**CS 32 Project 3 Zinnia Kwan**

**205777626**

**1. DATA STRUCTURES**

Game class:

* Ships were stored in a vector of ship structs. Each ship struct contained the ship’s length, name, and symbol, and the ship’s id that was passed into the functions of this game were their index in the vector.

Board class:

* The board was represented by a 2d array of chars, with ‘.’ representing an empty spot and the character of the ship on the spots it occupies.
* The ships that were placed onto the board, were also stored into the vector so that the board class could access/store information about each ship, such as it’s location, and the amount of damage it had taken.

Player class:

* Awful player and human player required no data structures.
* Mediocre Player:
  + My function that placed the ships recursively, required a way for me to access all the ships I needed to place. Therefore, all the ship ids were stored in a vector that sorted them from smallest length to greatest.
  + I also used a 2d array, to store the points that I had already previously attacked, marking spots with an ‘X’ when they were hit already.
* Good Player:
  + My function that placed the ships recursively, required a way for me to access all the ships I needed to place. Therefore, all the ship ids were stored in a vector that sorted them from smallest length to greatest.
  + I also used a 2d array, to store the points that I had already previously attacked, marking spots with an ‘X’ when they were hit already, and either did not hit a ship, or is a ship that sank. Otherwise, if I hit a ship without it sinking, those spots were marked with ‘@’.

**2. IMPLEMENTATION OF GOOD PLAYER:**

When placing my ships down, I did not change the implementation from mediocre player. Basically, it was the same process of blocking half the spots, then placing and removing ships until all the ships were able to be placed. The biggest part of my strategy for winning with goodplayer was using the attack and update attack functions. I left the recordAttackByOppenent function empty and never used it.

Similar to the function for mediocre player, I had two states, one where I had hit a ship and now my goodplayer was attacking the positions around that ship (state 2), and one where I was attacking positions blindly (state 1). However, I improved state 1, by not just selecting random points to attack, but instead, I would calculate the probability of each of the remaining ships to be located at each position in the board, and choose the point with the highest probability. I did this by creating a separate function (isPlacable) that returned a bool if a ship fit on a spot based on the points previously attacked, and looping through every single ship and coordinate on my board, incrementing the Points the ship could fit on by one. For state 2, I improved mediocre player’s implementation by checking in one direction at a time until the ship was destroyed. I would continue to recommend points to the north first until I hit a Point that was previously attacked or part of a sunken ship, then go south, east, and then west.

The only bug that causes my goodplayer odds of winning to decrease a lot, are when its opponent places their ships in a fashion that they are all grouped next to each other and going in different directions.

**3. PSEUDOCODE:**

**Game.cpp**

string GameImpl::actionName( bool shotHit, bool shipDestroyed, int shipId){

if the shot is unsuccessful, return “missed”;

if ship is destroyed return "hit something";

else return "destroyed the “ name of ship;

}

Player\* GameImpl::play(Player\* p1, Player\* p2, Board& b1, Board& b2, bool shouldPause)

{

if either player is not able to successfully place their ships{

return nullptr;}

create two bools p1viewShotsOnly and p2viewShotsOnly , and set them to

true if their players are human;

while neither of the players have all their ships destroyed{

for the odd rounds{

Print “ Player 1’s name’s turn. Board for player 2’s name: endl”;

display player 2’s board;

Get recommended attack from player 1;

create 3 bools: shotHit, shipDestroyed, validShot;

create int shipId;

if the shot is not successful {

set validShot=false;

print “P1’s name wasted a shot at point.” << endl;

}

else {

set validShot=true;

create a string action and call actionName function to get a

value for it

print “P1’s name attacked point and " action ", resulting in:"

<< endl;

display P2’s board;

}

record p1’s attack;

record attack by opponent for player 2

}

for the even rounds{

do the same thing as for odd, except reverse player 1 and 2;

}

if we should pause and the game is not over,{

call waitforenterfunction ;

}

}

Display the winner’s name, and “ wins!”;

if all of player 1’s ships are destroyed

{if p1 is human, display player2’s board showing the ships}

}

else {

{if p2 is human, display player1’s board showing the ships}

}

return winner;

