A. A description of the design of your data structures. For example, how do

you represent the board? How do you record shots made?

Board.cpp: I use a two-dimensional array to represent the board. Initially, the board is empty (filled with ‘ ’) . Using clear function can make the board filled with ‘.’ . The blocked places are represented by ‘\_’ . We use ‘X’ to represent ship wreck and ‘o’ to represent attacked places that we miss.

Game.cpp: We have 3 integers to represent the number of total rows, number of total columns in the board, and the total number of types of ship in this game respectively. I create a date type called Ship to represent different types of ship and their information. And I use a vector to store the kinds of ships and information related to ships.

HumanPlayer.cpp: I do not have any private data member except for those provided by the skeleton.

MediocrePlayer.cpp: I use an integer to record the times that we have been attacking in state2. I use a bool to record the state: true---state1, false---state2. I use a two-dimensional array called mindBoard to record the result of attack as will be seen by the human player. I create a Point type which is just the same as that in the Game. I have a Point variable to store the first position attacked that cause us to be in state2.

GoodPlayer.cpp: I create a ship class to represent the types of ship that the opponent still has. And I use a vector of ships to store those ships that have not been destroyed. I have an integer to record the time that we have been in state2. I use a bool to represent the state we are in. I use a Point to record the position that, after attack, put us into state2, and I use another Point to record the latest position according to which we attack we are in state2. We also use a two-dimensional array called mindBoard to record the result of attack as will be seen by the human player.

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.

Placing ships: the same strategy as used by MediocrePlayer.

Recommending moves:

In state1, I randomly choose whether to attack even coordinates points or odds coordinates points so that we won’t waste too much chances at nearby position of previous attacks. After we have tried most of the even and odd coordinates, we randomly try other position. At the same time, we check each potential attack position we get to see whether it is possible to have a ship at that position by record the space near it. According to the above strategy, we find a position with the best chance to have a ship segment.

In state2, we first see the possible directions (horizontal or vertical) that the attacked ship could be. Then according to its direction, if it is horizontally placed, we attack to the right or to the left 1 position away, then record the result. If we previously attacked a position that’s not the position that bring us into state2, we keep attacking along the direction of that position, if we missed, we turn to the opposite direction. The same strategy works with vertically placed ship. If we cannot decide whether it is vertically placed or horizontally placed, but we have previously find its direction, we keep going on that direction, if we missed, we turn to the opposite. But if we do not have a direction yet, we randomly try one of the four possible direction that’s not been attacked before, then record the result. At the end, if by any accident, we do not have a valid point yet, we just randomly return a position that’s with in 4 points away from the position that brings us into state2.

C. Pseudocode for non-trivial algorithms. For example, how does a

MediocrePlayer place ships? How does your GoodPlayer implement its

strategies?

**HumanPlayer**

placeShips

*check whether the board if big enough to fit all ships*

*for each ship in the game*

*display the board for the user*

*keep prompting the user to enter a direction for the ship if the direction given by the user is not valid (not h or v)*

*after you get a direction*

*ask for the leftmost cell*

*try place the ship*

*if succeed, continue on to another ship*

*else, ask for point and try place ship again until you get the valid result*

*after you finish place all five ships, return true*

recommendAttack:

*ask the user to input a point, then return it*

**MediocrePlayer**

For placing ships (same for Mediocre and Good Player):

*Use an integer to record the number of ships placed*

*Using an integer to count the times we have re-blocked the board*

*Call the recursive function that we use to place ships*

*If we success, unblock the points, return true*

*If we fail, return false*

Recursive function to place ship----tryPlaceShips:

*If we have tried 50 times, return false*

*If there is no ship placed on the board, re-block the board and count one more time*

*If the number of ships we have placed on the board is the same as the total number of ship, we return true*

*For every row on the board*

*For every column on the board*

*Try to place the ship at each point vertically*

*If successful, record the number of ships on the board*

*Call the function again*

*If the function succeeds, return true*

*If we can place it vertically, Try again horizontally*

*If successful, record the number of ships on the board*

*Call the function again*

*If the function succeeds, return true*

// *If we come to here, that means there is no place on the board that can fit the current ship!*

