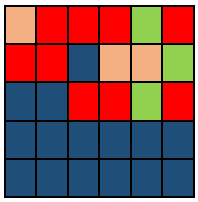
had

Lab 6: Mini-project - Problem G: Spot the ships

Submit source code and UML diagram in PDF to Mooshak problem G at:

http://deei-mooshak.ualg.pt/~hdaniel



up to May 31, 2021

This lab assignment submission must include:

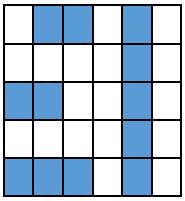
1) the **source code**
2) a report in a *.pdf file (around 5 pages should be enough) with:
   a) the **corresponding UML class diagram**
   b) for each class **the description of its responsibilities**
   c) **design patterns used** and what classes implements each design pattern
   d) a brief explanation on how the classes work together to solve problem G

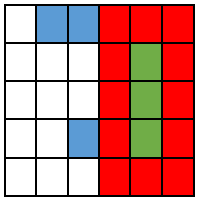
When submitting to Mooshak include inside the folder with the source code, a **report.pdf** file with the report.

Make sure that all the c**onstraints specified below** and required for validation, are met before discussion.

1. Description

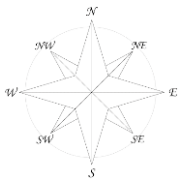
It is intended to develop a spotlight system to scan and spot ships in a bay or harbour during the night. The dimensions of the bay or harbour can vary but can be modelled by a rectangular board with a given number of rows and columns. Ships occupies some contiguous positions or cells of the board, without touching. In the example below, board on the left, no ship is touching other ship. In the board on the right, assuming that the green ship was the first placed in the board, red cells are invalid for positioning other ships:





There can be ships with size 1 to 5 cells. These ships can be oriented horizontally or vertically only. From a given position there are 8 neighbours, named by the cardinal and ordinal directions:

	NW	**N**	**NE**	
	W		**E**	
	SW	**S**	**SE**	



When a chip cell is spotted, its diagonal neighbours, **NE, SE, SW, NW**, cannot have a ship cell. If this ship has more cells, they must be at one of the other neighbours: **N, E, S, W**.

had

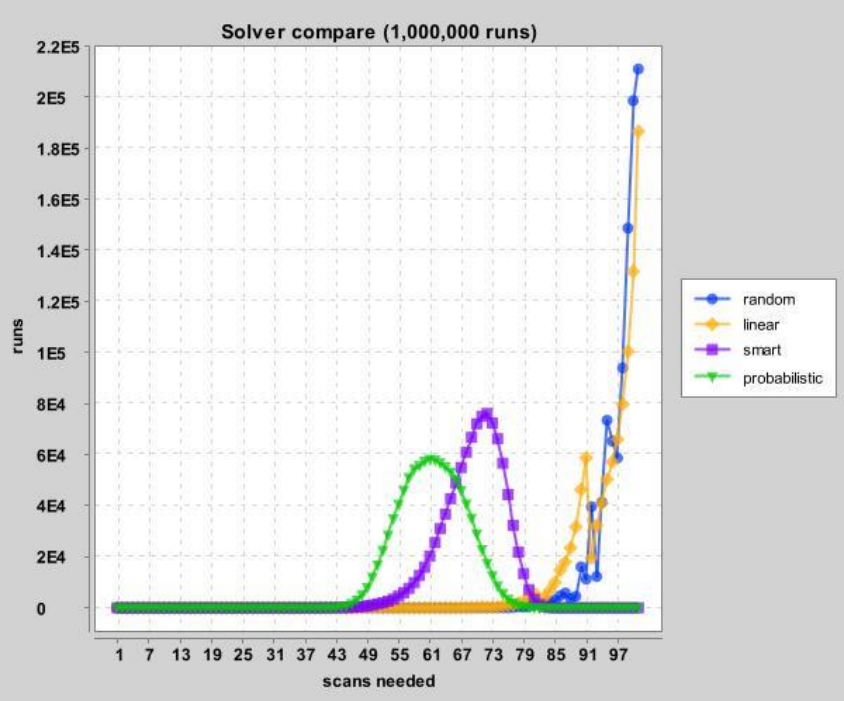
Sometimes it will be needed to spot and check all the ship cells in the harbour to make sure everything is all right. However, to save energy, and thus generating the minimum possible environmental footprint, the number of cells scanned must be also the minimum possible.

