COSC 2123/1285 Algorithms and Analysis
Semester 2, 2018

Assignment 2
Battleship

Due date: 11:59pm Friday, October 12th 2018 **Weight:** 15%

Pairs Assignment

1 Objectives

There are three key objectives for this project:

- Implement a Battleship game.
- Design and implement Battleship guessing algorithms.
- Have fun!

This assignment is designed to be completed in *groups of 2, or pairs*. We suggest that you work in pairs, but you may work individually.

2 Background

Battleship is a classic two player game. The game consists of two players, and each player has a number of ships and a 2D grid. The players place their ships onto the grid, and these ships takes up a number of cells. Each player takes turn at guessing a cell to fire at in their opponent's grid. If that cell contains part of a ship, the player gets a hit. If every part of a ship has been hit, then it is sunk and the owner of the ship will announce the name of the ship sunk (see ships section below for ships in the standard game). The aim of the game is to sink all of your opponent's ships before they sink all of yours. For more details, see `https://en.wikipedia.org/wiki/Battleship_(game)`.

Traditionally, Battleship is played between human players. In this assignment, your group will develop algorithms to automatically play Battleship, that uses a variety of the algorithmic paradigms we have covered in class. It will also give you a taste of artificial intelligence (AI), as algorithms is an important component of AI.

2.1 Ships

In the standard Battleship game, there are five ships available for each side. For this assignment, we have ships that are different from the standard Battleship game and have dimensions as follows:

Name	Dimensions
Patrol Craft	1 by 2 cells
Cruiser	2 by 2 cells
Submarine	1 by 3 cells
Frigate	1 by 4 cells
Aircraft Carrier	2 by 3 cells

3 Tasks

The project is broken up into a number of tasks to help you progress. Task A is to develop a random guessing algorithm as an initial attempt at a Battleship playing agent. Task B and C develops more sophisticated algorithms to play Battleship. For details on how each task will be assessed, please see the "Assessment" section.

To help you understand the tasks better, we will use the example illustrated in Figure 1, which is a Battleship game played on a 10 by 10 grid.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
9										
8										
7										
6										
5										
4										
3										
2										
1										
0										

(a) A 10 by 10 grid.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
9										
8										
7										
6										
5										
4										
3										
2										
1										
0										

(b) A 10 by 10 grid with a submarine and a frigate placed.

Figure 1: Running example used to illustrate the concepts in the tasks.

Task A: Implement Random Guessing Player (3 marks)

In this task, your group will implement a random guessing player for Battleship, which can be considered as a type of brute force algorithm. Each turn, this type of algorithmic player will randomly select a cell it hasn't tried before, fire upon that cell, and continue this process until all the opponent's ships are sunk.

As an example, after some rounds, this random guessing player has fired a few shots and have hit the frigate (see Figure 2). But its next shot will still be a random cell that it hasn't fired upon before (in this example, cell (3,0), highlighted in red). We can do better then this, which is what the next type of player is about (task B).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
9		X								
8						X				
7										
6	X			X						
5								X		
4					X					
3										
2		X								
1								X		
0					X					

Figure 2: A 10 by 10 grid with a submarine and a frigate placed. 'X' denotes a cell fired upon previously. Random guessing player next shot is a random cell it has not fired upon yet. Randomly selecting a cell, the player decides to fire at (6,0) (highlighted as red X).

Task B: Implement Greedy Guessing Player (5 marks)

In this task, your group will make two improvements to the random guessing player. First one is rather than randomly guess, we can utilise the fact that the ships are at least 2 cells long and use the partiy principle. See Figure 3a. As ships are at least of length 2, the player do not need to fire at every cell to ensure we eventually find the opponent's ships. It just need to fire at every 2nd square (Figure 3a). Hence, when hunting for one of the opponent's ships, it can now randonly select a cell from this checkboard type of pattern

The second improvement is to implement more sophicated behaviour once we have a hit. We now divide the process into two parts: *hunting* mode, where the player is seeking opponent's ships (for this task B type of player, they will use the parity guessing improvement), and *targeting* mode, where once there is a hit, the player greedily tries to sink the partially hit/damaged ship. For the targeting mode, once a cell register a hit, we know the rest of the ship must be in one of the four adjacent cells, as highlighted as oranged circles in Figure 3b. The player seeks to destroy the ship before moving on, hence will try to fire at those four possible cells first (assuming they haven't been fired upon, if they have, then no need to fire at a cell twice). Once all possible targeting cells have been exhausted, the player can be sure to have sunk the ship(s) (can be more than one if ships are adjacent to each other) and it returns to the hunting mode until it finds the next ship.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
9	X		X		X		X		X	
8		X		X		X		X		X
7	X		X		X		X		X	
6		X		X		X		X		X
5	X		X		X		X		X	
4		X		X		X		X		X
3	X		X		X		X		X	
2		X		X		X		X		X
1	X		X		X		X		X	
0		X		X		X		X		X