}

**board.cpp**

bool BoardImpl::placeShip(Point topOrLeft, int shipId, Direction dir)

{

return false if the ship id does not exist;

for all of the ships already place{

if shipId is already placed, return false;

}

if top or left does not exist on the board, return false;

if the direction is horizontal{

if the ship falls off the board, return false

for each space on the board that will be place{

if spot on board is not open, return false;

}

for each space on the board that will be place{

set point on the board to have the ship’s character;

}

}

else if direction is vertical{

if the ship falls off the board, return false;

for each space on the board that will be place{

if spot on board is not open, return false;

}

for each space on the board that will be place{

set point on the board to have the ship’s character;

}

}

push that temp with the shipId, coordinate, and length onto the idPlaced vector;

return true;

}

bool BoardImpl::unplaceShip(Point topOrLeft, int shipId, Direction dir)

{

repeatedly:

while you do not reach the end of the idPlaced vector

look through every single ship in the idPlaced vector until you find the

ship with the same id;

if you can’t find the shipid {return false;}

repeatedly:

look through every single point in the grid;

if the point at the grid has the same character as the shipId{

set that point into a variable called location

}

}

}

if location does not equal top or left, return false;

Set every single point in the board which the shipId takes up back to a ‘.’;

erase that ship from the idPlaced vector;

return true;

}

bool BoardImpl::attack(Point p, bool& shotHit, bool& shipDestroyed, int& shipId)

{

initialize all the bools to be false in the beginning;

return false if the point is invalid, or already hit;

if the point hit is not ‘.’, it must be a ship so:

Set shot hit to true;

shipsPlaced finder;

Loop through all the ships placed on the board, and we increase the

amount of damage

Find the ship that was hit using the character at the location,

Decrement the amount of undamaged pieces for that ship.

Set that place in the grid to an ‘X’

If the ship is destroyed because there are no remaining undestroyed pieces, change shipDestoryed to true;

}

else {

set the point to ‘o’ because the player missed;

}

return true;

}

bool BoardImpl::allShipsDestroyed() const

{

for every single point in the board{

if the point does not equal ‘.’, ‘o’, ‘X’, return false

}

return true;

}

**Player.cpp**

**HumanPlayer:**

bool HumanPlayer::placeShips(Board &b){

Display the number of ships human must place;

Repeatedly:

until all the ships are placed,

tell the player to enter h or v,

continue prompting for input until they enter either h or v;

tell the player to enter a row and column number

continue prompting for input until the ship can be successfully

placed at that point

}

return true;

}

Point HumanPlayer::recommendAttack(){

Tell player to enter row and column to attack;

Continue prompting for the correct input;

return that point;

}

**MediocrePlayer:**

void MediocrePlayer::getAllShips(const Game & m\_game, vector<ship> & SHIPS){

push all ships from the game into the SHIPS vector;

}

bool MediocrePlayer::placeRecursively(Board &b ,const Game & g, const vector<ship> SHIPS ){

return true if there are no more ships in SHIPS vector;

create a vector with one less ship that SHIPS;

for every point in the board:

if we can successfully place the last ship in the array horizontally{

call the function again recursively, except using the copied

array with one less ship

if it returns true, we return true, else remove ship

}

else if we can successfully place the last ship in the array vertically{

call the function again recursively, except using the copied

array with one less ship

if it returns true, we return true, else remove ship

}

return false;

}

bool MediocrePlayer::placeShips(Board &b){

create a vector called ShipsInOrder with all the ships sorted by size;

repeat until we get to 50 iterations:{

call the block function;

if we can successfully place our ships, we unblock and return true;

call the unblock function;

}

return false since we couldn’t place successfully in 50 tries;

}

Point MediocrePlayer::recommendAttack(){

if in state 1{

Find the number of possible points to attack that have not been already;

Randomly select and return one of those points;

}

else {

randomly select one of the ships in the Possible Attacks vector

remove that ship from the vector

return that ship;