Several scanning strategies that ensure all ship cells are spotted can be used, some more efficient than others. The most inefficient ones are randomly scan or linear scan.

Strategies efficiency comparison

The graphic below compares 4 possible strategies, with a board size 10x10 and 8 randomly placed ships: two size 1 ships, two size 2 ship, two size 3 ships, one size 4 ship and one size 5 ship, a total of 21 ship cells to spot.

There were used 1,000,000 ship placement combinations. This did not exhaust all the possible ship placement combinations, and some of the combinations might be repeated.



The horizontal axis represents the number of scans required to spot all the ships. Vertical axis the number of combinations, or simulation ran. Since board is 10x10 the worst case for spotting all ship cells is 100 scans and the best case, ie not missing a scan, is 21.

Random board scanning or **Linear** scanning have similar results, since ships are placed randomly for each of the 1,000,000 board runs. Average scans to spot all ships is 96 for a random scan strategy and 94 for a **linear scan strategy, scanning from left to right and from top to bottom**.

Since ships cannot touch, there is no need to scan some neighbour cells of a spotted ship cell. A **Smart** scanning strategy, described below, that takes this into account is more efficient. Average of scans to spot all ship cells with these strategies dramatically drops to 68.

Another possible strategy takes in consideration the space that ships take, the space left in the board and gives a probability or likelihood of where in the board can be ship cells. This way, the next position to be scanned is chosen with more information than just linearly or randomly. This **Probabilistic** strategy can also eliminate neighbour cells not needed to scan. It preforms slightly better, than the smart strategy, the average of scans to spot all ship cells is 60.

had

2. Task

Develop a program that implements the three scanning strategies described below: **linear**, **smart** and **probabilistic**, for a given board with some ships. For each different strategy, present the **final board**, after all ships be spotted and also the number of scans that was needed by the strategy to spot all ships.

Board representation

For output purposes, a board is represented by a matrix of characters, with dimension rows x columns. A ship cell is represented by a capital 'O'. A spotted ship cell by a plus sign '+'. A capital 'X' represents a scan that missed a ship (reaching water) or cells not needed to be scanned, eliminated by the algorithm. Finally, a dot '.' represents a (water) cell not scanned.
Below is an example of an initial board in the left, and a board after some scans, in the right:

```
.....O          xxxxx+
OOOO..          ++OOxx
......          xxx...
...O..          ...O..
O..O..          O..O..
```

Implementing just the Linear strategy, but taking in consideration the constraints, developing the program using the studied principles of OOP, submitting a complete report and be able to explain what was done and how during the discussion, should be enough to achieve, for this problem assessment, a small but positive grade.
It is suggested to implement first the Linear strategy, to have a base functional structure for the implementation. Then submit it to Mooshak and make sure it passes the Linear strategy tests, before starting implementing the other strategies.

3. Constraints

The validation of an accepted submission depends on all the following requirements:

1) Client class must be called **SpotTheShips**.

2) Use the design pattern "**Strategy**" to implement the playing strategies.

3) For all methods, include in the comments, pre and post conditions.

4) Choose at least 2 classes and define JUnit 5 unit tests for all methods.

4. Suggestions for the implementation

To find classes, method and instance variables, ask if it is needed to store some information, do some task and who needs to do what:

- A position is represented by a row and a column coordinates.
- Scanned cells are a list of positions.
- Neighbour ship cells are a set of positions relative to a single position.
- A ship has some contiguous cells or positions.
- A board has all the possible positions and some ships.
- A board should be converted to a string to be printed.
- To print a board, it is needed to know the cells that were scanned, and from those which cells have ships.

had

The only information that any scanning strategy needs to know, to solve the problem is whether a scanned position is a ship cell or water. This info can be provided by the board, since this class has the ships.

List manipulation can be done with java interface **List** and class **ArrayList**.

