COMP-10205 – Data Structure and Algorithms

Assignment#6
Battleship Strategy

Selecting the best data structures and algorithms
7% of course grade (BONUS AVAILABLE)

Submission Requirements

Complete the following exercise and submit electronically in the assignments folder on eLearn as an IntelliJ Project – Zip the entire folder not just the source files in MyCanvas. Please refer the course Calendar for the exact date and time of the submission. **This assignment may be completed in a team of up to three students or can be done individually.** Each student must submit the SAME code if doing as a team. Only the first of the group will be evaluated and all students in the group will receive the same grade

Background

The game of battleship is typically played with two players, each of which place 5 ships of various sizes on a 10 x 10 grid. Ships can be placed horizontally or vertically. Each player on a turn by turn basis attempts to place a shot where the opponent has placed a ship. Of course, your opponent can not see where you have placed your ships and you cannot see where the opponent has placed their ships. You typically call out shots to try and find and sink each of the ships. The first player to sink all the other ships is the winner. For a more complete description see Battleship.

In this instance of BattleShip, the rules have been changed slightly. You are to write a program that will play a limited game of battleship where you will attempt to sink all of the ships with a minimum number of shots. You will write an algorithm to play against the random placement of ships by the BattleShip API. The ships when placed on the board cannot touch in the vertical or horizontal direction. See the diagram for an example on a 10 x 10 grid.

Columns

Rows		0	1	2	3	4	5	6	7	8	9
	0										
	1										
	2		🟧	🟧	🟧	🟧	🟧				
	3										
	4								🟧		
	5		🟧	🟧	🟧				🟧		
	6										
	7			🟧							
	8			🟧							
	9			🟧			🟧	🟧	🟧	🟧	

The top figure shows the placement of the ships with none touching.

Columns

Rows		0	1	2	3	4	5	6	7	8	9
	0										
	1			🟥							
	2			🟥	🟥	🟥	🟥				
	3			🟥						🟧	🟧
	4			🟥					🟧		
	5								🟧		
	6								🟧		
	7			🟧	🟧	🟧	🟧	🟧			
	8										
	9										

In the second figure two ships (shown in red) are touching (the four and the three). This is not allowed in the version being used in this assignment. However, the ships can touch corners diagonally.

In this version the size of the board has been increased to be 15 x 15 with 6 ships (6,5,4,4,3,2). This size and ship lengths will be fixed for the current game.

The starting project provided includes the BattleShip API. The public methods of the Battleship API are described below:

- **`BattleShip2(int numberOfGames, BattleShipBot bot)`** - you need to call the constructor once in your program to create an instance of the battleship game and you must pass a new instance of your solution (see Example Code).
- **`boolean shoot(Point shot)`** - you need to call this function to make each shot.
- **`int numberOfShipsSunk()`** - returns the total number of ships sunk at any point during the game. It is a good idea to use this method to determine when a ship has been sunk.
- **`boolean allSunk()`** - returns a boolean value that indicates whether all the ships have been sunk.
- **`int totalShotsTaken()`** - returns the total number of shots taken. Your code needs to be responsible for ensuring the same shot is not taken more than once.
- **`int[] getShipSizes()`** - returns an array of the ship sizes. The length of the array indicates how many ships are present. This array is fixed for the game with ship sizes of 2,3,4,4,5,6. It does not update when a ship is sunk. You must write logic in your code to determine which ship has been sunk.
- **`int[] void run()`** – runs all of the games until all ships are sunk. It does return an array of scores for each game as well if you want to do some statistics on the games. You do not need to call this method as it is called A6.java file.
- **`void printResults()`** - Prints the results and timing to the screen. You do not need to call this method as it is called in the A6.java file
- **`enum CellState`** - this enum object is very useful for marking cells as either Empty, Hit or Miss. It also has a convenience toString method that can be used for printing purposes. You may also create your own Enum / Class for this in your code, but it is suggested that you use this instead of integers / characters to mark a Cell state

Your solution must implement the **`BattleShipBot`** interface which requires you to override three methods.

- **`void initialize(BattleShip2 battleship)`** – This method will be called once by the BattleShip API at the start of the game. Typically, you would put any code in here that you would have in the constructor for your solution.

- **`String getAuthors()`** – This method will return a string with the author(s) names for the solution

- **`void fireShot()`** – This is your code that should fire a single shot. You must call the battleship API method shoot(Point p) within this code. It is best to only fire a single shot in this method as the API will check to see whether the game is finished after each shot.

The **`ExampleBot`** code provides an example of how each of these methods should be overridden and used.

Steps

1. Download the starting code for the project from the eLearn dropbox. Rename the project to Assignment6
2. Create a class for your battleship Logic - A starter solution is provided in the dropbox. The ExampleBot class can be used as a template for your own solution.
3. Make a copy of ExampleBot and give your bot a name (I called mine YendtBot2021)
4. Add some logic to your class to eliminate duplicate cell selection (i.e. - don't fire on the same cell twice)
5. **DO NOT MODIFY A6.java except to replace the ExampleBot with your class name.**
6. Your hand-in solution needs to play 10000 games. However, for debugging of your solution you may want to set the number of games down to 1.
7. Some Ideas for solving the problem:
   - Create a map that tracks where you have placed your shots so that you don't fire on a cell more than once.
   - After you have hit a ship, you may want to change how you select your next shot.
   - The probability of a ship being on a particular square is not the same for each square. This probability changes after each shot as well. This technique for shot selection is a little more advanced but yields better ship targeting and when used correctly can greatly reduce the number of shots required. See https://www.datagenetics.com/blog/december32011/ for a discussion on different strategies that can be employed.

Useful classes for this lab

The data structures that have been covered in the course could be used to solve the problem. Some classes are better suited than others so spend some time thinking about which classes to use.

Target score this year (15 x 15 – 6 ships)

```
------------------------------------------------------------
B A T T L E S H I P - Version 2.01 [December 22,2021]
------------------------------------------------------------
BattleShip 2 - Results for YendtBot2021
Author : Mark Yendt (Professor CSAIT)
------------------------------------------------------------
The Average Score over 10000 games    = 80.19
Time required to complete 10000 games = 7124 ms
------------------------------------------------------------
Machine specs
Processor   Intel(R) Core(TM) i7-9700T CPU @ 2.00GHz   1.99 GHz
Installed RAM     16.0 GB (15.8 GB usable)
System type 64-bit operating system, x64-based processor
```

Can you do Better?

Marking Scheme

(100% grade is 20/20 - A maximum grade is 24/20)

1. Code quality, modularity (should be refactored into methods , documentation, comments, naming conventions – 20% (4 marks)
2. Optimum Data Structures selected – 20% (4 marks)
3. Mapping employed – 10 % (2 marks)
4. Tracking sunk ships – 10% (2 marks)

- Time Performance – 15% (3 marks) **
    - Bot completes 10000 games in less than 10 s – 3 marks
    - Bot competes 10000 games in under 15 s – 1 mark

      ** Must employ mapping / sink strategy

5. Shot Performance (Average of 10000 games) – 25% (5 marks + BONUS)
    - < 85 (4 Bonus marks) total 9 marks
    - < 95 (2 Bonus Marks) total 7 marks
    - < 102 (5 marks)
    - < 110 (4 marks)
    - < 125 (3 marks)
    - < 150 (2 marks)
    - < 180 (1 mark)