* A description of the design of your data structures. For example, how do you represent the board? How do you record shots made?

class OnCrossData

{

public:

OnCrossData();

OnCrossData(Point point, int length, Direction dir);

Point getpoint() const;

int getlength() const;

Direction getdir() const;

private:

Point m\_point;

int m\_length;

Direction m\_dir;

};

------------------------------------------------------------------------------------------------------------------------------------------

I used class OnCrossData to keep to keep track of the length of a ship, its orientation (VERTICAL or HORIZONTAL) and the coordinate of its topmost or leftmost part depending on its orientation.

------------------------------------------------------------------------------------------------------------------------------------------

class Orientations

{

public:

Orientations();

Orientations(int rows, int cols, int shiplength, Point point);

Orientations(int rows, int cols, int shiplength, Point point, std::vector<Point> Attacked);

bool isValid(Point p, int rows, int cols) const { return((p.r >= 0) && (p.r < rows) && (p.c >= 0) && (p.c < cols)); }

bool isValid(Point p, std::vector<Point> Attacked) const;

bool isWholeShipValid(int m\_rows, int m\_cols, int shiplength, Point point, Direction Dir); // checks if the ship will fit on the board

bool isWholeShipValid(int rows, int cols, int shiplength, Point point, Direction Dir, std::vector<Point> Attacked);

bool isPointOnShip(Point point, OnCrossData Ship); // checks if point is on the ship

int reEvaluateOrientations(bool HitOrMiss, Point Attack); // returns number of new orientations removed

int emptyVector();

const int getSize() const { return(m\_size); }

const Point getPosition() const { return(m\_position); }

Direction getDirZero() { if (m\_orientation\_array.size() == 1) { return(m\_orientation\_array[0].getdir()); } }

private:

Point m\_position;

std::vector<OnCrossData> m\_orientation\_array; // possible orientation

int m\_size;

};

------------------------------------------------------------------------------------------------------------------------------------------

I used class Orientations to keep to keep track the way a ship could be oriented given a square. Point m\_position stores the square that we are considering. std::vector<OnCrossData> m\_orientation\_array stores the possible orientations which are defined by a ship orientation, location of its topmost or leftmost position, and the length of the ship. int m\_size stores the total number of orientations possible which also happends to be size of the vector m\_orientation\_array.

int reEvaluateOrientations(bool HitOrMiss, Point Attack) returns number of new orientations removed given if a shot resulted in a hit or miss (if it’s a hit, then the ships (OnCrossData) that would have been hit are kept and all other ships are discarded.

------------------------------------------------------------------------------------------------------------------------------------------

class Distribution

{

public:

Distribution();

Distribution(const Game& g, int shiplength);

Distribution(const Game& g, int shiplength, std::vector<Point> Attacked);

bool isValid(Point p) const { return((p.r >= 0) && (p.r < m\_rows) && (p.c >= 0) && (p.c < m\_cols)); }

//void getMax(int& max, Point& Location);

void reEvaluateDist(bool HitOrMiss, Point Attack, Point CenterOfCross);//, int& total\_total\_orientations);

//const int getTotalOrientations() const { return(m\_total\_orientations); }

const int getShipLength() const { return(m\_shipLength); }

const Orientations getPosOrientation(int r, int c) const { return(VariationShip[r][c]); }

private:

Orientations VariationShip[MAXROWS][MAXCOLS];

int m\_shipLength;

int m\_rows;

int m\_cols;

//int m\_total\_orientations;

};

------------------------------------------------------------------------------------------------------------------------------------------

I used class Distribution to keep to keep track of all the orientations that are possible for a given ship in a given game. Orientations VariationShip[MAXROWS][MAXCOLS] is an array that stores all the squares and the possible orientations a ship of a specified size can take on. The length of the ship is stored in int m\_shipLength. The size of the board is stored in int m\_rows and int m\_cols. void reEvaluateDist(bool HitOrMiss, Point Attack, Point CenterOfCross) updates the Orientations VariationShip[MAXROWS][MAXCOLS] or the number of orientations per square accordingly (if there was a miss then the number of orientations on that square are zero, and the number of orientations on the adjacent squares are also affected)