(a) Parity idea. The player doesn't need to fire at every cell to guarantee all ships are at least hit. The cells with Xs (can be the other alternative, if the top left most X is at (9,1)), are the ones to try.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
9										
8		X			X					
7										
6				X						
5						O				
4		X			O	X	O			
3						O				
2		X								
1									X	
0						X				

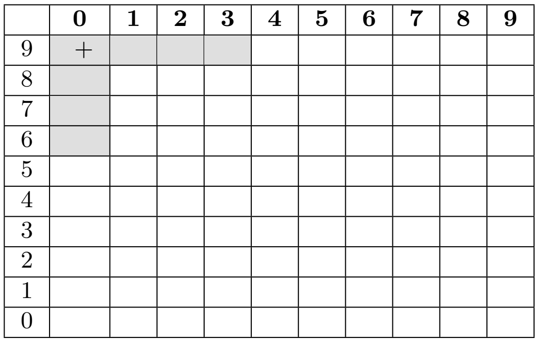
(b) In targeting mode. Red X denote a hit, and the next four cells to target are the ones with orange Os.

Figure 3: Illustration of the parity principle and (greedy) targeting mode.

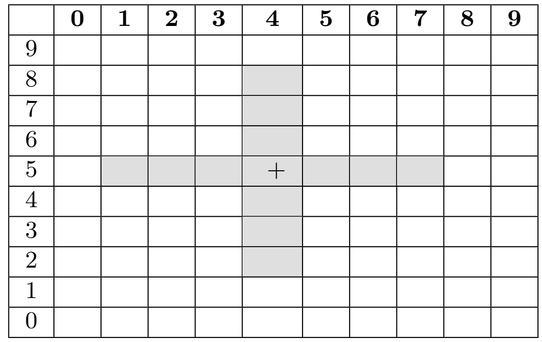
Task C: Implement Probabilistic Guessing Player (5 marks)

In this task, your group will implement a smarter type of player. This one is based on the transform-and-conquer principle, where we do some preprocessing to improve our hunting and targeting strategies.

When a ship is sunk, the opponent will indicate which ship of theirs have been sunk. We can make use of this fact to improve both the hunting and targeting mode. In the two previous type of players they assumed every cell is as likely to contain a ship. But this is unlikely to be true. For example, consider the frigate and a 10 by 10 grid. It can only be in two placement configurations if placed in top left corner (see Figure 4a), but in 8 different placement configurations if part of it occupies one of the centre cell, e.g., cell (4,4) (see Figure 4b). Hence, assuming our opponent randomly places ships (typically they don't, but that is beyond this course, as we are going towards game theory and more advanced AI), it is more likely to find the frigate occupying one of the centre cells. This exercise can be repeated for all ships, and for each ship, we end up with a count of the number of ship configurations



(a) The number of configurations (2) that the frigate placed into top left cell (highlighted with the '+').



(b) The number of configurations (10) that the frigate placed into top left cell (highlighted with the '+').

Figure 4: Illustration of the idea behind counting the number of ship placements that can go through the '+' cell.

that can occupy that cell. The cell with the highest total count over all ships is the one most likely to contain a ship.

In hunt mode, this type of player will select from those cells yet to be fired upon, the one with the highest possible ship configuration count (if there are several, randomly select one). If there is a miss, then the count of that cell and neighbouring cells (because we missed, it means there isn't any ship that can occupy that cell, so need to update its count and the neighbouring ones, as the count of neighbour ones may depend on a ship being upon to fit onto the fired upon cell). If hit, we go to targeting mode. In targeting more, the player makes use of the fact that there has been a hit to calculate which adjacent cell is the most likely to contain a ship (with the highest configuration count). Using the same counting method as the hunting mode, we can calculate the number of possible ship configurations that pass through the hit cell (remember previous misses, previously sunk ships and grid boundaries should be considered as obstacles and taken into account). When we get a miss or another hit, update the counts correspondingly and repeat at firing at an adjacent cell with the highest count. Once a ship is sunk, the counts of the whole grid must be updated to reflect this ship is no longer in play. When the player has sunk the ship(s), then it goes back to hunting mode.

4 Details for all tasks

To help you get started and to provide a framework for testing, you are provided with skeleton code that implements some of the mechanics of the game. The main class (BattleshipMain) implements functionality of a two player Battleship game, a method to log the game to check the correctness and to parse parameters. The list of files provided are listed in Table 1.

