CS32 Battleship Lab Report

Eli Foerst

205702952

**Data Structures:**

*Board* — Board is designed using a private data member called *m\_grid* which is an array of MAXROWS, MAXCOLS. When implementing the boardimpl class the grid board creates (associated with Game g) consists of ‘.’ characters to simulate water. Invalid operands for a Board to use to place a ship are ‘.’, ‘o’, ‘X’, and ‘#’ (where # is the symbol for block()).

*Game* — Game utilizes a class called *LogShips* which is utilized for a vector of this type named *m\_log*. This log tracks all the properties of applicable ships to use in gameplay.

*Mediocre Player —* The mediocre player uses vector *attackLog* of class of *LogAttacks* to track the properties of each attack (was a ship hit, was a ship destroyed, if so what is it’s id?). Additionally mediocre uses stack *m\_recall* of type *Point* to keep track of ship placement until all ships are placed, this ensures the code is able to unplace a ship when necessary.

*Good Player* — In addition to the data structures of *mediocre player,*  good player uses stack *m\_nextOne* of type *Point* to keep track of point’s attacked. When a player successfully attacks a ship segment *m\_nextOne* pushes the point onto the stack as it will use it to test the surrounding points, to ensure that all ship segments are hit AND to keep track of users who place ships close to one another.

**Good Player Strategy**:

A good player utilizes the same recursive ship placement call as the mediocre player but adjusts the number of tries to 100 attempts, as this is the product of MAXROWS, MAXCOLS it makes sense that the good player would be unable to place after this many attempts… but lets say life is hard and it doesn’t work, then Good Player resorts to another form of ship placement where it places ships randomly.

A good player attacks at only even intervals. Think about a 10x10 grid. If every other grid point were a ‘O’ versus a ‘.’ It would look like a checkerboard. This is the idea good player uses. It ONLY goes after random points in these “diagonals” until it hits a ship. Once it does it searches around it for another segment and follows the segment until it can’t anymore. Once it finishes a segment it resorts back to the log’s last value (stack *top*) and tries again. Thus following any and all connected ships, whilst checking for “neighborly” ships.

**Pseudocode:**

*Mediocre Player* —

*placeShips():*

Loop up to 50 times{

Block half of the cells

If the recursive call can place all ships then return true and deallocate the stack.

Unblock the cells

Try again

}

If 50 tries went by then deallocate the stack and return false

*placedShips()* **(recursive function):**

If the shipId is the last ship return true

If the column is “out of bounds” move down a row and to that start of all columns

If the ship can be placed horizontally push the point onto the stack. Call recursion for next ship.

If recursion didnt work unplaced ships from the stack (BACKTRACK) and return the function recursively for the next column over.

Do the same for vertical

At the end… if the row is “out of bounds” return false, otherwise… return the function recursively for the next column over.

*Good Player—*

*recommendAttack()*:

If in state 1 attack a random original “diagonal” point so long as there are “diagonal” points left to attack. If there are still elements in the point log stack switch to state 2. If there are no “diagonal” original points remaining then just try a random original point.

If in state 2 attack a random original point surrounding the point of the last attack. If all directions of said attack have been tried then pop the top of the stack and adjust to the new top (BACKTRACK). If the stack contains no more elements then choose a random point.

*recordAttackResult()*:

The only thing changed was the addition of this line:

**if** (shotHit == **true** && shipDestroyed == **false**)

m\_nextOne.push(attackLog[attackLog.size()-1].attackPoint());

Within the state 2 if statement.

None of the players used *recordAttackByOpponent()* . Personally I didn’t think it would be useful because battleship is all about chance and knowing your opponent. An AI can get to know you over time but a 15 minute max game of battleship one time (before memory wipe) is not going to teach it much. For the future I plan to apply a rule to this game where once it has to attacks right next to each other it continues in the direction until a ship is destroyed. Though this is not needed as the Good Player still averages a 80% lead on the mediocre player.

**Testing Code/ Main statement:**

// Tests for HumanPlayer

/\*

int main(){

Game g(6,3);

g.addShip(5, 't', "Tributary");

g.addShip(5, 'c', "Clown");

g.addShip(5, 'l', "Loser");

g.addShip(4, 'w', "Won't Work");

Board b(g);

HumanPlayer me("Johnny", g);

me.placeShips(b);

me.recommendAttack();

}

\*/

// Tests for MediocrePlayer

/\*

int main(){

Game g(10,10);

g.addShip(5, 'a', "aircraft carrier");

g.addShip(4, 'b', "battleship");

g.addShip(3, 'd', "destroyer");

g.addShip(3, 's', "submarine");

g.addShip(2, 'p', "patrol boat");

Board b(g);

MediocrePlayer you("Mancy", g);

you.placeShips(b);

b.display(false);

}

\*/

// Tests for GoodPlayer

/\*

int main(){

Game g(10,10);

g.addShip(5, 'a', "aircraft carrier");

g.addShip(4, 'b', "battleship");

g.addShip(3, 'd', "destroyer");

g.addShip(3, 's', "submarine");

g.addShip(2, 'p', "patrol boat");

GoodPlayer us("Charles", g);

Board b(g);

assert(us.placeShips(b));

