**William Chen**

**405-131881**

**CS32 Project 3 Report**

**Description of Design**

The main data structure that was used was through vectors, which are essentially just arrays that can be resized when an element is added. This allowed me to easily reshape the size of the arrays depending on how many elements were called for when the board is initialized. Vectors also make it very easy to add elements in and the array format allows for easy manipulation. The design for game is rather straightforward, following the structure of the rules and making way for exceptions such as captures and free turns in Kalah. The Bad Player class creates a computer player that simply chooses the hole with the largest number of beans and “sows” this, essentially placing a bean in every hole after it aside from the opponent’s pot. The private data members such as the player pointers in Game were very important to manipulating the players regardless of if they were BadPlayers or SmartPlayer or a HumanPlayer.

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

If I had the opportunity to create SmartPlayer, the code would essentially search through choosing each of the moves to see if there was a move that would return a free turn starting from the spot closest to the player’s pot. This would allow the smartplayer to maximize the number of free turns it receives. If no free turns are found or if there are no more to be used, then the next best move would be a move that would allow the player to capture the beans of the opponents. This would mean that the code would have to run through all of the holes and determine if a capture is possible, which essentially means that the final bean lands in an open hole and the opponent has beans in the hole directly opposite of the empty hole that just received one. If this is not possible or if 5 seconds is reached before this, then the hole with the highest number of beans is called.

**Pseudocode**

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

Check if the hole given is invalid

if the side is south

Set the hole chosen to zero beans

While the number of beans taken from the initial hole is greater than zero

Place the beans in each hole after until it hits the pot

If there are still beans, place a bean in the pot; if not, break

If there are still beans, go through the north side and add one bean to each hole

if the side is north

Set the hole chosen to zero beans

While the number of beans taken from the initial hole is greater than zero

Place the beans in each hole after until it hits the pot

If there are still beans, place a bean in the pot; if not, break

If there are still beans, go through the south side and add one bean to each hole

}

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

{

Find the hole with the most amount of beans

If the max is zero, then there is an error and -1 is returned

Return the hole with the largest number of beans

}

bool Game::move(){

Create all varibales required including two player pointers, the ones required for status, and then set the first player pointer to the current side and the second player to the opposite side

Let the current player choose their move and then sow

Check if someone won

If the last bean is placed in the current players pot, then free turn is given

If the final bean lands in an open hole and the opponent has beans in the hole directly opposite of the empty hole that just received one, then a capture is made

Check if someone won

If not, continue and return true

}

void Game::play(){

While moves can still be made, allow the display to be created

If both players are computers, then require the human to input enter to contiue with the computer turns

Display the final board

Display the winner or if it is a tie

}

**Any known bugs, serious inefficiencies, or notable problems**

SmartPlayer is implemented the same way as BadPlayer

**Test Cases**

Board b(3, 3);

assert(b.holes() == 3);

assert(b.beansInPlay(SOUTH) == 9);

assert(b.beans(NORTH, 2) == 3);

assert(b.totalbeans() == 18);

Side endSide;

int endHole;

assert(b.sow(SOUTH, 2, endSide, endHole));

assert(b.beans(SOUTH, 2) == 0);

assert(b.beans(SOUTH, 3) == 4);

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

b.moveToPot(SOUTH, 3, SOUTH);

assert(b.beans(SOUTH, POT) == 5);

b.setBeans(SOUTH, 2, 0);

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

HumanPlayer human("Cheese");

assert(human.name() == "Cheese");

assert(human.isInteractive());

BadPlayer bad("Koreaman");

assert(bad.name() == "Koreaman");

assert(!bad.isInteractive);

SmartPlayer smart("Yes");

assert(smart.name() == "Yes");

assert(!smart.isInteractive);

Board b(3, 2);

b.setBeans(SOUTH, 2, 0);

b.setBeans(SOUTH, 1, 0);

assert(bad.chooseMove(b, SOUTH) == 3);

b.setBeans(NORTH, 3, 0);

b.setBeans(SOUTH, 1, 0);

assert(bad.chooseMove(b, SOUTH) == 2);

BadPlayer a("A");

BadPlayer b("B”);

Board b(3,2);

Game g(b, &A, &B);

g.display(); //This should show the Names A and B at the top and bottom of the board and the second and fourth row should be 3 numbers that all have 2 that represent the holes the third row should show two zeros that represent the pots

assert(g.beans(SOUTH, 2) == 2);

g.play(); //play should test status and move on its own

//The final display should show the holes having zeros and the pots should have different numbers

//A winner or a tie should be shown