}

}

void MediocrePlayer::recordAttackResult(Point p, bool validShot, bool shotHit,

bool shipDestroyed, int shipId){

if we are in state one

set that point in our trackAttacked array to ‘X’;

if the shotHit without the ship being destroyed;

store that point in currentAttacking

call the get all attacks function to place all points in the

cross into our vector

set state to 2;

}

if we destroy the ship and are in state 2{

clear the PossibleAttacks vector;

set back to state 1;

}

}

if there are no more ships to attack in the PossibleAttacks vector{

set back to state 1;

}

}

bool MediocrePlayer::validPoint(Point p){

if point is off board return false;

if point is already hit return false;

else return true;

}

**Goodplayer:**

GoodPlayer::GoodPlayer(string nm, const Game& g) : Player(nm, g),currentShipDest(true), currentAttacking(0, 0), d(undecided){

Set cols and rows equal to game’s cols and rows;

Initialize every point in trackAttacked to ‘.’

Set the probability vector to have the size of the game, and set every point to zero

initialize the shipsToFind vector to have every ship id;

}

bool GoodPlayer::placable(Point p, Direction d, int shipId){

If the point is off the grid return false;

if the direction is horizontal{

if the ship doesn’t fit on grid return false;

if any of the points on the board where the ship is going to be placed

is ‘X’ return false

}

else direction is vertical{

if the ship doesn’t fit on grid return false;

if any of the points on the board where the ship is going to be placed

is ‘X’ return false

}

return true;

}

void GoodPlayer::FindProb(){

for every single ship we have to place

for every single spot on the board

if we can place the ship on the board here (first horizontal, then vertically)

increment the probability vector for every single point

where the ship lands, unless the spot is a point that is part of an unsunken ship

}

bool GoodPlayer::getPointToAttack(Point &p){

if direction is north{

repeatedly:

move point in the upward direction,

if that point is not hit before, return true;

if we hit an invalid point, change the direction to south;

}

if (d==SOUTH){

repeatedly:

move point in the downward direction,

if that point is not hit before, return true;

if we hit an invalid point, change the direction to east; }

if (d==EAST){

repeatedly:

move point in the eastward direction,

if that point is not hit before, return true;

if we hit an invalid point, change the direction to west;

}

if (d==WEST){

repeatedly:

move point in the westward direction,

if that point is not hit before, return true;

if we hit an invalid point, break;

}

return false;

}

Point GoodPlayer::recommendAttack(){

If we are in state 2{

if getPointToAttack returns true then return that point;}

call clearProb and FindProb;

Return the point with the maximum probability in the probability vector;

if the probability of every point is zero, return a random point that has

not been hit before;

}

void GoodPlayer::recordAttackResult(Point p, bool validShot, bool shotHit,

bool shipDestroyed, int shipId){

if it was a valid shot and didn’t hit, set that point in trackAttacked to

‘X’;

else if shotHit

set that point to an ‘@’;

if you are in state one and hit a ship{

change to state 2, and record the point hit in currentAttacking, and set

d to north

}

if we are in state 2 and the ship is destroyed{

change to state 1;

call removeAts(p, d, shipId)to set the destroyed ship’s points to

an ‘X’;

change direction back to undecided;

remove that ship from ships to find;

currentShipDest=true;

}

if you are in state 2 and miss a shot, switch to the next direction;

if you are in west and miss, switch to state 1;

}

bool GoodPlayer::placeShips(Board &b){

the implementation is the same as mediocre player’s, except it enters

this loop 250 times

If we are unable to place, pass into place recursively without blocking

any points;

return false;

}

void GoodPlayer::removeAts(const Point & p, Dir d, int shipId){

if direction is north

While we do not repeat more than the length of the ship

set the points from in trackAttacked from p to

CurrrentAttacking to ‘X’;

}

Count how many ‘@’ there are above and below the points we set to an

‘X’

If the the total number of points above and below + the number of points {

set to ‘X’ is greater than the ship length, we return and do not remove

those @’s

(this is because they may belong to another ship that has not been sunk)

}

Else{

set all the points above and below that are ‘@’ to zero;

The code repeats itself for each direction, except for east and west we check points to left and right

}