Konner Macias

004603916

Project 3 – Report

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

As for my board, I created a 2D character array set with MAXROWS and MAXCOLS. This approach allowed for easy access of information at each grid point on the board and easy alterations at these points.

At each point on the grid, it was either marked by ‘.’ Indicating water, ‘X’ indicating that a ship at that location was hit there, ‘o’ indicating that it was a missed shot, and ‘#’ to represent a blocked spot. I declared const characters representing these values.

To keep track of how what unique ships were on board, I create a vector of ship IDs. This served very useful to check if I was accidently placed a ship that was already on the board.

In Game.cpp, I declared a Ship struct within the private section of GameImpl, then I was able to create a vector of Ship pointers called myShips. This was extremely useful in easily adding ships to the game, or deleting them. A struct makes all the data members public, so it was easy for me to access a ship’s name for example.

In my MediocrePlayer class, I created another character grid set with MAXROWS and MAXCOLS again. I did this because I wanted to keep a private board just for me, the developer, to have access to. With this hidden grid, I wanted to be able to mark where the player has taken their shots. Since it’s a character 2D array, I was able to mark locations that were shot with ‘#’. This made is very easy to check if I had already shot at that location.

In my GoodPlayer class, I created the same grid as in Mediocre. With each player, I added the data member ‘isSearch’ allow myself to understand which state they were in for attack.

1. 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:

I decided to use the same strategy of placing ships as I used for my MediocrePlayer. I did this not because I’m lazy, though I am a little bit, but because I believe the blocking of cells at random until we find a board where we can place our ships down is optimal. It’s optimal because the ships are randomly scattered so that even me, the human player, cannot figure out its pattern. Here is the is the pseudocode for my placeShips function with a recursive place ships function as a helper.

*GoodPlayer::placeShips(board &b){*

*Declare integer variables k, trial and count. Set k and trial = 0.*

*While (trial < 50){*

*Block the board*

*Count = 0;*

*Increment trail by 1*

*If (recursivePlaceShips(b,k,count)) // if my recursive function returns true*

*Unblock the board and break out of loop*

*Otherwise, just unblock board and start loop over*

*}*

*If (count == number of ships in game)*

*Return true;*

*O/w, return false;*

*}*

*GoodPlayer::recursivePlaceShips(b,k,count){*

*If (count == number of ships in game) return true;*

*For each grid point:*

*Get random direction and assign to dir.*

*If dir is Horizontal {*

*If (b.placeShip(r,c), k, HORIZONTAL) { // if we can place the ship down*

*Increment count;*

*If (!recursivePlaceShips(b, k+1, count) // recursively checking*

*Decrease count and unplace ship from board.*

*O/w, return true;*

*} try same thing except now with VERTICAL*

*} Now basically repeat if dir was VERTICAL*

*If nothing returns true after all this checking, then return false.*

*}*

The recursivePlaceShips function was extremely useful because depending on the random direction to try to orient the ship first, we can then try checking if the ship can fit on the board in the other direction thus increasing our chances of placing them ship down.

Recommending Moves:

I will go through first in words how I recommended words then use pseudocode for the more complicated processes.

First, I checked whether I was in ‘Searching Mode’ (a.k.a. state1). If so, in order to increase my chances of hitting a ship at random, I wanted to attack spots similar to the pattern of colored squares on a checker board. This spacing truly allowed myself to take advantage of the length of the board and the fact that they cannot be placed any other way except horizontal or vertical. So, I then went through every grid cell starting at the top left and checked if I was on the ‘colored square’ if you will, and whether that grid spot had already been a previous move. If these conditions are satisfied, I returned this point! Let’s say it goes all the way through the board to the bottom, then I added another double loop now checking the other spaces I did not initially attack.

If we are not in ‘Searching Mode’, then we are targeting! I then decided to create a queue of points! A queue is effective for breadth-first searching and finding the shortest path. Searching from close to the spot out really helped the goodPlayer’s win rate. From myRecordAttackResult function, I was able to indicate the cell where we initially hit, let’s call this the source cell. For the rest of this explanation I must use pseudocode.

*Queue myPoints // queue of points that I am focusing on*

*Push Source Cell*

*While (!myPoints. Empty()){*

*Store front Point as current then pop item from queue.*

*If (on my grid, this location is blocked){ // this is a good thing, it should be marked in the beginning*

*Now for each direction, we must check if it first has an issue with the border: // we will be moving in index so don’t want to go over*

*Then if this new location incremented 1 in that direction is free, then mark on grid and return this point.*

*If not, push the Point to the back of the queue.*

*}*

*}*

This is method of when in state1, randomly choosing points to attack in checkerboard was very effective compared to the random attack by the mediocre player. Also, by using the queue, I worked from cells close to the ship out rather than the mediocre player attacking at random within 4 spots in each direction. These two distinct differences made the difference in earning the good player an >80% win percentage.

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

MediocrePlayer::placeShips(Board &b) is the same as GoodPlayer’s as discussed above!