5. Scanning strategies

a) Linear strategy

Linear strategy simply keeps scanning cells from left to right and from top row to bottom row. A list of Positions sorted this way, **boardList** is created. For a board with **r** rows and **c** columns, the initial list contains r * c positions, sorted as:

(0,0) (0,1) (0,2) … (0,c-1) (1,0) (1,2) (1,3) … (1,c-1) (r-1,0) (r-1,1) … (r-1,c-1)

Removing always the first position in the **boardList** at each turn, ensures that every position in the board will be at most scanned once.
Also, an empty list called **scanList** is created. All scanned cells are added to **scanList**. This **scanList** will be useful to print the current state of the board.
The algorithm will keep scanning cells until the number of ship cells to spot, **count**, reaches zero.

```
solveLinear(board)
   Init boardList with all positions in board sorted by rows, left to right
   Init empty scanList

   count = number of ship cells to spot in board
   While count > 0
      pos = remove first position from boardList
      add pos to scanList
      if pos == ship cell
         count = count -1
```

Example board states, for a given board. The linear scan will end when the last ship cell is spotted:

Initial board		Final board
`......`	**`XXXXXX`**	`XXXXXX`
`.OOO..`	**`X+`**`OO..`	`X+++XX`
`......`	`......`	`XXXXXX`
`...O..`	`...O..`	`XXX+XX`
`...O..`	`...O..`	`XXX+..`

There is a full example in Appendix A.

b) Smart strategy

The Smart algorithm is similar to the Linear one, scanning board from left to right and from top to bottom, but when a ship cell is spotted, diagonal neighbours (NE, SE, SW, NW) are eliminated from search, since there is no needed to scan them, because ships cannot touch each other.

had

Then East and South neighbours, if they are present, are moved to the front of the search queue **boardList** to be scanned first. These are the only differences to the Linear algorithm.
As an example, consider the board below. After spotting the first ship cell at position (1,1), diagonal neighbours still present in **boardList** are removed from this list. In this case there are only 2, the ones in red. These two positions are also added to **scanList**:

```
XXXXXX      boardList: (1,1) (1,2) (1,3) (1,4) (1,5) (2,0) (2,1) (2,2) (2,3) ...
X+OO..
X.X...      scanList:  (0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (1,0)
...O..
...O..
```

Then, if present in **boardList,** East and South neighbours are moved to the front of the list, first the East and then the South neighbour:

```
XXXXXX      boardList: (1,2) (2,1) (1,3) (1,4) (1,5) (2,3) (2,4) ...
X+1O..
X2X...      scanList:  (0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (1,0) (1,1)(2,0)(2,2)
...O..
...O..
```

Note that when a E neighbour is also a ship cell, eliminating this E neighbour's diagonal neighbours, also eliminate from the **boardList** the previous S neighbour, cell (2,1)

```
XXXXXX      boardList: (1,3) (1,4) (1,5) (2,4) (2,5) (3,0) ...
X++O..
XXXX..
...O..
...O..
```

This procedure is repeated until all ship cells are spotted, ie. when count = 0. The final board for this case is:

```
XXXXXX
X+++XX
XXXXXX
XXX+X.
..X+X.
```

The full example is in Appendix B.

The **Smart** algorithm in pseudo code is below. Differences from the **Linear** algorithm are in red:

```
solveSmart(board)
   Init boardList with all positions in board sorted by rows, left to right
   Init empty scanList

   count = number of ship cells to spot in board
   While count > 0
      pos = remove first position from boardList
      add pos to scanList
      if pos == ship cell
         remove diagonal neighbours from boardList
         add diagonal neighbours to scanList
         move E and S neighbours to front of boardList if in boardList
         count = count -1
```

had

This algorithm still scans from left to right and bottom to top. A random scanning fashion might have more success for some boards. However, to avoid possible differences of implementations of Random class in different versions of java, that might lead to different valid solutions, the random scanning was dropped.

c) Probabilistic strategy

This algorithm tries to discover the most likely positions where a ship can be. A matrix of likelihood has an integer value for all the positions in the board. The higher the value in a matrix position the most likely is a ship cell at that same position in the board.

The likelihood matrix is computed before each scan, to give the next position to scan.
To compute the value of the matrix cells, it is tested what ship sizes can fit in the board cells. If a ship with some size fits in a range of cells, all these cells are incremented 1 unit.

