a given hole is valid. It then adds the beans from that hole to the pot and sets the beans at the given hole to 0.

## Copy Constructor

| Initialize # holes and initial beans per hole to be the same as other board's  Initialize mBoard with vector of the same dimensions and values  Go through each NORTH hole of the other board:  Set # beans at given hole to be the same as other board at the same position  Go through each SOUTH hole of the other board:  Set # beans at given hole to be the same as other board at the same position |
| --- |

## Assignment Operator

| Check whether the two Board objects are the same  Copy the # holes and initial beans per hole to be the same as other board's  Set mBoard to be 2D vector of the same dimensions and values as other board's  Go through each NORTH hole of the other board:  Set # beans at given hole to be the same as other board at the same position  Go through each SOUTH hole of the other board:  Set # beans at given hole to be the same as other board at the same position  Return this board |
| --- |

# Player

## HumanPlayer::chooseMove(const Board&b, Side s)

| Prompt player to choose a hole:  If chosen hole is out of bounds:  Warn them and reprompt them  If chosen hole has no beans:  Warn them and reprompt them  If hole is valid:  Stop prompting them and choose that hole  Return the hole they chose |
| --- |

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

* Will always choose the farthest hole from the pot with beans

| if side is NORTH:  Go through each hole starting from rightmost hole  If the hole has beans, pick that hole  if side is SOUTH:  Go through each hole starting from leftmost hole  If the hole has beans, pick that hole |
| --- |

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

| If unable to make any valid move:  Return -1  Helper Function call to find best valid move given a depth  If any move leads to a guaranteed win/loss/tie  Choose the first valid hole |
| --- |

## 

## void chooseMoveHelper(const Board& b, Side s, int&bestHole, int& value, int depth, Timer& clock) const;

| Figure out who's who  current value = value of current pos(using evaluate function)  If time is about to be over 5s:  End search preemptively here  if no beans left in play for either side or reached max depth limit  No best hole found here  value = current value  return  if current value is a guaranteed win, lose, or tie  Best hole is any hole(doesn't matter)  value = current value  return  Set arbitrary values for the best value found for now  Go through each possible hole to choose from  if hole is empty, skip the hole  Make a copy of the board  Use the board copy to "make" moves and go from there  Sow beans at the given hole on the board copy  Possible next best hole is set to current hole  Possible next value set to 0 (arbitrary)  If you end up in your own pot after moving  recursively call this function to simulate going again (but depth doesn't increase b/c it doesn't count as a turn)  If you end up with a capture  Move all the beans to player's pot  Evaluate the position from there  Otherwise, your turn will end there and your opponent goes:  Search decreases by 1 level  Compare whether the outcome after your opponent goes led to a better outcome for you  if next value is better:  best Hole = current hole chosen  best Value found (so far) = value you found from after your opponent went  value = best value found  end function |
| --- |

## 

## evaluatePos(const Board& board, Side evaluatingSide) const

| If game is over:  If you win:  Return 9999  If opponent wins:  Return -9999  Game not over:  If you win no matter what(more than half the total beans in pot)  Return 9999  If you lose no matter what(opponent has more than half the total beans in pot)  Return -9999  Otherwise, return difference in pots |
| --- |

# Game

## Display()

| Print North Player's name (with spacing before)  Print North Player's holes (with spaces in between)  Print North's pot, then print South's pot farther away on same line  Print South Player's holes (with spaces in between)  Print Player's name (with spacing before) |
| --- |

## status(bool& over, bool& hasWinner, Side& winner)

| if no beans can be played for either side, indicate that game is over  If North's pot has more beans:  Assert that there's a winner and it's North  If South's pot has more beans:  Assert that there's a winner and it's South  Otherwise it's a tie, so assert there's no winner  Otherwise, game is not over so don't update winning side |
| --- |

## move(Side s)

| If no possible moves able to be chosen by player(no beans left on their side):  Print message that indicates nothing left and that remaining beans are swept into opponent's pot  Go through each hole on board:  Count # beans in North's holes with running total  Count # beans in South's holes with separate running total  Add up beans in each of North's holes to North's running total  Erase the beans in that hole on North's side  Add up beans in each of South's holes to South's running total  Erase the beans in that hole on South's side  Add the running total to North's pot  Add the running total to South's pot  Update status to indicate which side wins  Return false b/c no move possible  Otherwise: must be a valid move  Current player attempts to choose a hole to move  Sow from the chosen hole by the player  If sow results in capture:  Add up beans in the same hole on both sides  Add the total beans to current player's pot  Remove beans from the same hole on both sides  End turn here and swap side and current player  If sow results in extra turn (ends in current player's pot)  Give player another turn by going through entire function again (recursive call) without changing the current player or current side  Otherwise:  Swap to opposing player and end turn for current player |
| --- |