*If there is no ship on the board (we can’t even place the first ship)*

*Call the function without changing any parameter*

*Else*

*remove the latest ship that we placed, record the number of ships on the board, then return false*

RecommentAttack

*If in state 1*

*Get a random point using our function----randomReturnP (for M)*

*set corresponding position to ‘o’*

*return the position*

*if in state 2*

*find the limits on the board in all four direction*

*keep track of the time we are in state 2*

*if we are in state 2 for a time that’s more than the possible points around*

*set track time to 0*

*set state to state1*

*record and return a position using randomReturnP (for M)*

*else*

*record and return a random point we get from our function----constrainedRandom*

randomReturnP (for M)

*get a random point within the board*

*if the point is not attacked before*

*return it*

*else*

*get another one and check again, until we get a valid point*

*return the valid point*

constrainedRandom

*randomly decide the direction of this attacking (horizontal or vertical)*

*get a random point within the direction and limits*

*if the point is not attacked before*

*return it*

*else*

*get another random one and check again, until we get a valid point*

*return the valid point*

recordAttackResult

*if in state1*

*if shot is not valid*

*return*

*if shot missed*

*return*

*if destroy a ship*

*record the result on the board*

*return*

*if hit a ship but have not destroyed it*

*record the result on the board*

*record the position*

*go to state2*

*if in state2*

*if shot is not valid*

*return*

*if shot missed*

*return*

*if hit a ship but have not destroyed it*

*record the result on the board*

*return*

*if destroy a ship*

*record the result on the board*

*set the time in state2 to 0*

*go back to state1*

**GoodPlayer**

PlaceShips: the same as MediocrePlayer

recommendAttack:

*if state1*

*get a point using function----randomeReturn (for GoodPlayer)*

*return the point*

*if state2*

*find the limits on the board in all four direction*

*keep track of the time we are in state 2*

*if we are in state 2 for a time that’s more than the possible points around*

*set track time to 0*

*set state to state1*

*record and return a position using randomReturnP (for G)*

*else*

*check the possible direction of the ship*

*record and return a random point we get from our function----advancedTargeting*

randomReturnP

*randomly decide whether choose an even coordinate or an odd coordinate*

*randomly get a point*

*record how many time we get such kind of point*

*if the point is attacked before or it is impossible to be a ship segment*

*get the point again and record the time*

*if we hit 80 times, break out of the loop*

*if the result point has not been attacked before, return it*

// if no point is returned above, we get here

*get a random point within the board*

*if the point is attacked before or it is impossible to be a ship segment*

*get the point again until we get a valid point*

*return the point*

advancedTargeting:

*if the direction is horizontal*

*if the latest hit point is to the right of the point that bring us into state2*

*get a point to the right of the latest hit point*

*if the point has not been attacked before*

*return it*

*if the latest hit point is to the left of the point that bring us into state2*

*get a point to the left of the latest hit point*

*if the point has not been attacked before*

*return it*

*else* //this mean we have not decide whether to right or to left

*get a random point either to right or to left*

*if the point has not been attacked before*

*return it*

*else*

*get a point that is at the opposite direction*

*if the point has not been attacked before*

*return it*

*if the direction is vertical*

*if the latest hit point is to the bottom of the point that bring us into state2*

*get a point to the bottom of the latest hit point*

*if the point has not been attacked before*

*return it*

*if the latest hit point is to the top of the point that bring us into state2*

*get a point to the top of the latest hit point*

*if the point has not been attacked before*

*return it*

*else* //this mean we have not decide whether to bottom or to top

*get a random point either to bottom or to top*

*if the point has not been attacked before*

*return it*

*else*

*get a point that is at the opposite direction*

*if the point has not been attacked before*

*return it*

*else* // this means we have not decide the initial direction

*if the latest hit point is to the right of the point that bring us into state2*

*get a point further down that direction*

*if the point has not been attacked before*

*return it*

*if the latest hit point is to the left of the point that bring us into state2*

*get a point further down that direction*

*if the point has not been attacked before*

*return it*

*if the latest hit point is to the bottom of the point that bring us into state2*