Let's see an example. Consider an empty board of size 5x6. Since the board is empty, all ships will fit, but some positions are more likely to has a ship cell.
So, for each size of ship, from 1 to 5, starting with size 1 test if ship fits in board cells. For this board, each row has 6 cells. Each cell can always have a size 1 ship, so each cell in the matrix starts with value 1. For size 2 ships in a row of 6 cells there are 5 different ways to place them without repetition, for 3 size ships 4 ways, for 4 size ships 3 ways and for 5 size ships 2 ways:

Ship size	**1**	**2**	**3**	**4**	**5**
	1.....	22....	333...	4444..	55555.
Possible	.1....	..22..	...333	.4444.	.55555
placement	..1...	22	.333..	..4444	
in a row	...1..	.22...	..333.		
	....1.	...22.			
	1				
Total for row	**111111**	**122221**	**123321**	**123321**	**122221**

Adding one, in the corresponding matrix cell, for each cell in the row that can have the ship, in a combination, we have the total likelihood, for each row cell and for each size of ship. This total in the last line.
Now adding the total for each ship, we have the relative likelihood of having a ship cell in each row:

	1	**1**	**1**	**1**	**1**	**1**
	1	**2**	**2**	**2**	**2**	**1**
	1	**2**	**3**	**3**	**2**	**1**
	1	**2**	**3**	**3**	**2**	**1**
	1	**2**	**2**	**2**	**2**	**1**
Total for each row	**5**	**9**	**11**	**11**	**9**	**5**

For each column it is also needed to compute likelihood. The process is the same, now we have columns of 5 cells. The combinations for 5 cells are:

Ship size	**1**	**2**	**3**	**4**	**5**
	1....	22...	333..	4444.	55555
Possible	.1...	..22.	..333	.4444	
placement	..1..	.22..	.333..		
in a row	...1.	...22			
	1				
Total	**11111**	**12221**	**12321**	**12221**	**11111**

had

Now adding the total for each ship, we have the relative likelihood of having a ship cell in each column:

```
                          1  1  1  1  1
                          1  2  2  2  1
                          1  2  3  2  1
                          1  2  2  2  1
                          1  1  1  1  1

Total for each column     5  8  9  8  5
```

Summing all columns and rows we get the likelihood matrix for an empty 5x6 board:

```
5 9 11 11 9 5          5 5 5 5 5 5          10 14 16 16 14 10
5 9 11 11 9 5          8 8 8 8 8 8          13 17 19 19 17 13
5 9 11 11 9 5    +     9 9 9 9 9 9    =     14 18 20 20 18 14
5 9 11 11 9 5          8 8 8 8 8 8          13 17 19 19 17 13
5 9 11 11 9 5          5 5 5 5 5 5          10 14 16 16 14 10
```

The next position to scan, is the first higher value in the likelihood matrix, searching form left to right and from bottom to top. In this example is matrix cell **(2,2) = 20**.

The likelihood matrix is computed before each scan, to get the next position to scan. After scanned, this position is given impossible priority -1, to make sure it will not be scanned again.
So, if position (2,2) is not a ship cell, the likelihood matrix is simply updated putting -1 at this cell and resetting all the cells. If it is a ship cell, the diagonal neighbours are also set to -1, as we can see in the matrix examples respectively on the left and right:

```
0  0  0  0  0  0          0  0  0  0  0  0
0  0  0  0  0  0          0 -1  0 -1  0  0
0  0 -1  0  0  0          0  0 -1  0  0  0
0  0  0  0  0  0          0 -1  0 -1  0  0
0  0  0  0  0  0          0  0  0  0  0  0
```

From here the matrix must be computed again. Now each **-1** is a barrier, for instance for the middle row will be computed now as:

```
2  2 -1  2  3  2
```

At the left of -1 there can be placed only 2 1 size ships and one size 2 ship, so both cells to the left have value 2. To the right we have 3 cells. We can place three size 1 ships, one in each cell, two size 2 ships and one size 3 ship, giving: 2 3 2.
Computing the same way the column where are the solo -1, it has likelihood:

```
2  2 -1  2  2
```