------------------------------------------------------------------------------------------------------------------------------------------

class TotalDistribution

{

public:

TotalDistribution(const Game& g);

Point MostProbableLocation(const Game& g);

void UpdateDist(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId, const Game& g);

std::vector<Point>& getAttacked() { return(m\_attacked); }

bool isValid(Point p) const { return((p.r >= 0) && (p.r < m\_rows) && (p.c >= 0) && (p.c < m\_cols)); }

void adjustRegisteredHits(int shipLength);

private:

void storeProbInf(const Game& g, Distribution dist);

int m\_probability[MAXROWS][MAXCOLS];

std::vector<Distribution> DistGame;

std::vector<int> m\_shipId;

std::vector<Point> m\_attacked;

std::vector<Point> m\_hits;

bool m\_justDestroyedShipButThereAreHitSquares;

int m\_nShips;

bool m\_FoundCross;

Point m\_CrossCenter;

int m\_rows;

int m\_cols;

//int m\_total\_total\_orientations;

};

------------------------------------------------------------------------------------------------------------------------------------------

int m\_rows and int m\_cols stores the size of the board. std::vector<Distribution> DistGame is a vector that stores the Probability Distributions for all the ships in the game. std::vector<int> m\_shipId stores the shipIds of all the ships present on the board.

int m\_probability[MAXROWS][MAXCOLS] stores the sum of all Probability Distributions for all the squares on the board (i.e. the probability that a ship is on a given square (doesn’t specify the type of ship unlike distributions)). It is used to return the point at which the probability of there being a ship is the highest. std::vector<Point> m\_attacked stores all the points that have been attacked so far. std::vector<Point> m\_hits stores all the points at which a ship has been hit before a ship has not been destroyed. bool m\_justDestroyedShipButThereAreHitSquares is true when a ship has been destroyed but there are points at which another ship has been hit but not destroyed. int m\_nShips stores the number of ships in the game. bool m\_FoundCross is a flag indicating that the GoodPlayer will attack in a cross pattern around a square which resulted in a hit but not a destroyed ship. Point m\_CrossCenter stores the center of this cross. void UpdateDist(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId, const Game& g) updates all the distributions after a shot has been taken. Point MostProbableLocation(const Game& g) returns the point at which a ship is most likely to be stationed at. However, if bool m\_justDestroyedShipButThereAreHitSquares is true and m\_hits is not empty, it attacks in a cross pattern around m\_hits[0] until it sinks a ship. If it can’t sink a ship it recommends a random point. void adjustRegisteredHits(int shipLength) removes points from the m\_hits vector once a ship is destroyed. These points correspond to the points which the ship that was sunk occupied.

------------------------------------------------------------------------------------------------------------------------------------------

class Ship

{

public:

Ship(int length, char symbol, std::string name, int shipId);

private:

std::string m\_name;

char m\_symbol;

int m\_length;

int m\_shipId;

};

------------------------------------------------------------------------------------------------------------------------------------------

I used class Ship to keep to keep track of the name, length, shipId, and symbol of a given ship.

------------------------------------------------------------------------------------------------------------------------------------------

template < class T>

class Node

{

public:

Node() {}

Node(T theData, Node<T>\* theLink) : data(theData), link(theLink) {}

Node<T>\* getLink() const { return link; }

const T getData() const { return data; }

void setData(const T& theData) { data = theData; }

void setLink(Node<T>\* pointer) { link = pointer; }

private:

T data;

Node<T>\* link;

};

------------------------------------------------------------------------------------------------------------------------------------------

Using class Node I implemented singly-linked list in class GameImpl whose head is Node<Ship>\* m\_head. This list does not have a dummy node and is not circular. The list keeps track of all ships added into the game.

I also used multiple singly-linked lists in class MediocrePlayer whose heads are Node<Point>\* m\_head, Node<Point>\* m\_cross. This helped me keep track of the positions that the Mediocre Player attacked and the positions it had already attacked when it was in state 2.