*get a point further down that direction*

*if the point has not been attacked before*

*return it*

*if the latest hit point is to the top of the point that bring us into state2*

*get a point further down that direction*

*if the point has not been attacked before*

*return it*

*else* //we just got in state2 and have not attacked

randomly choose a direction

get a point in that direction within 1 position away from the point that -bring us into state2

if that point has not been attacked before

return it

else

try another direction

if that point has not been attacked before

return it

else

try another direction

if that point has not been attacked before

return it

else

try the last direction

if that point has not been attacked before

return it

//if by any accident I do not know, I get here

*randomly get a point using the strategy of attack of MediocrePlayer in state2*

*return it*

recordAttackResult:

*if state1*

*if shot not valid*

*return*

*if shot missed*

*record the result on the board*

*return*

*if destroy a ship*

*record the result on the board*

*erase the corresponding ship storage from the vector of existing ships*

*return*

*if hit a ship but have not destroy it*

*record the result on the board*

*record this position*

*set the latest hit point to it*

*go to state2*

*if state2*

*if shot invalid*

*return*

*if shot missed*

*record the result on the board*

*set the latest hit point back to the point that bring us into state2*

*return*

*if hit a ship but have not destroyed it*

*record the result on the board*

*set it the latest hit point*

*return*

*if destroy a ship*

*erase the ship from the vector of existing ships*

*set the latest hit point to invalid*

*set the recorded position to invalid*

*set time in state2 to 0*

*go to state1*

**Game**:

addShip

*create a ship with length, symbol and name*

*push that ship into the vector of ships*

*add one to the total number of ship*

*return true;*

play

*clear the two board for two players*

*call placeship function for two players, if one of them cannot place all ship, return a nullptr*

*set a Player pointer to the result of playGame function*

*output the result*

*return the pointer to the winner*

playGame

*display the opponent’s board for the player*

*get a point from the player to attack*

*call the attack function and record the result for player*

*if attack missed*

*display the result for the player*

*if attack hit something*

*if the ship is destroyed*

*display the result for the player*

*if all ships are destroyed*

*this play wins*

*else*

*display the result for the player*

*else*

*display the result for the player*

*return the result of calling this function with the player switch their condition*

**Board**

Block

*For each point within the board*

*Using a random integer to decide whether to block it (set it to ’\_’) or not*

Unblock

*For each point within the board*

*If a position is ‘\_’*

*Set it back to ‘.’*

placeShip

get the information of the ship that needs to be placed

check if the information is valid

check if the symbol of the ship has been used before on the board

check whether the initial point is valid

check if the direction is valid

if the direction is valid

corresponding to the direction

check if the ship would be out of boundary

check if the position that we are going to use is blocked or occupied

if all check passed

place the ship at the corresponding position

return true

else

return false

unplaceShip

*check if the information of the ship is valid, if not, return false*

*check if the initial position of the ship is valid, if not, return false*

*if the direction is valid*

*corresponding to the direction*

*check if the entire ship is contained on the board*

*if not, return false*

*else, return false*

*remove the symbol of the ship from the board according to the direction and the initial point*

*return true;*

attack

*check the validity of attack, if not, return false*

*check if the attacking position has been attacked before, if yes, return false*

*make the attack, set the attacked position to ‘X’*

*record the result of the attack (if hit or missed)*

*if we hit something*

*check the row and column of the attacked point in the board to see if the ship is still on the board, if not, that means it is destroyed*

*if the ship is destroyed*

*record the result and return true*

*else*

*record the result and return true*

*if missed*

*record the result and set the board*

*return true*

allShipsDestroyed

*for every kind of ships*

*for each point on the board*

*if no symbol of the ship exist*

*return false*

*return true*