Assignment 3: Automata

2IP90, Programming - Q1 (2023)

1 | Automata

This assignment consists of two parts. For part A, you will earn at most 7.5 points. If you want a higher grade than that, do also part B, which will earn you up to 2.5 points. We start with a short introduction about cellular automata, but first set up VSCode for this assignment:

Startup

1. Create a new directory `automata` and open it in VSCode.
2. Download files
   - `ABAutomaton.java`
   - `ABAutomatonTest.java`
   - `UniversalAutomaton.java`, and
   - `UniversalAutomatonTest.java`

   Put them in this folder.
3. Open all files, and fill in your names and student IDs in all four files.
4. Enable unit tests in VSCode:
   [a] Open the Testing pane, and click the button Enable Java Tests.
   [b] From the drop-down menu that appears at the top of your VSCode window, select option JUnit.

When you're finished with the assignment, submit all four files to Canvas.

Introduction

A cellular automaton consists of a row of cells. A cell is either empty or occupied. We call such a row of cells a generation. According to a set of rules, you can generate a new generation based on the current generation. These rules are the same for each cell in the generation.
The occupancy of a cell in the next generation is determined by its own occupancy in the current generation and that of its direct left-hand and right-hand neighbors. The first and last cells of a row only have one neighbor, but by considering their missing neighbors as empty cells, we can apply the rules anyway.
We define a single rule with the following notation:

```
current: ( )[*](*)
next   :    [*]
```

On the first line, we denote the situation of a cell in the current generation by describing its neighbors with ( ) and the cell with [ ]. A * in a cell means that this cell is occupied, and an empty space that it is empty. In the example above, the cell we're looking at is occupied, as is its right-hand neighbor. Its left-hand neighbor is empty.
The second line denotes the cell's occupancy in the next generation. In this example, the cell stays occupied in the next generation.
With this notation, we can define a complete rule set with eight rules:

```
- For occupied cells we have four possible configurations:
  current: ( )[*]( ) (*)[*]( ) ( )[*](*) (*)[*](*)
  next   :    [?]        [?]        [?]        [?]

- For empty cells we also have four possible configurations:
  current: ( )[ ]( ) (*)[ ]( ) ( )[ ](*) (*)[ ](*)
  next   :    [?]        [?]        [?]        [?]
```

Depending on the rule set, ? is either occupied (*) or empty.
Using this notation, we define the rule sets for automata A and B as follows:

```
- Automaton A:
  - Occupied cells remain occupied only if exactly one of the neighbors is occupied:
   current: ( )[*]( ) (*)[*]( ) ( )[*](*) (*)[*](*)
   next   :    [ ]        [*]        [*]        [ ]
  - Empty cells remain empty only if both neighbors are empty:
   current: ( )[ ]( ) (*)[ ]( ) ( )[ ](*) (*)[ ](*)
   next   :    [ ]        [*]        [*]        [*]

- Automaton B:
  - Occupied cells remain occupied only if the right-hand neighbor is empty:
   current: ( )[*]( ) (*)[*]( ) ( )[*](*) (*)[*](*)
   next   :    [*]        [*]        [ ]        [ ]
  - Empty cells become occupied if exactly one neighbor is occupied:
   current: ( )[ ]( ) (*)[ ]( ) ( )[ ](*) (*)[ ](*)
   next   :    [ ]        [*]        [*]        [ ]
```

Part A

For part A, work in files `ABAutomaton.java` and `ABAutomatonTest.java`. First, read the problem description. Then implement your program by following the steps in section Approach with unit testing.

Problem description

Write a program that implements automata A and B. The user picks and configures the automaton to use. Subsequently, following that configuration, the program prints a sequence of generations. The program works as follows:

Input

When the program starts, the user enters:

- Either “A” or “B” to select the automaton to run.
- A positive integer representing the number of cells in a generation, or the length of a generation (L). This number does not include extra empty border cells you might add in your implementation.
- A positive integer representing the number of generations that the program should display (G).

- The string `init_start` followed by one or more positive integers, followed by the string `init_end`. These integers designate the position of the cells in the initial generation. Note: The first cell in a generation has position 1. All cells that aren't specified are initially empty. When an integer is higher than the generation length L, that number should be ignored.

Notes

- Newlines aren't significant in the input. In other words, don't use Scanner's `nextLine` method to read input.
- Don't ask the user for input; expect the user to input without prompting for it. See the example runs below.