------------------------------------------------------------------------------------------------------------------------------------------

class GameImpl

{

public:

GameImpl(int nRows, int nCols);

~GameImpl();

int rows() const;

int cols() const;

bool isValid(Point p) const;

Point randomPoint() const;

bool addShip(int length, char symbol, string name);

int nShips() const;

int shipLength(int shipId) const;

char shipSymbol(int shipId) const;

string shipName(int shipId) const;

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

private:

void AskAttack(Player\* p1, Player\* p2, Board& b1, Board& b2, bool shouldPause);

int m\_rows;

int m\_cols;

Node<Ship>\* m\_head;

int m\_nShips;

}; ------------------------------------------------------------------------------------------------------------------------------------------

class GameImpl stores:

1. the size of the board through int m\_rows, int m\_cols
2. the number of ships through int m\_nShips
3. the ships that are allowed in the game are stored in a linked list of type Ship whose head is Node<Ship>\* m\_head
4. AskAttack function is a helper function for the play function and it writes to cout

------------------------------------------------------------------------------------------------------------------------------------------

class BoardImpl

{

public:

BoardImpl(const Game& g);

void clear();

void block();

void unblock();

bool placeShip(Point topOrLeft, int shipId, Direction dir);

bool unplaceShip(Point topOrLeft, int shipId, Direction dir);

void display(bool shotsOnly) const;

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

bool allShipsDestroyed() const;

private:

char m\_board[MAXROWS][MAXCOLS];

int m\_rows;

int m\_cols;

const Game& m\_game;

};

------------------------------------------------------------------------------------------------------------------------------------------

class BoardImpl stores:

1. the size of the board through int m\_rows, int m\_cols
2. **the representation of the board through char m\_board [MAXROWS][MAXCOLS]**
3. the game its being used in through const Game& m\_game

------------------------------------------------------------------------------------------------------------------------------------------

class HumanPlayer : public Player

{

public:

HumanPlayer(string nm, const Game& g);

virtual bool isHuman() const;

virtual bool placeShips(Board& b);

virtual Point recommendAttack();

virtual void recordAttackResult(Point p, bool validShot, bool shotHit,

bool shipDestroyed, int shipId);

virtual void recordAttackByOpponent(Point p);

private:

};

------------------------------------------------------------------------------------------------------------------------------------------

This class represents a Human player. A game played as this class will ask the user for inputs to the game

------------------------------------------------------------------------------------------------------------------------------------------

class MediocrePlayer : public Player

{

public:

~MediocrePlayer();

MediocrePlayer(string nm, const Game& g);

//std::string name() const { return m\_name; }

//const Game& game() const { return m\_game; }

virtual bool placeShips(Board& b);

virtual Point recommendAttack();

virtual void recordAttackResult(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId);

virtual void recordAttackByOpponent(Point p);

private:

bool recursivePlace(Board& b, int nShips, Point point, bool backtrack);

bool recursivePlace(Board& b, int nShips, Point point, bool backtrack, int& depth);

int Lvalid(Point point);

int Rvalid(Point point);

int Uvalid(Point point);

int Dvalid(Point point);

bool doneCross();

int m\_state;

Node<Point>\* m\_head;

Point m\_lastCellAttacked;

Point m\_centerOfCross;

int u\_size;

int d\_size;

int l\_size;

int r\_size;

Node<Point>\* m\_cross;

};------------------------------------------------------------------------------------------------------------------------------------------

This class represents a Mediocre player. This player behaves as specified in the spec. The m\_cross Node<Point> pointer is used to keep track of attacks in a cross pattern centered at m\_centerOfCross when the player gets a hit and the ship that was hit has not been destroyed. This cross is specified by u\_size, d\_size, l\_size, r\_size which are given a useful value by the Uvalid(..), Dvalid(…), Lvalid(..), Rvalid(…) functions. m\_head Node<Point> pointer stores all the points on the board that have been attacked thus far. m\_state stores the state that the player is in and m\_lastCellAttacked stores the last point that was attacked. The bool doneCross() function determines if all parts of the cross has been attacked when the player is in state 2, and is used as a flag to transition back into state 1.