Computing every row and column, as above, we get for the 2 cases above:

```
10 14 16 16 14 10          10 10 13 12 14 10
13 17 19 19 17 13          9  -1  3 -1 10 10
14 18 -1 20 18 14          11  3 -1  4 13 12
13 17 19 19 17 13          9  -1  3 -1 10 10
10 14 16 16 14 10          10 10 13 12 14 10
```

the highest likelihood and next position are in red, respectively (1,2) = 19 and (0,4) = 14.

had

The probabilistic algorithm

The **probabilistic** algorithm takes the next position from a **highProbabilityList** if not empty. If empty the position is taken from the most likely position in the likelihood matrix.
This position is added to the **scanList**.
If this position has a ship cell, the diagonal neighbour cells, NE, SE, SW, NW, are given impossible priority, since there is no needed to scan them, because ships cannot touch each other. Finally, to try spotting nearby ship cells first, the neighbour N, E, S, W positions are put in an **highProbabilityList**, by this order, to be scanned next. **If this list is not empty, next position will be the first in this list**.

In pseudo code, the probabilistic algorithm is:

```
solveProbabilistic(board)
   Init LikeliHoodMatrix
   Init empty highProbabilityList
   Init empty scanList

   count = number of ship cells to spot in board
   While count > 0
      if highProbabilityList NOT empty
          pos = remove first position from highProbabilityList
       else
          compute LikeliHoodMatrix
          pos = first high probability position from LikeliHoodMatrix

       add pos to scanList
       set pos as scanned in LikeliHoodMatrix
       if pos == ship cell
          count = count -1

          add diagonal neighbours to scanList
          set diagonal neighbours as scanned in LikeliHoodMatrix
          remove diagonal neighbours from highProbabilityList

          add neighbours to highProbabilityList (sorted as N, E, S, W)
```

A full example is in Appendix C.

6. Input

The input has 3 lines. The first line has only a string “linear”, “smart” or “prob”, that specifies the search strategy to use. The second line is the size of the board: rows and columns, specified with 2 integers and a third integer that specifies the number of ships in the third line. The third line has a list of ships in the board, each one with 3 integers and a character. The first 2 integers define the position of the first cell of the ship in the board, the third integer the size of the ship, the number of cells. Finally, the character gives the orientation, how cells are placed from first position: ‘E’ to East or ‘S’ to South:

linear	represents this board:	`......`
5 6 2		`.0000.`
1 1 4 E 3 3 2 S		`......`
		`...0..`
		`...0..`

7. Output

The output has a line with the number of scans needed to show all ships. Below this line is the final board, represented as defined in **2. Task**.
Note that number of scans in the first line is less or equal to the sum of all 'X's and '+'s.

8. Operation samples

Sample Input 0
```
linear
5 6 2
1 1 4 E 3 3 2 S
```

Sample Output 0
```
28
XXXXXX
X++++X
XXXXXX
XXX+XX
XXX+..
```

Sample Input 1
```
smart
5 6 2
1 1 4 E 3 3 2 S
```

Sample Output 1
```
18
XXXXXX
X++++X
XXXXXX
XXX+X.
..X+X.
```

Sample Input 2
```
prob
5 6 2
1 1 4 E 3 3 2 S
```

Sample Output 2
```
11
XXXXXX
X++++X
XXXXXX
..X+X.
..X+X.
```

Sample Input 3
```
linear
4 4 2
1 1 1 E 3 0 4 E
```

had

Sample Output 3

```
16
XXXX
X+XX
XXXX
++++
```

Sample Input 4

```
smart
3 10 5
0 0 2 E 2 0 1 E 2 5 5 E 0 6 4 E 1 3 1 E
```

Sample Output 4

```
20
++XXXX++++
XXX+XXXXXX
+XXXX+++++
```

Sample Input 5

```
prob
10 10 5
0 0 1 E 1 2 2 E 3 3 3 S 4 5 4 S 9 4 5 E
```

Sample Output 5

```
58
+XXXX.XX.X
XX++XXX.X.
XXXXXX.X.X
.XX+XXXXXX
XXX+X+X.X.
X.X+X+XX.X
.XXXX+XXXX
X.X.X+X.X.
.X.XXXXXXX
X.XX+++++X
```

9. References and history

The Smart strategy is very much the same that humans usually use to play battleships board game, since this problem is very similar. However, in battleships when there is a hit, it is known the size of the ship, while here it is not.

