

Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS231-B

February 12, 2023

Cellular Automation: A Java Simulation

- **Abstract**

This second project is a simulation of Conway's Game of Life, a classic example of cellular automata. The purpose of the assignment is to implement a data structure to represent the grid of cells and the rules of the game, as well as to create a graphical user interface to visualize and interact with the simulation. In this project, we will be developing a simulation using Java Swing graphical user interface package. Here, we will create three classes: Cell, Landscape, and LifeSimulation. The Cell class represents a location on the Landscape, and it stores whether it is alive and has methods to get and set its state. Next, the Landscape class represents a 2D grid of Cells using out arraylist concept, which has methods to allocate and reset the grid, get the number of rows and columns, get a Cell reference at a specific location, and get the neighbors of a Cell. One of a bigtake way from the project is to learn the ArrayList class concept that is a resizable array and its the difference between a built-in array and an ArrayList in Java (the size of an array cannot be modified). We will also create a visualization of the Landscape using Java's Graphics class and add a method to update the state of the Cell and create a LifeSimulation class that runs the simulation loop and animates the game concisely as planned. Thus, over the process, we will learn how a cell represents a unit of life that can either be alive or dead and how cell interacts with its neighboring cells, and the position of a cell in the grid is crucial to its behavior, as it affects which neighboring cells it interacts with.

Results

The Cell Class

We are developing the Cell class as part of a larger project to simulate Conway's Game of Life, which is a classic example of a cellular automaton. The Cell class represents an individual cell in the game and will be used to construct the Landscape and ultimately the LifeSimulation.

Khan 2

Here, the Cell class stores information about whether the cell is alive or dead and provides methods to access and modify this information. The constructor methods allow us to create Cell objects with an initial state, and the `getAlive ()` and `setAlive ()` methods provide a way to access and modify the state of a Cell object. The `toString ()` method is used to represent the state of the Cell as a string, which will be used in the final display of the simulation.

The Cell class represents a single cell in the grid. It has two constructors, one that creates a dead cell and one that creates a cell with the specified initial status. It also has methods to get and set the current status of the cell, as well as a `toString()` method that returns "1" if the cell is alive and "0" if it is dead.

UpdateState (): In addition, the `updateState()` method is an extension of the Cell class, which updates the state of the cell based on the state of its neighboring cells. This method takes an ArrayList of neighboring cells as input and counts the number of alive neighbors. If the cell is alive, it stays alive only if it has 2 or 3 alive neighbors, otherwise, it dies. If the cell is dead, it comes back to life only if it has exactly 3 alive neighbors. This method is essential to simulate the evolution of the cells in the grid.

Last but not the least, the `main()` method creates a 3x3 grid of cells, sets some cells as alive, updates the state of each cell based on its neighbors, and prints the updated grid. The neighboring cells of each cell are added to an ArrayList, and the `updateState()` method is called on each cell with its respective list of neighbors. Let's run and see what it gives.

A 3x3 grid of cells, all initially dead (000). The grid is displayed as a 3x3 array of strings, each string being "000".

```
000
000
000
```

Figure 1: Running Cell.java for creating a 3*3 grid of cells.

Why did we get this kind of output? Because the output represents the initial state of a 3x3 grid of cells, with all cells initially dead. The `main()` method of the code initializes a 3x3 grid of cells and sets all of them to dead using the default constructor, which sets the alive instance variable to false. The code then prints the state of each cell in the grid using the `toString()`

method, which returns "0" for a dead cell. Therefore, the output is a 3x3 grid of "0" characters, indicating that all cells in the grid are dead.

On the other, I tried another main method, which serves as a simple demonstration of how to create instances of the Cell class and how to get and set the status of the cells. It consists of several lines of code that create instances of the Cell class, set their status using the `setAlive()` method, and print their status using the `getAlive()` method and the `toString()` method.

The first line of the `main()` method creates an instance of the Cell class using the default constructor, which creates a dead cell. Then, the `getAlive()` method is then called on this instance, which returns false to indicate that the cell is dead. There, `toString()` method is also called on the same instance, which returns "0" to represent a dead cell.