------------------------------------------------------------------------------------------------------------------------------------------

class GoodPlayer : public Player

{

public:

GoodPlayer(string nm, const Game& g);

~GoodPlayer();

virtual bool placeShips(Board& b);

virtual Point recommendAttack();

virtual void recordAttackResult(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId);

virtual void recordAttackByOpponent(Point p);

private:

bool CheckCross(Point ourShip, Node<OnCrossData>\* ThierShip);

bool OnCross(Board& b, int shipLength, Point point, Direction Dir);

bool recursivePlace(Board& b, Point point, int shipId, int i);

bool recursivePlace(Board& b, int nShips, Point point, bool backtrack);

Node<OnCrossData>\* m\_head;

TotalDistribution TransitionMatrix;

int m\_rows;

int m\_cols;

};

------------------------------------------------------------------------------------------------------------------------------------------

This class represents a Good player. m\_rows and m\_cols stores the size of the board. The m\_head Node<OnCrossData> pointer is used to keep track of the cross pattern in which the MediocrePlayer attacks. If the game has more than 3 ships, GoodPlayer places its first 3 ships in such a way that they are not caught in a crossfire if one of them gets hit(i.e. the cross attack pattern doesn’t hit one of the other ships). Then, half of the board is blocked. The rest of the ships are placed with the same recusive algorithm that the MediocrePlayer uses. If that fails, then we clear the board and replace all the ships just like the MediocrePlayer. If we can’t place after 50 tries then placeShips(Board& b) returns false.

If there are less than 3 ships then we place ships just like the MediocrePlayer. TotalDistribution TransitionMatrix is a data structure that stores all the possibilities of there being a ship in a given game.

It is used to recommend attacks and recordAttackresults.

------------------------------------------------------------------------------------------------------------------------------------------

* A prose description of your GoodPlayer's strategies for placing ships and recommending moves. If they're easy to describe, this may well be only a paragraph or two. This is a high-level summary; details go in the pseudocode.

If the game has more than 3 ships, GoodPlayer places its first 3 ships in such a way that they are not caught in a crossfire by MediocrePlayer if one of them gets hit(i.e. the cross attack pattern doesn’t hit one of the other ships). Then, half of the board is blocked. The rest of the ships are placed with the same recusive algorithm that the MediocrePlayer uses. If that process fails, then we clear the board and replace all the ships just like the MediocrePlayer. If we can’t place after 50 tries then placeShips(Board& b) returns false. If there are less than 3 ships then we place ships just like the MediocrePlayer. To recommend moves we use the TotalDistribution class. This class essentially the implementation of a Probabilty Density function. I update this data structure according to the history of the shots taken. It always returns one of the squares with the highest probability at which there will be a ship. However, when the game comes close to an end what ends up happening is all the squares end up having zero probability or the same square gets shot at repeadetly. When this happens, the GoodPlayer looks at the squares which have been hit but are not parts of the squares in which a ship has been sunk. Goodplayer attacks squares around this point in a cross pattern. If a ship is sunk then it operates in the way described above. If a ship is not sunk and we attacked in this cross pattern around all the squares that have been hit and are not a part of a ship that has been sunk yet. If this does not sink a ship then we resort to guessing random points on the board despite if they have been attacked before or not.

* Pseudocode for non-trivial algorithms. For example, how does a MediocrePlayer place ships? How does your GoodPlayer implement its strategies?

------------------------------------------------------------------------------------------------------------------------------------------

bool GameImpl::addShip(int length, char symbol, string name)

Create a new ship with int length, char symbol, string name and a distinct int shipId

Add a new ship after the last element of the linked list pointed to by m\_head

(handle empty linked list case seperately)

Increment m\_nShips

return(true)

------------------------------------------------------------------------------------------------------------------------------------------

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

While outputting the necessary cout statements

Display p2’s board b2 accordingly

Call p1’s recommendAttack() (for a human this should ask for coordinates to execute the attack on)

Call b2’s attack member function. Depending on its output cout the correct sentences. If the shot lands on the board b2, display b2 accordingly.

