report Project 3

Xiwei Ma 704755732

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

For GameImpl(), there is a vector of struct “Ship” as private member. The position of each ship in vector is its shipId, and the struct contains information of the ship’s symbol, name, length. Every ship added will be push\_back to the vector. So we can return such information using only shipId.

For BoardImpl(), there is a 2-D array to represent each board object. The size of array is the MAXROWS and MAXCOLS, but actually will only display the part depends on m\_game. Every unit [row][col] of the array is a char which represents the playing field(such as ‘.’, ‘o’, ‘X’) and changing the playing field(such as a shot made) will cause a specific element in array represent that position being assigned with a different value.

Also, there is a vector of struct “Ship” (represent every ship on the board). The struct Ship will save information of every ship, such as symbol, length remained on board. If the length of one ship is zero, meaning that the ship is destroyed. If all “Ship”s in the vector don’t have a length, then the ships of the board allDestroyed.

For MediocrePlayer(), there are two vectors of Point. The first one contains all points already been attacked. So the player won’t make a wasted shot. The other contains the set of points around a transition shotHit from state one to state two. So the next point to attack can be randomly chosen from this set.

There is also a Point data member keep tracking of the transition point from state 1 to state 2, and an integer indicate state1 or 2.

For GoodPlayer(), besides all the similar member of MediocrePlayer(), there are a vector of default points and a subset vector of the Set vector. The default points are substitution of randomly selected points in state one. The points are in a sequence that are more efficient than random selected in finding ships. The subset is used to determine the direction and thus narrow the range of set.

Also, there is a Point lastCellAttacked record the last cell attacked to help determine the direction of further attack.

**b. 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.**

To win the standard battleship, I created a function to place ships on specific appropriate points. At the beginning of GoodPlayer’s placeShips() function, the stdPlaceShips() will be called. To fit other kinds of settings of the game, the stdPlaceShips may not work and return a false. If a false is returned, the board will be clear()ed and then randomly add required ships.

For recommending moves, there will be three states. For state 1, the function will return a Point following the sequence of a vector. The vector of Points is constructed in the constructor of GoodPlayer. Instead of picking random points, the Point to attack will be from up to down, from right to left of the board.

When a ship is hit but not destroyed, the state will be switched to state 2. At the same time, four points from four directions(left, right, up, down to the transition point) will be pushed back to a vector. During state 2, the function will randomly return a point from this vector.

When a ship is hit and destroyed, state 2 will clear its vector and back to state 1.

However, if a ship is hit but not destroyed during state 2, the shot Point will be recorded and the vector will be cleared and switch to state 3. From point recorded and previously recorded transition point, we can infer the direction of the ship(Horizontal or Vertical). And then push back eight points to another vector from transition point in that direction (4 points each sides). And state 3 will attack points following the sequence of Points from vector.

If a ship is destroyed in state 3, it will clear the vector up and go back to state 1.

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

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

{

player 1 places ships;

or return nullptr if fails;

player 2 places ships;

or return nullptr if fails;

while(no one wins)

{

print instruction lines;

display player2’s board(depends on is human or not);

get point to attack;

attack the point;

record this attack;

if(the shot is not valid)

print wasted a shot;

print and display the result of the shot;

if first player destroyed all ships of second player

{

first player win

print and return result;

}

if(pause)

wait for enter;

reset boolean result to false;

display first player’s board;

player two attack;

record result;

if(the shot is not valid)

print wasted a shot;

print and display the result of the shot;

if second player destroyed all ships of first player

{

second player win ;

print and return result;

}

if(pause)

wait for enter;

}

return null;

}

**BoardImpl::BoardImpl(const Game& g) : m\_game(g)**

{

for every grid in 2-D array

set char to ‘ .’;

}

**void BoardImpl::block()**

{

for every grid in 2-D array

50% randomly set it to ‘#’;(blocked)

}

**void BoardImpl::unblock()**

{

for every grid in 2-D array{

if it is blocked

reset to ‘.’;}

}

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

**{**

if(one of the parameters are not valid)

return false;

for every grid in 2-D array board

{

if(same symbol previously been placed on this Board)

return false;

}

if(direction is HORIZONTAL)

{

if(the ship would be partly or fully outside the board)

return false;

for(every point the ship will be placed)

if(the grid != '.')//The ship would overlap an already-placed ship./blocked space

return false;