## play()

| If Robot vs Robot:  As long as game's not over:  If South's turn:  South Player make move  Check if South's move ended game  If game doesn't end:  Prompt human to continue simulating  Swap turns  If North's turn:  North Player make move  Check if North's move ended game  If game doesn't end:  Prompt human to continue simulating  Swap turns  If Human vs Robot OR Human vs Human:  As long as game's not over:  If South's turn:  South Player make move  Check if South's move ended game  Swap turns  If North's turn:  North Player make move  Check if North's move ended game  Swap turns  If game is over:  If one side wins:  Sow remaining beans to each pot  Display board and winner  If neither side wins:  Sow remaining beans to each pot  Display board and state that there's a tie |
| --- |

# Test Cases

## Board Tests:

// Original board

Board original(3,5);

for(int hole = 0; hole <= original.holes(); hole++)

{

original.setBeans(NORTH, hole, hole+2\*hole);

original.setBeans(SOUTH, hole, original.holes()-hole+2);

}

// Copy constructor

Board copy(original);

assert(copy.totalBeans()==original.totalBeans());

for(int hole = POT; hole <= copy.holes(); hole++)

{

assert(copy.beans(NORTH, hole)==original.beans(NORTH, hole));

assert(copy.beans(SOUTH, hole)==original.beans(SOUTH, hole));

}

// Assignment operator

Board assigned(5,1);

assigned = copy;

assert(assigned.totalBeans()==original.totalBeans()

&& assigned.totalBeans() == copy.totalBeans());

for(int hole = POT; hole <= copy.holes(); hole++)

{

assert(assigned.beans(NORTH, hole)==original.beans(NORTH, hole) && assigned.beans(NORTH, hole)==copy.beans(NORTH, hole));

assert(assigned.beans(SOUTH, hole)==original.beans(SOUTH, hole) && assigned.beans(SOUTH, hole)==copy.beans(SOUTH, hole));

}

Side endS;

int eHole;

// South player sows and doesn't go into North's pot

Board sow1(3,14);

sow1.sow(SOUTH, 1, endS, eHole);

assert(sow1.totalBeans()==14\*3\*2);

assert(sow1.beans(NORTH, POT)==0);

assert(sow1.beans(SOUTH, POT)==2);

assert(sow1.beans(SOUTH, 1)==2);

assert(endS == SOUTH && eHole == 1);

// South player sows and ends up in own pot

Board sow2(3,2);

sow2.sow(SOUTH, 2, endS, eHole);

assert(endS == SOUTH && eHole == POT);

assert(sow2.beans(SOUTH, POT)==1);

assert(sow2.totalBeans()==12);

// Sowing, but doesn't end up in opponent's pot after

Board sow3(3,5);

sow3.display();

sow3.sow(NORTH, 1, endS, eHole);

assert(sow3.beans(NORTH, POT)==1);

assert(sow3.beans(SOUTH, POT)==0);

assert(endS==NORTH && eHole==3);

sow3.display();

sow3.sow(SOUTH, 2, endS, eHole);

assert(endS==SOUTH && eHole==1);

assert(sow3.beans(NORTH, POT)==1);

assert(sow3.beans(SOUTH, POT)==1);

// Testing other board functions

Board basics(3,4);

// Out of bounds hole

assert(basics.moveToPot(NORTH, 4, NORTH)==false);

// Valid hole

assert(basics.moveToPot(NORTH, 1, NORTH)==true);

assert(basics.beans(NORTH, 1)==0);

assert(basics.beans(NORTH, POT)==4);

assert(basics.moveToPot(NORTH, 2, NORTH)==true);

assert(basics.beans(NORTH, 2)==0);

assert(basics.beans(NORTH, POT)==8);

assert(basics.totalBeans()==3\*4\*2);

assert(basics.beansInPlay(SOUTH)==3\*4);

assert(basics.beansInPlay(NORTH)==3\*4-8);

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

/// Player Tests

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

BadPlayer bPlayer("BadBot");

SmartPlayer sPlayer("Einstein");

HumanPlayer hPlayer("You");

// Bad Player vs Smart Player

// Smart player should win vs Bad Player

Board BadvsSmartBoard(4, 3);

Game BadvsSmartGame(BadvsSmartBoard, &bPlayer, &sPlayer);

bool isOver;

bool hasWinner;

Side winner;

BadvsSmartGame.play();

BadvsSmartGame.status(isOver, hasWinner, winner);

assert(isOver == true && hasWinner == true && winner == NORTH);

// Only performs capture if on your side

// 0 0 0 0 0

// 0 0

// 0 0 5 5 0

Board captureTest(5,0);

captureTest.setBeans(SOUTH, 3, 5);

captureTest.setBeans(SOUTH, 4, 5);

Game captureTestGame(captureTest, &bPlayer, &hPlayer);

captureTestGame.move(SOUTH);

assert(captureTestGame.beans(SOUTH, POT)==1);

assert(captureTestGame.beans(NORTH, 4)==1);