The next line creates an instance of the Cell class using the constructor that takes a boolean argument, which creates a live cell. The `getAlive()` method is then called on this instance, which returns true to indicate that the cell is alive. The `toString()` method is also called on this instance, which returns "1" to represent a live cell. The next line calls the `setAlive()` method on the first Cell instance to set its status to true, or alive. The `getAlive()` method is then called on this instance, which returns true to indicate that the cell is alive. The `toString()` method is also called on this instance, which returns "1" to represent a live cell. So, let's see what it gives.

```
false
0
true
1
true
1
```

Figure 2: Running Cell.java for creating a 3*3 grid of cells.

The first Cell instance `c1` is created using the default constructor, which sets its status to false or dead. The `getAlive()` method returns false, and the `toString()` method returns "0", which represents a dead cell. The second Cell instance `c2` is created using the constructor that takes a boolean argument, which sets its status to true or alive. The `getAlive()` method returns true, and

the `toString()` method returns "1", which represents a live cell. The `setAlive()` method is then called on `c1` to set its status to true or alive. The `getAlive()` method returns true, and the `toString()` method returns "1", indicating that `c1` is now a live cell.

So, what was the purpose of this class? By developing the Cell class, what we did was to encapsulate the behavior and data of an individual cell, which makes it easier to reason about and manipulate individual cells within the larger simulation.

The cellTests:

The purpose of this is to test the Cell class, which is a fundamental building block for the Game of Life simulation. The Cell class represents a single cell in a grid and is responsible for keeping track of whether it is alive.

So, the CellTests class contains thirteen test cases which test various aspects of the Cell class. The first three cases test the constructor and `getAlive()` and `setAlive()` methods of the Cell class. The remaining ten cases test the `updateState()` method, which determines whether the cell should be alive or dead in the next generation of the simulation based on the status of its neighbors. Each test case prints out a verification statement which compares the expected output to the actual output of the method being tested. Additionally, each test case uses an assertion to automatically verify the correctness of the output. Here, one thing should be noted that I ran using the command `java -ea CellTests`, so that it enables Java assertions. If all assertions pass, the output "Done testing Cell!" is printed.

```

0 == 0
1 == 1
0 == 0
false == false
true == true
false == false
true == true
false == false
true == true
false == false
false == false
false == true
false == true
false == false
false == false
false == false
false == false
false == true
false == false
*** Done testing Cell! ***

```

Figure 3: Running the cellTests.

The output of the code indicates the results of the unit tests for the Cell class. There are 13 test cases in total, which cover the constructor, `getAlive()`, `setAlive()`, and `updateState()` methods of the Cell class. Each test case prints out an expected value and an actual value. If the expected and actual values are the same, then the test passes. If they are different, then an error message is printed. For example, what we can understand from the result is, in the first test case, the expected value is 0 and the actual value is the result of calling `toString()` on a new Cell object and since `toString()` returns "0" for a dead cell, the expected and actual values are the same, and the test passes.

The Landscape Class

Here, Landscape class represents a two-dimensional grid of Cell objects. The Landscape class has two constructors. The first constructor takes the number of rows and columns as arguments and creates a landscape of dead cells. The second constructor takes the number of

rows, columns, and an initial chance of a cell being alive. It creates a landscape of cells where each cell has an initial chance of being alive.

Moreover, the Landscape class provides a `reset()` method that randomly sets the status of each cell based on the initial chance. The `toString()` method returns a string representation of the landscape, with each cell represented by a "0" or "1" character. Furthermore, the `getCell()` method returns the Cell object at the specified row and column in the landscape. The `getNeighbors()` method takes the row and column of a cell and returns an ArrayList of neighboring cells.

Advance() Method:

The `advance()` method is the main method for advancing the state of the landscape. It creates a new temporary grid to hold the new generation of cells. It then loops through the current grid, gets the neighbors of each cell, and updates the status of the corresponding cell in the temporary grid. Once all the cells have been updated, the temporary grid is set as the new state of the landscape. We also used the `draw()` method that draws the landscape on a Graphics object with a specified scale.