}

create a ship;

set the length , symbol, id of the ship;

push\_back the ship to vector;

while(i in the length of the ship)

{

the char grid assigned with the symbol of ship;

i++;

}

}

else if(direction is VERTICAL)

{

while(i in the length of the ship)

{

if(the ship would be partly or fully outside the board)

return false;

if(the point is not ‘.’)

return false;

i++;

}

create a ship;

set the length , symbol, id of the ship;

push\_back the ship to vector;

while(i in the length of the ship)

{

the char grid assigned with the symbol of ship;

i++;

}

return true;

}

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

**{**

if(one of the parameters are not valid)

return false;

if(direction is HORIZONTAL)

{

if(out of board)

return false;

for(given range of points)

{

if(one is not the given symbol)

return false;

}

for(every ship in vector)

{

if(the ship is this one)

destruct the ship in the vector;

}

for(every points in the range)

set the char in the board to '.';

}

if(direction VERTICAL)

{

if(out of board)

return false;

for(given range of points)

{

if(one is not the given symbol)

return false;

}

for(every ship in vector)

{

if(the ship is this one)

destruct the ship in the vector;

}

for(every points in the range)

set the char in the board to '.';

}

return true;

}

**void BoardImpl::display(bool shotsOnly) const**

{

print the space and column numbers new line

for(every row)

{

print row number;

for(every column)

{

if(print all)

print each grid on the board;

else

{

if(the grid is symbol of ships)

print ‘.' instead;

else

print char at that position;

}

}

print newline;

}

}

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

{

if(out of boundary)

return false;

record the original char of the attacked position;

if(the position is not valid to attack)

return false;

if(hit a ship)

{

for(every ship in the vector)

{

if(find the ship being hit)

{

decrease its length on board by one;

record the place of ship in vector;

}

}

change the grid on board to ‘X’ present being hit;

change shotHit to true;

if(the ship has no length left on board)

{

change shipDestroyed to true;

record shipId;

}

}

else

{

set the grid to 'o';

set shotHit to false;

set shipDestroyed to false;

}

return true;

}

**bool BoardImpl::allShipsDestroyed() const**

{

for(every ship in vector)

{

if(length remain more than 0)

return false;

}

return true;

}

**char HumanPlayer::getHorV(int shipId)//private function help to get direction**

{

print instruction for enter direction;

get user input;

if(the first char is indicating a direction)

return the char;

else

{

print error message;

return this function again;

}

}

**bool HumanPlayer::placeShips(Board& b)**

{

print instruction message and display board;

for(every ship to be placed)

{

get direction for this ship using the private function;

change the char to Direction;

print instruction;

get the point form input;

try to place to ship;

while(point is not valid)

{

if(input is not valid)

print error message;

else

print another error message;

get new point;

try to place ship on new point;

}

if(not the last one to be placed)

display the updated board;

}

return true;

}

**Point HumanPlayer::recommendAttack()**

{

Print instruction;

get point from input;

while(input is not valid)

{

print error message

get new point;

}

return valid new point;

}

**bool MediocrePlayer::legalMove(int& shipId , Board& b)//recursive**

{

if(all ship placed)

return true;

for(every point of the board)

{ try place ships HORIZONTAL on this point;

if(ship placed)

{

update to next ship;

call this function again with updated shipId;

if(rest of the ships placed)

return true;

else

unplaced this ship;

}

try place ships VERTICAL on this point;

if(ship placed)

{

update to next ship;

call this function again with updated shipId;

if(rest of the ships placed)

return true;

else

unplaced this ship;

}

}

no point to place return false;

}

**bool MediocrePlayer::placeShips(Board& b)**

{

while(no more than 50 times)

{

block the board;

call recursive function start from first ship;

if(all ship placed)

{

unblock;

return true;

}

unblock;

times tried + 1;

}

return false;

}

**bool MediocrePlayer::NotYetAttacked(Point p)//private member function**

{

if(no points has been attacked)

return true;

vector<Point>::iterator pos = Points.begin();

while(iterate through the vector)

{

if(the Point exists in the vector)

return false;

else

continue iteration;

}

return true;

}

**Point MediocrePlayer::recommendAttack()**

{

if (in state 2)

{

random a point p in the vector set;

erase the point in the vector set;

if(the set becomes empty)

switch state to 1;

return p;

}

else

{

random a point;

while(the point already been attacked)

{

random another point;