The likelihood matrix used in the **Probabilistic strategy** was proposed in:

https://www.datagenetics.com/blog/december32011

to develop a probabilistic algorithm to play battleships. However, the version of battleships discussed there, allows ships to touch. Also, there is no size 1 ships, which gives better results.
The probabilistic strategy proposed here uses a likelihood matrix, similar to the one proposed in the reference above, however the method to localize ships and eliminate unneeded scans is different and similar to the one used in the Smart strategy.
The performance, for the same number of ships, is similar to the one in the reference above.

Appendix A: Linear solver algorithm example

```
solveLinear(board)
   Init boardList with all positions in board sorted by rows, left to right
   Init empty scanList

   count = number of ship cells to spot in board
   While count > 0
      pos = remove first position from boardList
      add pos to scanList
      if pos == ship cell
         count = count -1
```

```
Board       keeps scanning from top to bottom, from left to right

......      Initial board and scanning order:
.OOO..
......      (0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (1,0) (1,1) (1,2) ... (4,5)
...O..
...O..

XXXXXX      After 8 scans
X+OO..
......      (1,2) (1,3) (1,4) (1,5) (2,0) (2,1) (2,2) (2,3) (2,4) ... (4,5)
...O..
...O..

XXXXXX      After 28 scans
X+++XX
XXXXXX      (4,4) (4,5)
XXX+XX
XXX+..
```

Appendix B: Smart solver algorithm example

```
solveSmart(board)
   Init boardList with all positions in board sorted by rows, left to right
   Init empty scanList

    count = number of ship cells to spot in board
   While count > 0
      pos = remove first position from boardList
      add pos to scanList
      if pos == ship cell
         remove diagonal neighbours from boardList
          add diagonal neighbours to scanList
         move E and S neighbours to front of boardList if in boardList
         count = count -1
```