*MediocrePlayer::placeShips(board &b){*

*Declare integer variables k, trial and count. Set k and trial = 0.*

*While (trial < 50){*

*Block the board*

*Count = 0;*

*Trial ++;*

*If (recursivePlaceShips(b,k,count))*

*Unblock the board and break out of loop*

*Otherwise, just unblock board and start loop over*

*}*

*If (count == number of ships in game)*

*Return true;*

*O/w, return false;*

*}*

*MediocrePlayer::recursivePlaceShips(b,k,count){*

*If (count == number of ships in game) return true;*

*For each grid point:*

*Get random direction and assign to dir.*

*If dir is Horizontal {*

*If (b.placeShip(r,c), k, HORIZONTAL) { // if we can place the ship down*

*Increment count;*

*If (!recursivePlaceShips(b, k+1, count) // recursively checking*

*Decrease count and unplace ship from board.*

*O/w, return true;*

*} try same thing except now with VERTICAL*

*} Now basically repeat if dir was VERTICAL*

*If nothing returns true after all this checking, then return false.*

*}*

*MediocrePlayer::recommendAttack(){*

*If (state1){*

*Generate random point that has not been marked yet on grid.*

*Mark the point and return it.*

*}else{*

*Declared bool values ‘bordered’ set to true and ‘worked’*

*While(bordered){*

*Generate random number between 0 and 3 then assign a point ‘pnt’ to the sourceCell.*

*Based off random number, use switch statement for cases.*

*In each case using switch statement:*

*Worked is true*

*While we keep getting a blocked grid point:*

*Check if there is an available space between point and border in that direction.*

*If not, worked is false so we break out of loop and out of case.*

*O/w, mark bordered as false*

*Then while adjusting for whether close to border, mark our point and return it.*

*}*

*}*

*}*

*MediocrePlayer::recordAttackResult (parameters){*

*If (in state1 and the shot was hit)*

*Switch to state2*

*Assign source cell to this point*

*If the ship was destroyed however, make sure state1 is true*

*Else*

*If (shipDestroyed)*

*State1 is true.*

*}*

*BoardImpl::placeShip(Parameters){*

*Check if shipId violates its bounds*

*Use loop to check if shipId already exists in ShipIDs vector.*

*If so, return false.*

*If (dir is Horizontal){*

*From k = 0 while k < the ship length of the ID incremented by 1:*

*Return false if (ship will fall off board*

*Return false if ship overlaps something not water (use k to look at spaces ship will cover)*

*Now, same loop except we mark every spot the ship will occupy with its symbol.*

*Add this shipId to my vector of shipIds.*

*Return true;*

*} repeat for Vertical case.*

*}*

*BoardImpl::uplaceShip(parameters){*

*Check shipId bounds*

*Check if shipId is within vector.*

*If (dir is horizontal)*

*Make sure everything the ship covers is its symbol just in case.*

*For every spot the ship will occupy:*

*Mark on grid that it is now water.*

*Erase shipId from vector*

*Return true;*

*} repeat for vertical case.{*

*Return false if direction wasn’t entered correctly.*

*}*

*BoardImpl::attack(parameters){*

*In case we get an early ‘return false’*

*Set shotHit and shipDestroyed equal to false*

*Declare int k = 0 for later counting purposes.*

*Return false if invalid point location*

*Return false if that location is already marked as hit or water*

*Otherwise,*

*For each ship:*

*If (point on grid is equal to the corresponding shipSymbol)*

*Then mark as hit, and make shotHit true. Break.*

*If ShotHit is still false*

*Mark as missed.*

*If (shotHit is true)*

*Mark shipDestroyed as true.*

*Loop through every grid point and check the ship’s symbol is still on the board.*

*If so, shipDestroyed is false.*

*If (shipDestroyed)*

*Increment number of ships destroyed and set shipId to k.*

*Return true;*

*}*

*GameImpl::play(parameters){*

*Declare a player pointer ‘winner’ to nullptr;*

*If there is any error placing the ships down from either player, return nullptr.*

*Bool gameOn is true.*

*While (gameOn){*

*Put everything within a scope so local variables are deleted at the end.*

*If (p1 is human)*

*Don’t display other player’s ship on board*

*O/w, do.*

*Get a point by calling player’s recommendAttack function.*

*Declare several variables necessary to put into a board attack function.*

*If (there is an error attack the point)*

*validShot is false, and say that the player wasted the shot.*

*O/w,*

*validShot is true.*

*Depending on whether the ship was hit, destroyed, or missed, alter what to cout*

*Check if (p1 is human)*

*Don’t display other player’s ships on board*

*O/w, do.*

*Player will record attack result.*

*If (all of player 2’s ships are destroyed)*

*Print player 1 wins!*

*Winner is player 1, GameOn is false, break;*

*If (shouldPause)*

*Have user press enter to continue*

*} Do same thing for player 2 (again within brackets) except have the player and board roles switched.*

*Finally, return the winner.*

*}*

*For placing ships for a human player, it was a trivial algorithm of asking for user input and making sure everything was fine to call the board’s placeShip function. Also, I did not implement a record attack by opponent function for my good player because I wholeheartedly believe that even if the good player could detect that it is close to losing, it should not play any different. It should always be playing to win!*