}

push the valid point into vector points;

return the point;

}

}

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

{

if (in state 1, shot hit but not destroyed)

{

record the transition point;

for(up and down for 4 points)

{

if(the point is valid)

push\_back the point to vector set;

}

for(left and right 4 points)

{

if(the point is valid)

push\_back the point to vector set;

}

while(iterator through the vector)

{

if(a point is already attacked)

erase the point;

}

switch state to 2;

}

if(in state 2, shot hit and destroyed)

{

clear the set and switch to state 1;

}

}

**GoodPlayer::GoodPlayer( std::string nm, const Game& g ): Player(nm,g),lastCellAttacked(0,0),m\_transition(0,0), m\_default(0)**

{

m\_state = 1;

for(go through rows, every two rows)

{

for(go through cols)

{

if(every two cols)

push back certain points to vector defaultP;

}

}

for(go through remaining rows)

{

for(go through cols)

{

if(remainig cols)

push back certain points to vector defaultP;

}

}

}

**bool GoodPlayer::stdPlaceShips(Board& b)//private member function**

**{**

for(go through ship id and when result stay true)

{

if(first ship)

{

set a specific point for this ship;

set a specific direction for this ship;

}

else if(second ship)

{

set a specific point for this ship;

set a specific direction for this ship; }

else if(third ship)

{

set a specific point for this ship;

set a specific direction for this ship; }

else if(fourth ship)

{

set a specific point for this ship;

set a specific direction for this ship; }

else if(fifth ship)

{

set a specific point for this ship;

set a specific direction for this ship; }

else cannot use this method

return false;

try to place this ship;

return false if cannot be placed;

}

return result;

}

**bool GoodPlayer::placeShips(Board& b)**

{

try to place ships using the above function;

if succeed

return true;

otherwise clear the board.

for(each ship to be placed)

{

while(not yet been placed successfully)

{

random a number represent direction;

random a point on board;

try to place the ship on the randomly got point and direction

}

}

return true;

}

**Point GoodPlayer::recommendAttack()**

{

if(in state 2)

{

if(the vector of Points to choose is empty)

set state to 1;

else

{

get a random position of the vector;

get the point in that position;

push back the point into vector of points already attaked;

erase the point from the vector for state 2;

return the point;

}

}

if(in state 3)

{

if(the vector for state 3 to choose is empty)

set state to 1;

else

{

get and record the first Point in the vector;

erase the first point;

push back the point into the vector of attacked points;

return the point;

}

}

if(in state 1)

{

while(the vector store points to attack is not finished)

{

if(the point in current position of the vector is attacked already)

{

move position to next;

}

else

{

push back the point into vector of already attacked points;

return the point;

}

}

no preset Points left, random a point;

while(the point already been attacked)

{

random another point;

}

push back this point into vector of already attacked points;

return this point;

}

return point;

}

**void GoodPlayer::setTheSet(Point p)//private function to help construct vector for state 3**

{

if(in the same row)

{

for(1 to 4)

{

a point 1 to 4 distance left to transition point;

if(the point is valid)

push back the point to the vector;

a point 1 to 4 distance right to transition point;

if(the point is valid)

push back the point to the vector;

}

}

else if(in the same column)

{

for(1 to 4)

{

a point 1 to 4 distance upward to transition point;

if(the point is valid)

push back the point to the vector;

a point 1 to 4 distance downward to transition point;

if(the point is valid)

push back the point to the vector;

}

}

while(iterate through the vector)

{

if(the point already been attacked)

erase the point from vector;

else

move to next position;

}

}

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

{

if(in state 1 and hit something without destroy)

{

record this transition point;

Point left to this point;

if(the point is valid)

push back the point to vector of state 2;

Point right to this point;

if(the point is valid)

push back the point to vector of state 2;

Point above this point;

if(the point is valid)

push back the point to vector of state 2;

Point below this point;

if(the point is valid)

push back the point to vector of state 2;

while(iterate through this vector)

{

if(already attacked this point)

erase this point;

else

move to next;

}

change state to 2;

}

else if(in state 2, hit something and destroyed)

{

clear the vector for 2;

change state to 1;

}

else if(in state 2, hit something but not destroyed)

{

record this point;

clear vector for state 2;

call function setTheSet(), use this point;

change state to 3;

}

else if(in state 3, hit something and destroyed)

{

clear the vector for state 3;

change state back to 1;

}

}