If shouldPause == true , then call waitForEner()

------------------------------------------------------------------------------------------------------------------------------------------

void BoardImpl::block()

while R\*C/2 of the board is not blocked

iterating through m\_board starting from m\_board[0][0] to m\_board[0][g.cols()-1] to m\_board[g.cols()–1][g.rows()–1]

block a position 50% of the time if the position was empty previously

------------------------------------------------------------------------------------------------------------------------------------------

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

If shipId is bigger or equal to m\_game.nShips() or topOrLeft is an inValid point return false

else create a ship with the attributes of the ship added to the game with an m\_shipId == int shipId.

Check if the ship is already added to the board. If so, return false

Check if the ship can fit on the board (this check depends on Direction dir). If not, return false

Check if the positions that the ship is going to occupy are empty (hence also unblocked). If not, return false.

Add the ship to the board and return true.

------------------------------------------------------------------------------------------------------------------------------------------

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

If shipId is bigger or equal to m\_game.nShips() or topOrLeft is an inValid point return false

else create a ship with the attributes of the ship added to the game with an m\_shipId == int shipId. We want to unplace this ship:

Check if the ship can fit on the board (this check depends on Direction dir). If not, return false.

Check if the ship with the is already added to the board at the specified location with the specified length. If not, return false

remove the ship from the board and return true.

------------------------------------------------------------------------------------------------------------------------------------------

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

If point p is not valid return false

If m\_board[p.r][p.c] == X or m\_board[p.r][p.c] == o , return false

If m\_board[p.r][p.c] == . , then set shothit == false update board, return true

Else set shothit == true. Store the symbol and update the board. Then, find the shipId of the ship that was hit using the symbol. Set int& shipId to the shipId you found. Then, determine if the ship was destroyed or not. If it was destroyed set bool& shipDestroyed to true. Otherwise. Set it equal to false. Return true.

------------------------------------------------------------------------------------------------------------------------------------------

bool HumanPlayer::placeShips(Board& b)

while outputting the necessary cout statements

for nShips() number of ships

display the board appropriately

ask the user for a direction

while the direction is not valid, keep asking for the direction

store the direction

ask for the row and column you want to place the ship

while the row and column are not valid or the ship cannot be placed, keep asking for the row and column

store the row and column in a point

place the ship

return true

------------------------------------------------------------------------------------------------------------------------------------------

void MediocrePlayer::recursivePlace(Board& b, int nShips, Point point, bool backtrack)

if nShips == game().nShips() return true

if backtrack == false

if we are at ((game().rows() – 1),game().cols() – 1))

if nShips == 0 return false

if nShips == game().nShips-1 and we can place the ship, return true

backtrack = true

call rescursiveplace with nShips – 1

if ship can be placed VERTICAL, place it, return recursivePlace with nShips + 1

if ship can be placed HORIZONTAL, place it, return recursivePlace with nShips + 1

move to the next point and return rescursivePlace with that point

else

if you can unplace the ship vertically

if you can place the ship horizontally increment nShips, set backtrack = false and return recursivePlace

increment point accordingly, set backtrack equal to false and return recursivePlace

if you can unplace the ship horizontally

increment point accordingly, set backtrack equal to false and return recursivePlace

bool MediocrePlayer::recursivePlace(Board& b, int nShips, Point point, bool backtrack, int& depth)

This is the same as the function above but there is a limit to the depth of recursion to prevent stack overflow. I used this function in my placeShips function.

-----------------------------------------------------------------------------------------

bool MediocrePlayer::placeShips(Board& b)

block the board, set bool backtrack = false, int i = 0.