The framework is designed such that each player can have their own implementation. This allows your players to play against some of ours, or even other groups (given certain conditions are satisfied, please ask your lecturer first). Also, it defines how the players should interact. Examine BattleshipMain.java, particularly the code that iterates through the rounds. Note each player takes turn at making a guess via a Guess object, then the opponent answers via an Answer object and this is passed back to the first player. Examine the Guess and Answer classes and see "Guess Structure" and "Answer Structure" below to understand how they are implemented in this framework.

The framework also automatically logs the guess-answer traces of the game. This is one mechanism for us to evaluate if your players implementations are correct (see "Assessment" section for more details).

file	**description**
`BattleshipMain.java`	Class implementing basic framework of the Battleship game. *Do not modify unless have to.*
`player/Player.java`	Interface class for a player. *Do not modify this file.*
`player/RandomGuessPlayer.java`	Class implementing the random guessing player (task A).
`player/GreedyGuessPlayer.java`	Class implementing the greedy guessing player (task B).
`player/ProbablisticGuessPlayer.java`	Class implementing the Monte Carlo guessing player (task C).
`player/Guess.java`	Class implementing a 'guess'. *Do not modify this file.*
`player/Answer.java`	Class implementing an 'answer'. *Do not modify this file.*
`ship/Ship.java`	Interface class for a ship. *Do not modify this file.*
`ship/PatrolCraft.java`	Class implementing a patrol craft ship. *Do not modify this file.*
`ship/Submarine.java`	Class implementing a submarine ship. *Do not modify this file.*
`ship/Cruiser.java`	Class implementing a cruiser ship. *Do not modify this file.*
`ship/Frigate.java`	Class implementing a frigate ship. *Do not modify this file.*
`ship/AircraftCarrier.java`	Class implementing an aircraft carrier ship. *Do not modify this file.*
`world/World.java`	Class implementing the "world" of the game for a player, including the grid, location of their ships and where their opponent have fired before. It is used for visualisation. If you need to store game information, we suggest use attributes in the *Player classes. *Do not modify this file.*
`world/StdDraw.java`	Class that implements visualisation. *Do not modify this file.*

Table 1: Table of supplied Java files.

Note, you should not modify BattleshipMain class, as this contains the code for the game mechanics and the logging code and you do not want to break this. We also strongly suggest to avoid modifying the "Do not modify" ones, as they form the interface between players and basic ship information. You may add methods and java files, but it should be within the structure of the skeleton code, i.e., keep the same directory structure. Similar to assignment 1, this is to minimise compiling and running issues. However, you can change the *Player.java files, including implementing/extending from a common player parent class. However, ultimately your *Player classes must implement the Player interface.

Note that the onus is on you to ensure correct compilation on the core teaching servers.

As a friendly reminder, remember how packages work and IDE like Eclipse will automatically add the package qualifiers to files created in their environments.

Guess structure

(row,col) coordinates of the cell fired at.

Answer structure

The answer contains two attributes, isHit and shipSunk. isHit is a boolean, and should be set to True if a ship was hit by the latest shot, and False if missed ships. In addition, if a ship is destroyed after the hit, shipSunk should additionally be set to the object of the ship destroyed, one of {PatrolCraft, Cruiser, Submarine, Frigate, AircraftCarrier}.

Compiling and Executing

To compile the files, run the following command from the root directory (the directory that BattleshipMain.java is in):

```
javac -cp .:samplePlayer.jar BattleshipMain.java
```

Note that for Windows machine, remember to replace ':' with ';' in the classpath.

To run the Battleship framework:

`java -cp .:samplePlayer.jar BattleshipMain [-v] [-l <game log file>] <game configuration file> <ship location file 1> <ship location file 2> <player 1 type> <player 2 type>` where

- -v: whether to visualise the game.
- game log file: name of the file to write the log of the game.
- game configuration file: name of the file that contains the configuration of the game.
- ship location file 1: name of file containing the locations of each ship of player 1.
- ship location file 2: name of file containing the locations of each ship of player 2.
- player 1 type: specifies which type of algorithmic player to use for the first player, one of [random | greedy | prob | sample]. random is the random guessing player, greedy is the greedy guessing player, prob is the probabilistic guessing player, bonus is the bonus task player and sample is a sample player we provided for you to initially play with.
- player 2 type: specifies which type of algorithmic player to use for the second player, one of [random | greedy | prob | sample].

The jar file contains the sample player to get you going.

We next describe the contents of the game configuration and chosen person files.

4.1 Details of Files

Game configuration file

The game configuration files specifies the dimensions of the grid in a Battle game. The file has the following format:

```
[# of rows] [# of columns]
```

The row and column numbers are positive integers, and separated by a space.

An example game configuration file is as follows:

```
10 20
```

This specifies the following Battleship game configuration:

- The grid is 10 rows by 20 columns.

