**Description of the Design of the Classes**

* Board: The member variables of this class include mHoles (int), m\_SouthSide (vector <int>), and m\_NorthSide (vector <int>). mHoles are used to store the number of holes on each side when initializing the Board type. The m\_SouthSide and m\_NorthSide are used to store the hole, along with the amount of beans in each hole. The 0 index of the vector would each correspond to the Player’s pot, and the rest oif the vector would correspond to the hole number of the player. A helper function that I defined was void increment(Side s, int hole). I would use this helper function when implementing sow in order to increment the number of beans in a hole. I did this in order to make the implementation much more readable and organized.
* Player.h: The helper functions I included in the HumanPlayer and BadPlayer class is valid\_input(const Board& b, const Side s, const int input) const. For the HumanPlayer class, I would use the valid\_input function to check whether the input that the player inputted was valid. Otherwise, I would reask the player to redo their input. For the BadPlayer class, I would use the valid\_input function to while loop through the vector and choose the lowest hole that is valid. The description of the SmartPlayer is included below.
* Game.h: The private member variables of the game class include m\_b, m\_south, and m\_north. The first private member variable, m\_b, is used to store the board that will be used for the game. The second and third private variables, m\_south and m\_north, are player pointers so we can reference derived classes such as SmartPlayer and BadPlayer from the parent class Player.

**Description of SmartPlayer::chooseMove**

Implementation of chooseMove\_Helper: The chooseMove function solely relies on a helper function known as chooseMove\_Helper that takes in the following parameters: (1) const Board& b (2) Side s\_orig, (3) Side opp\_orig, (4) Side s, (5) Side s, (6) int& depth, (7) int& bestHole, (8) int& value. The function first checks if the beansInPlay is less than 0 in order to check if the game is over.

Second, the function checks wheter to check beyond a certain depth of the recursion case (the base case). Next, the function loops through the board’s holes to store the valid\_hole positions in a vector known as valid\_holes. Fourth, the function loops through every hole in teh valid\_hole vector and performs the move by sowing the board and performing the capture if permitted. After, the function can perform two recursion calls.

The first recursion call checks if the end side is the same side as the current side and the end hole is in the current player’s pot. If so, we perform the recursion call with the temporary board, without incrementing the depth to encounter the possibilities in going again. Otherwise, the function switches the sides and another a recursion call is created on the temporary board but with the sides switched, as well as incrementing the depth by 1.

Finally, in a postorder fashion, the function checks sets the bestHole to the function’s bestHole parameter if the value of the temporary board is greater than the current value of the original player, or, if the player is the opponent player, the bestHole parameter would be set the value of the valid\_holes position if the value of the temporary board is less than the current value.

Implementation of eval (heuristic function)l: The maximizer of the function would be the current player, adn the minimizer of the function would be the opponent player. If the game is over, the evaluation function checks for three cases. The first case is if the beans in the current player’s side is greater than the beans in the opponent side’s pot. In this case, the current side wins and the function returns 10000. The second case is if the beans int he opponent player’s side is greater thna the beans in the current side’s pot. In this case, the opponent isde wins and the function returns 10000. Otherwise, the third case would be a tie and the function returns 0. If the game is not over, the function would return the difference between the beans from the current and opponent’s pot.

**Pseudocode**

Board.cpp

Board::Board(int nHoles, int nInitialBeansPerHole)

If nHoles is not positive, act as if it were 1

Otherwise, set nHoles = mHoles

int mBeansPerHole

If nInitialBeansPerHole is negative, act as if it were 0.

Otherwise, set mBeansPerHole to nInitialBeansPerHole

Store North’s pot in vector

Store South’s pot in vector

for the number of holes

push back mBeansPerHole on to the South vector

push back mBeansPerHole on to the North vector

int Board::beans(Side s, int hole) const

if hole is invalid

return -1

if side is north

return beans in northside hole

otherwise

return beans in southside hole

int Board::beansInPlay(Side s) const

int total = 0

if side is SOUTH

for every hole in south side

sum the beans in the hole to total

if side is NORTH

for every hole in north side

sum the beans in the hole to total  
 return total

int Board::totalBeans() const

int total = 0

for every hole in south side

sum the beans in the hole to total

for every hole in north side

sum the beans in the hole to total  
 return total

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

Side curr\_s = s;

int curr\_hole = hole;

If the hole indicated by (s, hole) is empty or invalid or a pot, this function returns false without changing anything

int rotations = number of beans int eh current hole

set chosen south pot to 0 if player is south

set chosen north pot to 0 if player is north

while (rotations > 0)

if current hole is 0 and current side and current player is NORTH

set current hole to 0

set current side to South

if current hole is 0 and current side and current player is SOUTH

set current hole to 0

set current side to South

if current side is south

increment current hole by 1

if current side is north

decrement current hole by 1

If current hole is one more than mHoles and the current side is SOUTH

switch side to north

skip South’s pot if north player

If current hole is zero and the current side is NORTH

switch side to south

skip North’s pot if south player

Increment Current hole's beans

endSide = curr side

endHole = current hole

return true

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

If input is valid,

return false

Move to bean\_in\_hole variable and set hole to 0

Set number of beans to corresponding hole

return true

bool Board::setBeans(Side s, int hole, int beans)

If inputs are invalid

return false

If side is SOUTH, set the value of that hole to the number of beans passed

into the function.