While ((recursiveplace == false) && (i < 50))

Clear the board

Block the board

i++

unblock the board

if i >= 50

return false

else

return true

------------------------------------------------------------------------------------------------------------------------------------------

Point MediocrePlayer::recommendAttack()

If in state 1

if no points were visited before generate a random point within the board and return it

else generate a random point, check if the random point was visited before, if it was never visited return it, else generate a new random point and repeat this process

If in state 2

If the doneCross() == true

Delete all elements of m\_cross(delete the linked list which stores the set of points that were attacked and were on the cross)

Now guess a new point as in state 1 above, and return that point

While you haven’t iterated through this while loop a 100 times and have not succeeded

randomly guess a point on the cross.

If the point was attacked before add it to the set of points that have been attacked and are on the cross

Else add the point to the set of points that have been attacked and are on the cross and add it to the list of all points that have been attacked. Return that point.

Delete the set of points that have been attacked and are on the cross. Guess a random point that has not been attacked before. Return that point.

------------------------------------------------------------------------------------------------------------------------------------------

void MediocrePlayer::recordAttackResult(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId)

store p as the point that was lastattacked.

If m\_state == 1

If shot is valid, shot results in a hit, ship is not destroyed

Set m\_state = 2. Store p as the center of the cross. Calculate the number of point on the left (l\_size), right (r\_size), down(d\_size), up (u\_size) side of the cross respectively and store these values separately.

If m\_state == 2

Else if shot is valid, shot results in a hit, and ship is destroyed

Set m\_state = 1.

------------------------------------------------------------------------------------------------------------------------------------------

bool MediocrePlayer::doneCross()

iterate through all the points that have been attacked and are on the cross

if the number of all those points equals or is bigger than(l\_size + r\_size + u\_size + d\_size) return true

else return false

------------------------------------------------------------------------------------------------------------------------------------------

bool GoodPlayer::CheckCross(Point shipbeingplaced, Node<OnCrossData>\* ThierShip)

Note: I will refer to TheirShip as a “ship”. In the code this is not a ship but a pointer to a OnCrossData, so the necessary operations need to be done to get stuff like the length of the ship , its direction, etc…

if TheirShip != nullptr

depending on the orientation of the ship check if the point beingplaced comes into contact with a cross centered at a certain point on the ship, if so return true (Repeat this for all points on the ship) else return false

------------------------------------------------------------------------------------------------------------------------------------------

bool GoodPlayer::OnCross(Board& b, int shipLength, Point topOrleft, Direction Dir)

Check if the ship you are tring to place comes into contact with any crosses that might occur in game with a hit on all the ships that have been placed so far. If so return true. Else return false.

------------------------------------------------------------------------------------------------------------------------------------------

bool GoodPlayer::placeShips(Board& b)

{

if (game().nShips() >= 3)

{

make a random point

for (int i = 0; i < 3; i++)

int count = 0;

while you couldn’t place the ship and have not iterated this loop more than 49 times

{

if (randInt(2) == 0)

if (!OnCross(b, game().shipLength(i), point, VERTICAL))

if (b.placeShip(point, i, VERTICAL))

OnCrossData data(point, game().shipLength(i), VERTICAL);

store the ship in a linked list (m\_head)

else if (!OnCross(b, game().shipLength(i), point, HORIZONTAL))

if (b.placeShip(point, i, HORIZONTAL))

OnCrossData data(point, game().shipLength(i), HORIZONTAL);

store the ship in a linked list (m\_head)

else

if (!OnCross(b, game().shipLength(i), point, HORIZONTAL))

if (b.placeShip(point, i, HORIZONTAL))

OnCrossData data(point, game().shipLength(i), HORIZONTAL);

placed = true;

store the ship in a linked list (m\_head)

else if (!OnCross(b, game().shipLength(i), point, VERTICAL))

if (b.placeShip(point, i, VERTICAL))

OnCrossData data(point, game().shipLength(i), VERTICAL);

placed = true;

store the ship in a linked list (m\_head)

Point temp = game().randomPoint();

point.r = temp.r;

point.c = temp.c;

count++;

if (count >= 50)

if (recursivePlace(b, random point, i, 0) == false)

return(false);

