KAZAKH BRITISH TECHNICAL UNIVERSITY

Basics of Computer Simulation

Lab 1

“Conway’s Game of Life”

Prepared by
Mukhanov Almaz

Almaty, 2019

Introduction

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbors, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if caused by under-population.
2. Any live cell with two or three live neighbors become alive on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed—births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.

Method

In this lab I provide my implementation of Conway's Game of Life using the Java programming language and the Eclipse IDE.

Grid of size 10x10 is initialized with 0's as the dead and 1's as alive cells.

The nextGen() function loops through every cell and counts it's alive neighbors. The function gets a *field* matrix as input, processes it and outputs certain amount of *next* generations, defined by *cnt*, each output matrix acts like input for further iterations. My implementation is supposed to be played on finite plane (not toroidal version), so I don't count boundary cells' external neighbors. Cell's own value is subtracted from its neighbor number. Based on *aliveNear* value, the aforementioned rules are implemented. Thread.sleep() is used in order to display the output correctly.

```java
 static void nextGen(int field[][], int x) throws InterruptedException {

     int cnt = 10;

     while (cnt != 0) {
         int[][] next = new int[x][x];
         for (int i = 0; i < x; i++) {
             for (int j = 0; j < x; j++) {
                 int aliveNear = 0;
                 for (int m = -1; m <= 1; m++) {
                     for (int n = -1; n <= 1; n++) {
                         if ((i + m < 0) ||                  // upper border
                             (i + m > x - 1) ||              // lower
                             (j + n < 0) ||                  // left
                             (j + n > x - 1))                // right
                                 continue;
                         aliveNear += field[i + m][j + n];
                     }
                 }
                 aliveNear -= field[i][j];

                 if ((field[i][j] == 1) && (aliveNear < 2))
                     next[i][j] = 0;

                 else if ((field[i][j] == 1) && (aliveNear > 3))
                     next[i][j] = 0;

                 else if ((field[i][j] == 0) && (aliveNear == 3))
                     next[i][j] = 1;

                 else
                     next[i][j] = field[i][j];
             }
         }
         System.out.println("Next Generation");
         printField(next);
         System.out.println();
         field = next;
         Thread.sleep(500);
         cnt--;
     }
 }

static void printField(int field[][]) {
    for (int i = 0; i < field.length; i++) {
        for (int j = 0; j < field[0].length; j++) {
            System.out.print((field[i][j] == 0) ? "⬡ " : "⬢ ");
        }
        System.out.println();
    }
}
```

```java
public static void main(String[] args) throws InterruptedException {

    int[][] field = {
            { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 },
            { 0, 0, 0, 0, 0, 1, 1, 0, 0, 0 },
            { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 },
            { 0, 0, 0, 1, 0, 0, 0, 1, 1, 0 },
            { 0, 0, 0, 1, 0, 0, 1, 0, 0, 1 },
            { 0, 0, 0, 0, 0, 0, 0, 1, 1, 0 } };

    System.out.println("Start Generation");
    printField(field);
    System.out.println();
    nextGen(field, field.length);
}
```

Results

Output (part) of the following example is:

```
Start Generation
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡

Next Generation
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬢ ⬢ ⬢ ⬡ ⬡ ⬢ ⬢ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡
```

```
Next Generation
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡

Next Generation
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬢ ⬢ ⬢ ⬡ ⬡ ⬢ ⬢ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡

Next Generation
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡
⬡ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢ ⬡ ⬡ ⬢
⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬡ ⬢ ⬢ ⬡
```