Finally, the `main()` method creates a new Landscape object, prints its initial state using the `toString()` method, advances the landscape by one generation using the `advance()` method, and prints the new state of the landscape using the `toString()` method.

```
0011011101
1000101100
0111111101
1010001110
0101111111
0101010111
1000100011
1001010111
1111101110
0011110100

0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
0000000000
```

Figure 4.1: Running Landscape class using run icon.

One thing I was very confused about what how the outout came like all zeros there. I know that the output shows the initial state of the grid and the grid after one generation has passed. The numbers represent the status of each cell, with 1 indicating a live cell and 0 indicating a dead cell. But why all zeros? So, I ran it again in the terminal using command line “java Landscape.”

```

1110110010
0011100000
0010101111
0110010100
0001000100
0110111101
0001111011
1000101110
1111010001
0011011011

0110110000
0000001001
0000101110
0110110000
0001000100
0010000001
0110000001
1000000000
1000000001
0001011011

```

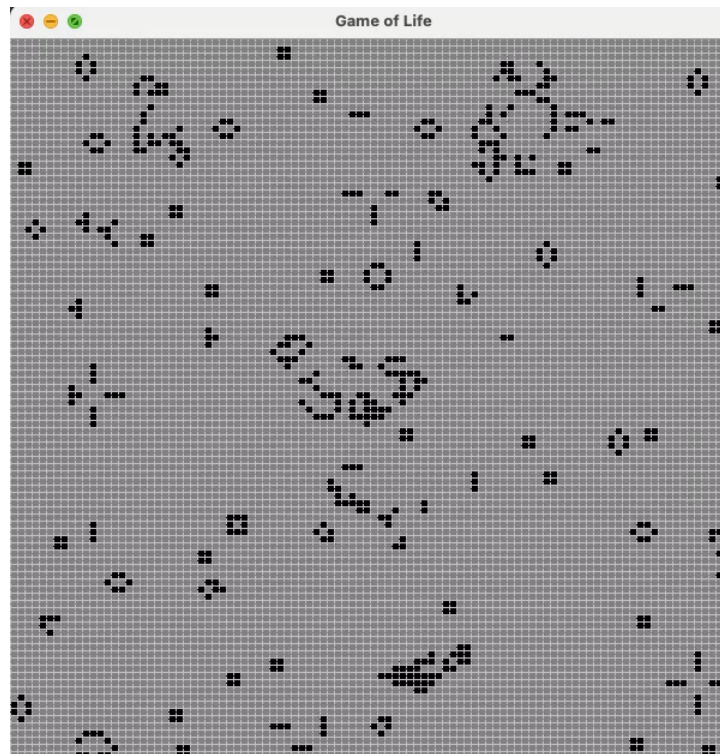
Figure 4.2: Running Landscape class using command line.

Damn! The output changed mysteriously! Let's try it to figure that out. First, the output shows the state of the automaton after two iterations. Let's try to examine the output again! First, the initial state of the automaton is randomly generated using the constructor that takes a probability of a cell being alive. In this case, the probability is set to 0.5, so roughly half of the cells are alive and half are dead. Side by side, the toString() method prints out the state of the automaton as a string of 1s and 0s, where 1 represents a live cell and 0 represents a dead cell.

More talk about Advance():

For more context, let's remind ourselves what we did here. The advance() method updates the state of the automaton according to the rules of Conway's Game of Life and that means each cell's new state depends on the states of its eight neighbors. If a cell is alive and has two or three live neighbors, it remains alive. Otherwise, it dies. If a cell is dead and has exactly three live

neighbors, it becomes alive. Otherwise, it remains dead. So, the output after the first call to `advance()` shows the state of the automaton after one iteration. Some cells have changed state, but the overall pattern is still recognizable. The output after the second call to `advance()` shows the state of the automaton after two iterations. The pattern has changed more dramatically, with some cells dying and others coming to life. So, we can say that the automaton will continue to evolve with each call to `advance()`.



Video 1: Running LandscapeDisplay class.

So, what we understand from the output? We used this program to create a graphical user interface to display and update a 2D grid of cells using the Swing library. But, one thing I did not understand, over all the time, I code, my awt package showed that to be error, but my classes still worked instead. How? I will ask the professor about that for sure, when I will come to the USA next week.