Otherwise, if side is NORTH, set the value of that hole to the

number of beans passed into the function.

Player.cpp

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

int input = -2;

while input is not valid

Select a hole

Record the input

If the input is greater than the holes or the input is less than or equal to 0,

tell the player that the hole number muset be from 1 to b.holes()

else if beans in the hole is empty

There are no beans in that hole

return input

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

if beansInPlay is lesse than or equal to 0

return -1

i = 1

while the holes is not valid

increment i by 1

Tell the viewer that bad player chooses this hole

return i

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

**if beansInPlay is less than or equal to 0**

**return -1**

**int val = 0**

**int besthole = 03**

**int depth = 0**

**call chooseMoveHelper**

**Display to viewer that smartplayer chose this hole**

**return besthole**

**void SmartPlayer::chooseMove\_Helper(const Board& b, Side s\_orig, Side opp\_orig, Side s, int& depth, int& bestHole, int& value, double timeLimit, JumpyTimer& timer) const**

**int curr\_value = current value of the board**

**If no move for player exists (i.e., game is over),**

**Create temporary board**

**bestHole = -1**

**Move the beans to determine winner**

**value = curr\_value**

**return;**

**If the criterion says we should not search below this node or the time limit is below 0**

**bestHole = -1**

**value = eval(b, s\_orig)**

**return;**

**Store valid-holes to check in valid\_holes vector**

**for every hole in the valid\_hole vector,**

**double thisBranchTimeLimit = timeLimit / (valid\_holes.size()-i);**

**double startTime = timer.elapsed();**

**Board t\_board = Board(b);**

**Side end\_s;**

**int end\_hole = -1;**

**int t\_value = 0;**

**int t\_bestHole = -3;**

**Sow the board**

**//Perform the capture**

**if side is the endside and endHole is greater then 0**

**if side is south and beans in north side does not equal 0 and beans in**

**endhole is 1 and endHole is not 0**

**move the beans from the endH South pot and pot across to**

**current pot**

**else if side is north and beans in south side does not equal 0 and beans in endHoel is 1 and endHOle is not 0**

**move the beans from the endH South pot and pot across to**

**current pot**

**If landed in own pot, go again**

**int t\_depth = depth;**

**chooseMove\_Helper(t\_board, s\_orig, opp\_orig, s, t\_depth, t\_bestHole,**

**t\_value);**

**Otherwise, switch sides**

**int t\_depth = depth + 1;**

**chooseMove\_Helper(t\_board, s\_orig, opp\_orig, t\_opponent, t\_depth,**

**t\_bestHole, t\_value);**

**If bestHole == Initialization case**

**bestHole = valid\_holes[i]**

**value = t\_value**

**If Player is the maximizer and t\_value > value**

**bestHole = valid\_holes[i]**

**value = t\_value**

**If Player is the minimizer and t\_value < value**

**bestHole = valid\_holes[i]**

**value = t\_value**

**timeLimit -= (timer.elapsed() - startTime);**

**if (timeLimit <= 0)**

**timeLimit = 0;**

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

If game is over

If current side wins

return 10000

If opposing side wins

return -10000

else it’s a tie

return 0

If Game is not over, return the difference between beans from the current and opposing

pot

Game.cpp

bool Game::move(Side s)

If beansInPlay is greater than 0 (game is still in play)

const Side Player\_s = s;

Side curr\_s = s;

int move;

Side endS;

int endH;

While the endHole isn't the player's pot

Store move

//Performing the capture

if side is the endside and endHole is greater then 0

if side is south and beans in north side does not equal 0 and beans in

endhole is 1 and endHole is not 0

move the beans from the endH South pot and pot across to

current pot

else if side is north and beans in south side does not equal 0 and beans in endHoel is 1 and endHOle is not 0

move the beans from the endH South pot and pot across to

current pot

If endH is greater than 0,

break.