// 0 0 1 1 1

// 0 1

// 0 0 0 6 1

// Successful capture

// 0 0 2

// 0 0

// 2 0 0

Board captureTest2(3,0);

captureTest2.setBeans(SOUTH, 1, 2);

captureTest2.setBeans(NORTH, 3, 2);

Game captureTestGame2(captureTest2, &bPlayer, &sPlayer);

captureTestGame2.move(SOUTH);

assert(captureTest2.totalBeans()==4);

assert(captureTestGame2.beans(SOUTH, POT) == 3);

// 0 0 0

// 0 3

// 0 1 0

// Multiple successive turns (land in pot)

// 0 1 3

// 0 0

// 0 0 2

Board multipleTurns(3,0);

multipleTurns.setBeans(NORTH, 2, 1);

multipleTurns.setBeans(NORTH, 3, 3);

multipleTurns.setBeans(SOUTH, 3, 2);

assert(multipleTurns.totalBeans()==6);

Game mTurnsGame(multipleTurns, &sPlayer, &bPlayer);

mTurnsGame.move(NORTH);

assert(mTurnsGame.beans(NORTH, POT) == 3);

assert(mTurnsGame.beans(SOUTH, 1) == 1);

assert(mTurnsGame.beans(NORTH, 1) == 0);

assert(multipleTurns.totalBeans()==6);

// 0 0 0

// 3 0

// 1 0 2

// Smart Player recognizes capture as best outcome

Board captureTest3(3, 0);

captureTest3.setBeans(NORTH, 3, 1);

captureTest3.setBeans(SOUTH, 1, 1);

captureTest3.setBeans(SOUTH, 2, 1);

// 0 0 1

// 0 0

// 1 1 0

assert(sPlayer.chooseMove(captureTest3, SOUTH)==2);

// Multiple successive turns (choose hole 3)

// 0 0 1

// 0 0

// 3 2 1

Board multiTurns(3,0);

multiTurns.setBeans(SOUTH, 1, 3);

multiTurns.setBeans(SOUTH, 2, 2);

multiTurns.setBeans(SOUTH, 3, 1);

multiTurns.setBeans(NORTH, 3, 1);

multiTurns.setBeans(NORTH, 1, 1);

assert(sPlayer.chooseMove(multiTurns, SOUTH)==3);

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

/// Game Tests

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

// No beans on one side (game ends)

Board b1(3,0);

b1.setBeans(NORTH, 2, 3);

Game g1(b1, &sPlayer, &bPlayer);

g1.play();

assert(g1.move(SOUTH)==false);

// bool isOver;

// bool hasWinner;

// Side winner;

// Tie game

Board b2(3,0);

b2.setBeans(NORTH, 1, 1);

b2.setBeans(SOUTH, 3, 1);

Game g2(b2, &bPlayer, &sPlayer);

g2.play();

assert(g2.beans(NORTH, POT)==1);

assert(g2.beans(SOUTH, POT)==1);

g2.status(isOver, hasWinner, winner);

assert(isOver);

assert(!hasWinner);

// One side wins

Board b3(3,0);

b3.setBeans(NORTH, POT, 3);

Game g3(b3, &bPlayer, &sPlayer);

g3.play();

g3.status(isOver, hasWinner, winner);

assert(isOver);

assert(hasWinner);

assert(winner == NORTH);

# Difficulties

## SmartPlayer

I had a problem where my SmartPlayer wouldn't recognize choosing a hole that would result in multiple successive turns, meaning that the Smart Player wouldn't choose the hole that resulted in landing in the pot and being able to get more turns that let them land in the pot easily. My problem was that I didn't keep the depth the same, so everytime it got successive turns, it wouldn't be able to determine the position after the opponent made a move.

I also had an unoptimized check for each hole initially. Whenever the hole was empty, it would still check for all the possible conditions for moves like capture or landing in pot even as if it were a valid move, even though it wasn't necessary. It hindered my move time and after I fixed it, my Smart Player became more efficient. Specifically, my fix was that whenever the hole was empty, it would just skip the entire loop and just check for the next valid hole with beans in it.

## 

## Board

I had a problem where if one side sowed and ended in their own pot, they wouldn't switch sides and would keep continuing on to a hole that wasn't valid. To do this, I made a check for when a player reached their own pot and still had beans to sow so that they could switch sides and continue sowing from the other side.

## Game

One major problem was that after the game was over, it would display the board and say the winner, but it would only show the board right before all the beans got moved to the pots. I realized that after the game was determined to be over by whether or not any beans were in play, I didn't actually move all the beans left on the board to their pots and just displayed it right there. To fix this, whenever the game was detected to be over, I made sure to move all the beans to the pot, then I would display the board.

## Another problem I had was not being able to give a player another turn after landing in their own pot. To fix this, I had to make sure that after the player makes their first move, it doesn't automatically continue on with the rest of the body of the code in my chooseMove function where the player and sides are swapped. Instead, after checking for the player landing in their own pot, I recursively called the move function again to give the current player another turn.

## 