Output

A sequence of G lines representing the successive generations. The first line is the initial generation input by the user. Each line consists of L characters. Each character represents a single cell in the generation. Display an empty cell as a space, and an occupied cell as the character '*'.

Example runs

In the example runs below, we indicate user input with an ¿. This symbol is not actually part of the input; the user doesn't enter it. Nor is it part of the output; your program doesn't print it.
Example run with automaton A

```
> A 11 10
>init_start 6 init_end
     *
    * *
   * * *
  * * * *
 * * * * *
* * * * * *
 * * * * *
* * * * * *
 * * * * *
* * * * * *
```

Example run with automaton B

```
> B 61 20
> init_start 20 40 init_end
```

```
                   *                   *
                  ***                 ***
                 *  **               *  **
                **** **             **** **
               *   *  **           *   *  **
              *** **** **         *** **** **
             *  *    *  **       *  *    *  **
            ******  **** **     ******  **** **
           *     ***   *  **   *     ***   *  **
          ***   *  ** **** ** ***   *  ** **** **
         *  ** **** *    *  *   ** **** *    *  **
        **** *    * **  ****** * *    * **  **** **
       *   * **  **  ***     * * **  **  ***   *  **
      *** **  *** ***  **   ** *  *** ***  ** **** **
     *  *  ***  *   *** ** * * ***  *   *** *    *  **
    *******  ***** *  *  * * *   ***** *  * **  **** **
   *      ***    * ******* * ** *    * ****  ***   *  **
  ***    *  **  **       * *  * **  **    ***  ** **** **
 *  **  **** *** **     ** ****  *** **  *  *** *    *  **
**** ***   *   *  **   * *    ***  *  ******  * **  **** **
```

Approach with unit testing

In the file `ABAutomaton.java`, we've already implemented a `run` method:

Java Code: `run()` Method

```java
void run() {
    // Read input to configure the automaton
    String automaton = scanner.next();
    int genLength = scanner.nextInt();
    int numOfGens = scanner.nextInt();
    boolean[] initGen = readInitalGeneration(genLength);

    // Run the automaton
    boolean[] gen = initGen;

    for (int i = 0; i < numOfGens; i++) {
        // Display the current generation
        System.out.println(genToString(gen));

        // And determine the next generation
        if ("A".equals(automaton)) {
            gen = nextGenA(gen);
        } else {
            // B
            gen = nextGenB(gen);
        }
```

Don't change this method. Instead, implement and unit test the methods that are called in the `run` method.

Implement and test `genToString`

5. In `ABAutomaton.java`, document and implement the `genToString` method.

6. In `ABAutomatonTest.java`, write a minimal test set for the `genToString` method. Add at least five test cases to the test method `testGenToString`. Motivate each test case with a single-line comment.

Make sure all tests pass.

Implement and test `nextGenA`

7. In `ABAutomaton.java`, document and implement the `nextGenA` method.

8. In `ABAutomatonTest.java`, write a minimal test set for the `nextGenA` method. Add at least eight test cases to the test method `testNextGenA`. Motivate each test case with a single-line comment.

Hints

- Why do we ask you to write at least eight test cases?
- You can use the testing method `assertArrayEquals` to test if an array matches an expected array. It works the same as the `assertEquals` method but works on arrays rather than singular values like numbers or strings.

Make sure all tests pass.

Implement and test `nextGenB`

9. In `ABAutomaton.java`, document and implement the `nextGenB` method.

10. In `ABAutomatonTest.java`, write a minimal test set for the `nextGenB` method. Add at least eight test cases to the test method `testNextGenB`. Motivate each test case with a single-line comment.

Make sure all tests pass.

Implement `readInitialGeneration` and test your whole program

11. In `ABAutomaton.java`, document and implement the `readInitialGeneration` method.

Because the method `readInitialGeneration` reads from input, we say it has side effects. We don't unit test methods with side effects. However, you will test these methods indirectly by testing your whole program.

12. Test your whole program. Make sure that your program behaves like the running examples. But ask yourself: Does only testing these running examples tests your program completely? Make sure your program works as specified in this assignment description.

Part B

For part B, work in files `UniversalAutomaton.java` and `UniversalAutomatonTest.java`.