Otherwise, Get another turn if endH is 0

If game is over

if s == SOUTH

Move the remaining beans to North's pot

Let the player know that the beans have been sowed

else

Move the remaining beans to North's pot

Let the player know that the beans have been sowed

return false

return true

else (game is not in play)

if s == SOUTH

Move the remaining beans to north’s pot

else

Move the remaining beans to south’s pot

return false

void Game::status(bool& over, bool& hasWinner, Side& winner) const

If game is over

if beans in south’s side is equal to beans in north’s side

hasWinner = false

else if beans in south’s side is greater than beans in north’s side

hasWinner = true

winner = south

else

hasWinner = true

winner = north

over = true

Otherwise, game is not over

over = false

void Game::play()

If both player's are not interactive

While the game is not over

cout << “Press ENTER to continue: “;

if ENTER is pressed,

display board

set status

Perform move

Rotate turns

else

cout << “Game canceled” << endl

else

While game is not over

display()

set status

move

Rotate turns

If game is over and has a winner

if winner is SOUTH

output that the winner is south

else

output tha the winner is north

Otherwise, the game is over and is a tie

display board

**Notable Problems**

A serious ineffcacy of the smartmove player is that the time it takes for the function to find the best choice after a depth of 4 exponentially increases. Although, on average, a depth of no more than produces the choice in no more than 4 second on average. If you go past a depth of over 4, the choice can take up to 45 second to produce, which is inefficient.

A notable obstacle that I overcame was figuring how the function would pass the bestHole and value to the next recursion call back. Although I had correctly implemented the evaluate function correctly and placed the postorder evaluation correctly as mirrored in pseudocode, the value never was passed back. After spending hours trying to fix the problem, and rearranging the evaluation statement of the recursive call, I found that the problem was actually that my parameters for the value and bestHole were not pass by reference. This explained why the value did not change and is a lesson for me to always check if the parameters that we are changing tduring the recursive call are pass by reference parameters and not pass by value.

**Test Cases**

Test Case 1

Board c2(6, 20);

c2.sow(NORTH, 1, temp\_s, temp\_h);

assert(c2.beans(NORTH, 1) == 1);

assert(c2.beans(NORTH, 0) == 2);

assert(c2.beans(SOUTH, 1) == 22);

assert(c2.beans(SOUTH, 2) == 22);

assert(c2.beans(SOUTH, 3) == 22);

assert(c2.beans(SOUTH, 4) == 22);

assert(c2.beans(SOUTH, 5) == 22);

assert(c2.beans(SOUTH, 6) == 22);

assert(c2.beans(SOUTH, 0) == 0);

assert(c2.beans(NORTH, 6) == 21);

assert(c2.beans(NORTH, 5) == 21);

assert(c2.beans(NORTH, 4) == 21);

assert(c2.beans(NORTH, 3) == 21);

assert(c2.beans(NORTH, 2) == 21);

assert(temp\_s == SOUTH);

assert(temp\_h == 6);

* Purpose: The purpose of this test case was to determine if sow worked correctly and to see if the board was constructed correctl

Test Case 2

Board c6(6, 20);

BadPlayer h1("Marc");

HumanPlayer h("Marc");

h.chooseMove(c5, SOUTH);

assert(h1.chooseMove(c6, SOUTH) == 1);

c6.setBeans(SOUTH, 1, 0);

h1.chooseMove(c6, SOUTH);

assert(h1.chooseMove(c6, SOUTH) == 2);

c6.setBeans(SOUTH, 2, 0);

assert(h1.chooseMove(c6, SOUTH) == 3);

c6.setBeans(SOUTH, 3, 0);

assert(h1.chooseMove(c6, SOUTH) == 4);

c6.setBeans(SOUTH, 4, 0);

assert(h1.chooseMove(c6, SOUTH) == 5);

c6.setBeans(SOUTH, 5, 0);

assert(h1.chooseMove(c6, SOUTH) == 6);

c6.setBeans(SOUTH, 6, 0);

assert(h1.chooseMove(c6, SOUTH) == -1);

* Purpose: The purpose of this test case to determine if BadPlayer was able to detect invalid holes and not choose that invalid hole as a choice. Int his case, I was able to set the beans to 0 and check if BadPlayer was able to detect if it was an invalid hole.

Test Case 3

Board b(6, 3);

SmartPlayer h1("Margarette");

SmartPlayer h2("Matthew");

Game g(b, &h1, &h2);

g.play();

* Purpose: Although this test case did not have any asset’s contained in the code, this particular case helped me determine what would happen if two SmartPlayer’s went against each other. In this case, it would be a tie. This also helped me determine whether the tie case actually got called in the play function

Test Case 4

Board b(6, 3);

b.setBeans(NORTH, 4, 4);

SmartPlayer h1("Marc");

assert(h1.chooseMove(b, NORTH)==2);

* Purpose: The purpose of this test case was to check if the chooseMove function correctly predicted the right move on the basis that the depth was 4 levels down.