}

Block the board

while ((recursivePlace(b, 3, POint, false) == false) && you haven’t gone through this loop more than 49 times

{

b.clear();

b.block();

return false if this is the 49th iteration

}

Unblock the board and return true

}

If the game does not have at least 3 ships

{

Use the same algorithm as the MediocrePlayer PlaceShips function

}

Otherwise return true

}

------------------------------------------------------------------------------------------------------------------------------------------

Point GoodPlayer::recommendAttack()

{

return(TransitionMatrix.MostProbableLocation(game()));

}

------------------------------------------------------------------------------------------------------------------------------------------

void GoodPlayer::recordAttackResult(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId)

{

TransitionMatrix.UpdateDist(p, validShot, shotHit, shipDestroyed, shipId, game());

}

Point TotalDistribution::MostProbableLocation(const Game& g) // Recommend Attack

{

If you’ve hit any ships you didn’t destroy && you have destroyed any ship

{

Until you succeed and you haven’t executed the following more than 50 times

If m\_hits is not empty

Pick a random point from a 4 by 4 cross centered at m\_hits[0] as long as it was not attacked before and is valid. Add this point to m\_attacked

If you’ve exectude the above 49 times or more

If there are hits which haven’t resulted in a sunk ship remove the first one in m\_hits

Else set m\_justDestroyedShipButThereAreHitSquares = false and return a random point

}

else

{

Set m\_justDestroyedShipButThereAreHitSquares = false. Find the square which has the maximum probability of having a ship. If the maximum probability is equal to 0, set m\_justDestroyedShipButThereAreHitSquares = true. Return that the point at which the probability of finding a ship is maximum.

}

m\_attacked.push\_back(Max);

return(Max);

}

}

Return a random point on the board

}

----------------------------------------------------------------------------------------- void TotalDistribution::UpdateDist(Point p, bool validShot, bool shotHit, bool shipDestroyed, int shipId, const Game& g) //RecordAttackResult

{

if (validShot == true)

{

if (shotHit == true)

{

Add p to m\_hits

If you haven’t found the cross before this

Set m\_CrossCenter to p and set m\_FoundCross = true;

if (shipDestroyed == true)

{

Delete the shipId of the ship that was destroyed from the m\_shipId vector. Call adjustRegisteredHits(g.shipLength(shipId)) and decrement m\_nShips

if m\_hits.size() != 0 Set m\_justDestroyedShipButThereAreHitSquares = true;

clear all existing distributions in DistGame and reinitialize all of them, accounting for all the squares that have been attacked thus far. Set m\_FoundCross = false

}

else

{

Call reEvaluateDist(true, p, m\_CrossCenter) for all distributions

}

}

If you missed the shot call reEvaluateDist(false, p, m\_CrossCenter) for all existing ships and their distributions

Set the m\_probability array values all equal to zeroa and call storeProbInf(g, DistGame[i]) for every distribution.

}

}

-----------------------------------------------------------------------------------------

void TotalDistribution::storeProbInf(const Game& g, Distribution dist)

{

For each position in the m\_probability array of each distribution, do

m\_probability[r][c] = m\_probability[r][c] + dist.getPosOrientation(r,c).getSize();

}

-----------------------------------------------------------------------------------------

void TotalDistribution::adjustRegisteredHits(int shipLength)

{

if (the size of m\_hits != 0)

{

If the size of m\_hits == shiplength then erase every element in m\_hits such that it is the empty vector

Else create std::vector<Point> erase;

From this point on you are looking at the se tof points which have resulted in a hit. A ship has been destroyed so you need to find the set of points that correspond to the ship that was destroyed. This is an ambiguious process in some instances. In those instances do nothing. Otherwise (i.e. if there is only one possible location that the ship can be in (maybe because not all hits are next to each other and the ones that are are equal to the size of the ship that was destroyed)). Make use of the orientations class and its member functions to determine which squares the ship that was sunk actually occupies. If the result is unambiguous remove those points from the m\_hits array