Game l(2,2);

l.addShip(2, 's', "SS Sarah");

l.addShip(2, 'l', "St. Lee");

Board f(l);

GoodPlayer eli("Eli", l);

eli.placeShips(f);

b.display(false);

f.display(false);

}

\*/

/\*

int main(){

Game g(10,10);

g.addShip(5, 'q', "charmander");

g.addShip(3, 'l', "james");

Board b(g);

//b.display(false);

Point t(0,0);

Point p(1,9);

b.placeShip(t, 0, HORIZONTAL);

b.placeShip(p, 1, VERTICAL);

bool shot = false, shipGone = false;

int id = -1;

for (int i = 0; i < g.shipLength(0); i++){

Point wow(0, i);

b.attack(wow, shot, shipGone, id);

assert(shot);

}

b.display(false);

assert(shipGone);

assert(id == 0);

cout << g.shipName(id) << " was destroyed in " << g.shipLength(id) << " hits! :D" << endl;

}

\*/

/\*

int main(){

Game g(10,10);

Board b(g);

b.block();

b.display(false);

}

\*/

#include "Game.h"

#include "Player.h"

#include <iostream>

#include <string>

**using** **namespace** std;

**bool** addStandardShips(Game& g)

{

**return** g.addShip(5, 'A', "aircraft carrier") &&

g.addShip(4, 'B', "battleship") &&

g.addShip(3, 'D', "destroyer") &&

g.addShip(3, 'S', "submarine") &&

g.addShip(2, 'P', "patrol boat");

}

**int** main()

{

**const** **int** NTRIALS = 100;

cout << "Select one of these choices for an example of the game:" << endl;

cout << " 1. A mini-game between two mediocre players" << endl;

cout << " 2. A mediocre player against a human player" << endl;

cout << " 3. A " << NTRIALS

<< "-game match between a mediocre and an awful player, with no pauses"

<< endl;

cout << " 4. A " << NTRIALS

<< "-game match between a good player and a mediocre player, with no pauses"

<< endl;

cout << " 5. A good player against a human player" << endl;

cout << " 6. A human player against a human player" << endl;

cout << "Enter your choice: ";

string line;

getline(cin,line);

**if** (line.empty())

{

cout << "You did not enter a choice" << endl;

}

**else** **if** (line[0] == '1')

{

Game g(2, 3);

g.addShip(2, 'R', "rowboat");

Player\* p1 = createPlayer("mediocre", "Popeye", g);

Player\* p2 = createPlayer("mediocre", "Bluto", g);

cout << "This mini-game has one ship, a 2-segment rowboat." << endl;

g.play(p1, p2);

**delete** p1;

**delete** p2;

}

**else** **if** (line[0] == '2')

{

Game g(10, 10);

addStandardShips(g);

Player\* p1 = createPlayer("mediocre", "Mediocre Midori", g);

Player\* p2 = createPlayer("human", "Shuman the Human", g);

g.play(p1, p2);

**delete** p1;

**delete** p2;

}

**else** **if** (line[0] == '3')

{

**int** nMediocreWins = 0;

**for** (**int** k = 1; k <= NTRIALS; k++)

{

cout << "============================= Game " << k

<< " =============================" << endl;

Game g(10, 10);

addStandardShips(g);

Player\* p1 = createPlayer("awful", "Awful Audrey", g);

Player\* p2 = createPlayer("mediocre", "Mediocre Mimi", g);

Player\* winner = (k % 2 == 1 ?

g.play(p1, p2, **false**) : g.play(p2, p1, **false**));

**if** (winner == p2)

nMediocreWins++;

**delete** p1;

**delete** p2;

}

cout << "The mediocre player won " << nMediocreWins << " out of "

<< NTRIALS << " games." << endl;

// We'd expect a mediocre player to win most of the games against

// an awful player. Similarly, a good player should outperform

// a mediocre player.

}

**else** **if** (line[0] == '4')

{

**int** nGoodWins = 0;

**for** (**int** k = 1; k <= NTRIALS; k++)

{

cout << "============================= Game " << k

<< " =============================" << endl;

Game g(10, 10);

addStandardShips(g);

Player\* p1 = createPlayer("mediocre", "Mediocre Mimi", g);

Player\* p2 = createPlayer("good", "Good Gracie", g);

Player\* winner = (k % 2 == 1 ?

g.play(p1, p2, **false**) : g.play(p2, p1, **false**));

**if** (winner == p2)

nGoodWins++;

**delete** p1;

**delete** p2;

}

cout << "The good player won " << nGoodWins << " out of "

<< NTRIALS << " games." << endl;

// We expect goodPlayer to win more often

}

**else** **if** (line[0] == '5')

{

Game g(10, 10);

addStandardShips(g);

Player\* p1 = createPlayer("good", "Good Gracie", g);

Player\* p2 = createPlayer("human", "Shuman the Human", g);

g.play(p1, p2);

**delete** p1;

**delete** p2;

}

**else** **if** (line[0] == '6')

{

Game g(10,10);

addStandardShips(g);

Player\* p1 = createPlayer("human", "Sarah", g);

Player\* p2 = createPlayer("human", "Eli", g);

g.play(p1, p2);

**delete** p1;

**delete** p2;

}

**else**

{

cout << "That's not one of the choices." << endl;

}

}