Problem description

Write a program that implements a so-called universal automaton. The input/output behavior of this program is the same as the program you wrote in part A except for the automaton selection. Instead of entering "A" or "B" to select the automaton, the user enters a rule sequence.
The rules that define an automaton are described in the following compact format:
Rule set for the universal automaton
Remember that the occupancy of a cell in the next generation depends only on its occupancy in the current generation and that of its left-hand and right-hand neighbors. As you have seen, there are eight different combinations of occupancy for three cells possible. We call such combinations a neighborhood pattern. By representing each cell by a 0 if it is empty and a 1 if it is occupied, we can describe each of these eight different neighborhood patterns with a three-bit number as follows:

```
pattern number  bit pattern  neighborhood pattern
0               000          ( )[ ]( )
1               001          ( )[ ](*)
2               010          ( )[*]( )
3               011          ( )[*](*)
4               100          (*)[ ]( )
5               101          (*)[ ](*)
6               110          (*)[*]( )
7               111          (*)[*](*)
```

If we know the occupancy of the cell in the next generation for each of the eight neighborhood patterns, we have completely defined the behavior of the automaton. To specify the automaton, let the user input eight numbers, either a 0 or a 1, that define the automaton's rule sequence:
- The first number specifies the behavior for the neighborhood pattern ( )[ ]( ) - The second number specifies the behavior for the neighborhood pattern ( )[ ](*) - Etc.
So, what the automaton should do for each cell is the following. If the neighborhood pattern of a cell in the current generation is found at number n, 0 ¡= n ¡= 7, that cell is empty in the next generation if the nth number is 0, and occupied if the nth number is 1.
Note: Numbering of the rule sequence starts at 0.

Example runs

In the example runs below, we indicate user input with an ¿. This symbol is not actually part of the input; the user doesn't enter it. Nor is it part of the output; your program doesn't print it.
Example run with automaton U
In this example run, the rule sequence is 0 1 0 1 1 1 1 0. The automaton will thus behave as follows: - An empty cell with empty neighbors stays empty in the next generation, since the neighborhood pattern 000 is the first entry in the table above, and 0 is the first number in the input rule sequence. - An empty cell with an empty left neighbor and an occupied right neighbor will be occupied in the next generation because 001 is the second entry in the table above, and 1 is the second number in the input rule sequence. - Etc.

```
> 0 1 0 1 1 1 1 0
> 11 5
> init_start 5 init_end
    *
   * *
  * * *
 * * * *
* * * * *
```

Approach with unit testing

In the file `UniversalAutomaton.java`, we've already implemented a `run` method:

```
void run() {
    // Read input to configure the universal automaton
    boolean[] ruleSequence = readRuleSequence();
    int generationLength = scanner.nextInt();
    int numberOfGenerations = scanner.nextInt();
    boolean[] initGen = readInitalGeneration(generationLength);

    // Run the automaton
    boolean[] gen = initGen;

    for (int i = 0; i < numberOfGenerations; i++) {
        // Display the current generation
        System.out.println(genToString(gen));
        // Determine the next generation
        gen = nextGen(ruleSequence, gen);
    }
}
```

Don't change this method. Instead, implement and unit test the methods that are called in the `run` method.

Copy `genToString`

13. Displaying a generation hasn't changed for the universal automaton: Copy the contents of your `genToString` method from `ABAutomaton.java` to `UniversalAutomaton.java`.

We also recommend copying your tests for `genToString` from `ABAutomatonTest.java` to `UniversalAutomatonTest.java` to make sure it still works as expected.

Implement and test `nextGen`

14. In `UniversalAutomaton.java`, document and implement the `nextGen` method.

Hints

- Note that we represent the rule sequence by a boolean array. Interpret `false` as zero (0) and `true` as one (1).
- To see which pattern applies to a certain cell, you could use a sequence of eight if statements.

However, observe that the patterns are ordered as if they're binary numbers, from small to large. You can make use of this observation and get a much shorter piece of code than using eight ifs.

15. In `UniversalAutomatonTest.java`, write a minimal test set for the **`nextGen`** method. Motivate each test case with a single-line comment.

Make sure all tests pass.

Copy `readInitialGeneration`

16. Reading the initial generation hasn't changed for the universal automaton: Copy the contents of your `readInitialGeneration` method from `ABAutomaton.java` to `UniversalAutomaton.java`.

Implement `readRuleSequence` and test your whole program

17. In `UniversalAutomaton.java`, document and implement the `readRuleSequence` method. This method has side effects, so you don't unit test it.

18. Test your whole program. Make sure that your program behaves like the running examples, including the ones for automata A and B. But ask yourself: Does only testing these running examples tests your program completely?

Make sure your program works as specified in this assignment description.