An example of the smart algorithm is shown below. **Note that** all scanned cells are added to **scanList**. When a ship cell is spotted, all its diagonal neighbours are also added to **scanList.** To simplify the example this info is omitted from the comments.

```
Board        List lead positions (ordered by row, left to right)

......       (0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (1,0) (1,1) (1,2) ...
.OOO..
......
...O..
...O..

XXXXXX       keep removing from the front of boardList until ship cell spotted
X+OO..
......       (1,1) (1,2) (1,3) (1,4) (1,5) (2,0) (2,1) (2,2) (2,3) ...
...O..
...O..


XXXXXX       Ship cell spotted.  Remove spotted cell and also
X+OO..       diagonal neighbours from boardList if present
X.X...
...O..       (1,1) (1,2) (1,3) (1,4) (1,5) (2,0) (2,1) (2,2) (2,3) ...
...O..       (1,2) (1,3) (1,4) (1,5) (2,1) (2,3) (2,4) ...


XXXXXX       Set to scan first E and second S neighbours of cell
X+1O..       if present, move them to the front of the boardList, E first
X2X...
...O..       (1,2) (1,3) (1,4) (1,5) (2,1) (2,3) (2,4) ...
...O..       (1,2) (2,1) (1,3) (1,4) (1,5) (2,3) (2,4) ...

REPEAT until all ship cells are spotted. (count = 0)
Note: We know the number of ship cells to spot, since it is given.

continuing the example:
next iteration of REPEAT:

XXXXXX       Try E neighbour, first in list. Remove it from boardList. Ship cell
X++O..       spotted, remove diagonal neighbours (this removes also S neighbour)
XXXX..
...O..       (1,2) (2,1) (1,3) (1,4) (1,5) (2,3) (2,4) ...
...O..       (1,3) (1,4) (1,5) (2,4) (2,5) (3,0) (3,1) (3,2) (3,3) ...
```

```
XXXXXX      Set to scan first E and second S neighbours of cell
X++1..      if present, pass them to the front of boardList, E first
XXXX..
...O..      (1,3) (1,4) (1,5) (2,4) (2,5) (3,0) (3,1) (3,2) (3,3) ...
...O..      (1,3) (1,4) (1,5) (2,4) (2,5) (3,0) (3,1) (3,2) (3,3) ...

next iteration of REPEAT:

XXXXXX      Try E neighbour, again first in list. Remove it from boardList and
X+++..      remove diagonal neighbours (this removes also S neighbour)
XXXXX.
...O..      (1,3) (1,4) (1,5) (2,4) (2,5) (3,0) (3,1) (3,2) (3,3) ...
...O..      (1,4) (1,5) (2,5) (3,0) (3,1) (3,2) (3,3) ...


XXXXXX      Set to scan first E and second S neighbours of cell
X+++1.      if present, pass them to the front of boardList, E first
XXXXX.
...O..      (1,4) (1,5) (2,5) (3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (4,0) ...
...O..      (1,4) (1,5) (2,5) (3,0) (3,1) (3,2) (3,3) (3,4) (3,5) (4,0) ...

next iterations of REPEAT:

XXXXXX      keep removing cells from front of boardList until ship cell spotted
X++++X
XXXXXX
XXXO..      (1,4) (1,5) (2,5) (3,0) (3,1) (3,2) (3,3) (3,4) (3,5) ...
...O..      (3,3) (3,4) (3,5) (4,0) (4,1) (4,2) (4,3) (4,4) (4,5)

next iteration of REPEAT:

XXXXXX      Try E neighbour, again first in list. Remove it from boardList and
X++++X      remove also diagonal neighbours (this removes also S neighbour)
XXXXXX
XXX+..      (3,3) (3,4) (3,5) (4,0) (4,1) (4,2) (4,3) (4,4) (4,5)
..XOX.      (3,4) (3,5) (4,0) (4,1) (4,3) (4,5)


XXXXXX      Set to scan first E and second S neighbours of cell
X++++X      if they are in list, pass them to the front of boardList, E first
XXXXXX
XXX+1.      (3,4) (3,5) (4,0) (4,1) (4,3) (4,5)
..X2X.      (3,4) (4,3) (3,5) (4,0) (4,1) (4,5)

next iteration of REPEAT:

XXXXXX      Try E neighbour, always first in boardList. Remove it from list.
X++++X      No ship cell spotted this time, move to next iteration
XXXXXX
XXX+X.      (3,4) (4,3) (3,5) (4,0) (4,1) (4,5)
..XOX.      (4,3) (3,5) (4,0) (4,1) (4,5)

next iteration of REPEAT:

XXXXXX      Now try S neighbour, this time first in boardList, remove it. Ship
X++++X      cell spotted, remove diagonal neighbours, however none is present
XXXXXX
XXX+X.      (4,3) (3,5) (4,0) (4,1) (4,5)
..X+X.      (3,5) (4,0) (4,1) (4,5)
```

had

```
XXXXXX      Set to scan first E and second S neighbours of cell
X+++XX      However E (3,5) was already scanned, and S is outside board,
XXXXXX      none is in boardList, so there are no modifications to list
XXX+X.
..X+X.      (3,5) (4,0) (4,1) (4,5)
```

Repeat ends (all the ship cells were spotted)

This algorithm still scans from left to right. A random scanning fashion might have more success for some boards, however, to avoid possible differences of implementations of Random class in different versions of java, that will lead to different solutions, **for submitting to Mooshak the random scanning was dropped.**

Just for sake of curiosity, to transform this in a random search algorithm, since the order of scanning is given by the ordered list of positions **boardList**, and extra step to randomly shuffle the **boardList** after being created, must be added. This will ensure that any position will be at most scanned once but in a random fashion.

```
Init boardList with all positions in board sorted by rows, left to right
shuffle boardList
(...)
```