Ship location file

The ship location file specifies the location of the ships of each player. It is formated as follows:

```
[ship name] [row coordinates] [column coordinates] [primary direction the ships spans
[secondary direction the ship spans]
```

The values are separated by space.
Ship names are one of {PatrolCraft, Cruiser, Submarine, Frigate, AircraftCarrier}. Directions are one of {N (North), S (South), E (East), W (West)}

An example ship location file is as follows:

```
AircraftCarrier 1 1 E S
Frigate 2 5 S E
```

len width

This specifies the following ship placements:

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

Figure 5: A 10 by 10 grid with a aircraft carrier at "1 1 E S" (red) and a frigate at "2 5 S E" (green).

This can also equally be specified as:

```
AircraftCarrier 2 3 W N
Frigate 5 5 N E
```

As sample, we provide:

- For normal (rectangular) grid world, a "config.txt", "loc1.txt" and "loc2.txt" as the configuration and ship location files.

4.2 Clarification to Specifications

Please periodically check the assignment FAQ for further clarifications about specifications. In addition, the lecturer will go through different aspects of the assignment each week, so even if you cannot make it to the lectures, be sure to check the course material page on Blackboard to see if there are additional notes posted.

5 Assessment

The project will be marked out of 15.

The assessment in this project will be broken down into a number of components. The following criteria will be considered when allocating marks. All evaluation will be done on the core teaching servers.

For all tasks, a cell should not be fired upon more than once. In addition, answering should be correct, e.g., your implementation should not return False in Answer.isHit when a ship is actually hit. If either of these are false, this will be considered as an incorrect algorithm.

Task A (3/15):

For this task, we will evaluate your player algorithm on whether:

1. It implements a random guessing strategy, as outlined in the specifications.
2. Produces a correct guessing trace, i.e. no cell fired upon more than once, and answering is correct.

Task B (5/15):

For this task, we will evaluate your player algorithm on whether:

1. It implements a greedy guessing strategy, as outlined in the specifications.
2. Produces a correct guessing trace, i.e. no cell fired upon more than once, and answering is correct.
3. Additionally, over a number of games, does it on average, beat the random guessing player of task A, i.e., does it win more than it loses against the random guessing player?

Task C (5/15):

Similar to task B, for this task, we will evaluate your player algorithm on whether:

1. It implements a probabilistic guessing player, as outlined in the specifications.
2. Produces a correct guessing trace, i.e. no cell fired upon more than once, and answering is correct.
3. Additionally, over a number of games, does it on average, beat the player types of task A and B, i.e., does it win more than it loses against each of the other two types of players.

Coding style and Commenting (2/15):

You will be evaluated on your level of commenting, readability and modularity. This should be at least at the level expected of a second year undergraduate student who has done some programming courses.

5.1 Late Submissions

Late submissions will incur a *deduction of 1.5 marks per day or part of day late.* Please ensure your submission is correct (all files are there, compiles etc), resubmissions after the due date and time will be considered as late submissions. The core teaching servers and blackboard can be slow, so please ensure you have your assignments are done and submitted a little before the submission deadline to avoid submitting late.

6 Team Structure

This project is designed to be done in *pairs* (group of two). If you have difficulty in finding a partner, post on the discussion forum or contact your lecturer. If there are issues with work division and workload in your group, please contact your lecture as soon as possible.

In addition, please submit what percentage each partner made to the assignment (a contribution sheet will be made available for you to fill in), and submit this sheet in your submission. The contributions of your group should add up to 100%. If the contribution percentages are not 50-50, the partner with less than 50% will have their marks reduced. Let student A has contribution X%, and student B has contribution Y%, and $X > Y$. The group is given a group mark of M. Student A will get M for assignment 1, but student B will get $\frac{M}{\frac{X}{Y}}$.

7 Submission

The final submission will consist of:

- Your Java source code of your implementations. We will provide details closer to submission date.

Note: submission of the code will be done via Blackboard.

8 Plagiarism Policy

University Policy on Academic Honesty and Plagiarism: You are reminded that all submitted project work in this subject is to be the work of you and your partner. It should not be shared with other groups, nor should you elicit external tutoring services to do the assignment for you. Multiple automated similarity checking software will be used to compare submissions. It is University policy that cheating by students in any form is not permitted, and that work submitted for assessment purposes must be the independent work of the student(s) concerned. Plagiarism of any form will result in zero marks being given for this assessment, and can result in disciplinary action.

For more details, please see the policy at `http://www1.rmit.edu.au/students/academic-integrity`.

9 Getting Help

There are multiple venues to get help. There are weekly consultation hours (see Blackboard for time and location details). In addition, you are encouraged to discuss any issues you have with your Tutor or Lab Demonstrator. We will also be posting common questions on Blackboard and we encourage you to check and participate in the discussion forum on Blackboard. Although we encourage participation in the forums, please refrain from posting solutions.