Also since now it is not known if the N and W neighbours were already scanned, step:

```
move E and S neighbours to front of boardList, if in boardList
```

must be:

```
move N, E, S and W neighbours to front of boardList, if in boardList
```

had

Appendix C: Probabilistic solver algorithm example

```
solveProbabilistic(board)
   Init LikeliHoodMatrix
   Init empty highProbabilityList
   Init empty scanList

   count = number of ship cells to spot in board
   While count > 0
      if highProbabilityList NOT empty
          pos = remove first position from highProbabilityList
       else
          compute LikeliHoodMatrix
          pos = first high probability position from LikeliHoodMatrix

       add pos to scanList
       set pos as scanned in LikeliHoodMatrix
       if pos == ship cell
          count = count -1

          add diagonal neighbours to scanList
          set diagonal neighbours as scanned in LikeliHoodMatrix
          remove diagonal neighbours from highProbabilityList

          add neighbours to highProbabilityList (sorted as N, E, S, W)
```

Below is a full example of the algorithm solving a given board. When in the LikeliHood matrix column is specified *"just added -1"*, the matrix was **NOT** computed. It was just added impossible likelihood (-1) at the positions already scanned, or not needed to be scanned (the step in red in the algorithm above). **In these cases the next position is the first in the highProbabilityList**.

Board	LikeliHoodMatrix	highProbabilityList	next Position
...... .OOO.. ....O.. ...O..	10 14 16 16 14 10 13 17 19 19 17 13 14 18 **20** 20 18 14 13 17 19 19 17 13 10 14 16 16 14 10		**(2,2)**
...... .OOO.. ..**X**... ...O.. ...O..	10 14 13 16 14 10 13 17 13 **19** 17 13 11 11 **-1** 12 13 12 13 17 13 19 17 13 10 14 13 16 14 10		**(1,3)**
..**X**1**X**. .O4+2. ..**X**3**X**. ...O.. ...O..	10 14 **-1** 16 **-1** 10 13 17 13 -1 17 13 11 11 -1 12 **-1** 12 13 17 13 19 17 13 10 14 13 16 14 10 *just added -1*	**(0,3)(1,4)(2,3)(1,2)**	**(0,3)**
..X**X**X. .O4+2. ..X3X. ...O.. ...O..	10 14 -1 **-1** -1 10 13 17 13 -1 17 13 11 11 -1 12 -1 12 13 17 13 19 17 13 10 14 13 16 14 10 *just added -1*	**(1,4)(2,3)(1,2)**	**(1,4)**

..XXX. .O4+X. ..X3X. ...O.. ...O..	10 14 -1 -1 -1 10 13 17 13 -1 **-1** 13 11 11 -1 12 -1 12 13 17 13 19 17 13 10 14 13 16 14 10 *just added -1*	**(2,3)(1,2)**	**(2,3)**
..XXX. .O4+X. ..XXX. ...O.. ...O..	10 14 -1 -1 -1 10 13 17 13 -1 -1 13 11 11 -1 **-1** -1 12 13 17 13 19 17 13 10 14 13 16 14 10 *just added -1*	**(1,2)**	**(1,2)**
.XXXX. .1+++X. .XXXX. ...O.. ...O..	10 **-1** -1 -1 -1 10 13 17 **-1** -1 -1 13 11 **-1** -1 -1 -1 12 13 17 13 19 17 13 10 14 13 16 14 10 *just added -1*	**(1,1)**	**(1,1)**
XXXXX. 1++++X. XXXXX. ...O.. ...O..	**-1** -1 -1 -1 -1 10 13 **-1** -1 -1 -1 13 **-1** -1 -1 -1 -1 12 13 17 13 19 17 13 10 14 13 16 14 10 *just added -1*	**(1,0)**	**(1,0)**
XXXXX. X+++X. XXXXX. ...O.. ...O..	-1 -1 -1 -1 -1 6 **-1** -1 -1 -1 -1 9 -1 -1 -1 -1 -1 10 7 11 **13** 13 11 13 7 11 13 13 11 9		**(3,2)**
XXXXX. X+++X. XXXXX. ..XO.. ...O..	-1 -1 -1 -1 -1 6 -1 -1 -1 -1 -1 9 -1 -1 -1 -1 -1 10 4 4 -1 5 6 11 7 11 12 **13** 11 10		**(4,3)**
XXXXX. X+++X. XXXXX. ..X1X. ..3+2.		**(3,3)(4,4)(4,2)**	**(3,3)**
XXXXX. X+++X. XXXXX. ..X+X. ..3+2.		**(4,4)(4,2)**	**END** **count = 0**