Let's come back to the topic. The program consists of two classes: Landscape and LandscapeDisplay. First, the Landscape class represents a grid of cells, where each cell is either alive or dead. The class provides methods to manipulate the grid, including resetting it to a

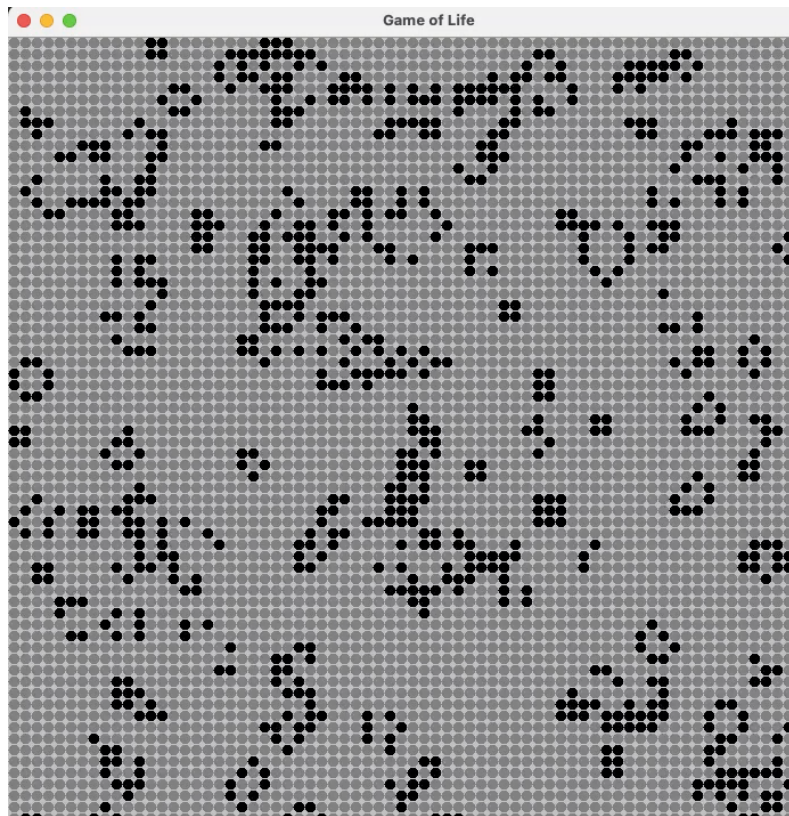
random state, advancing the grid to the next generation, getting the state of a cell at a given position, and drawing the grid using graphics. Then, the LandscapeDisplay class creates a graphical window to display the Landscape. And the class creates a JPanel to draw the Landscape and adds it to a JFrame. The class also provides a method to save an image of the display and a method to repaint the display. If we look at the main method we created, a LandscapeDisplay object and continuously calls the advance method of the Landscape object to update the grid and repaint the display.

In other words, to use the program, we can create a Landscape object and pass it to a new LandscapeDisplay object. The display window will show the current state of the grid, and it will update as the grid advances through the generations. The user can also save an image of the display by calling the saveImage method. The program runs indefinitely until the user closes the window.

The LifeSimulation Class:

The LifeSimulation class is a Java program that simulates a game of life with a 2D grid of cells. The program takes three command-line arguments: the size of the grid, the initial density of living cells, and the number of time steps to simulate.

First, the main method initializes a Landscape object with the given size and density and creates a LandscapeDisplay object to visualize the grid. Second, the program then runs a loop for the specified number of time steps, where it advances the grid by one generation, repaints the display, and sleeps for a short time to allow the display to update. The program can be compiled and run from the command line using the javac and java commands. For example, to run the simulation with a grid size of 20, an initial density of 0.3-, and 50-times steps, the following command can be used: "java LifeSimulation 20 0.3 50". Let's try running using 70*70 grid!!



Video 2: Running LifeSimulation class (70*70)

The program provides a basic implementation of the game of life, but it can be extended to perform further analysis and experimentation, as shown in the commented-out code block in the original main method. For example, the program can be modified to test different initial density values and compare the results to see if there is a correlation between the initial density and the number of living cells after a sufficient number of rounds. Additionally, the program can be modified to perform statistical analysis on the results to determine the significance of the correlation.

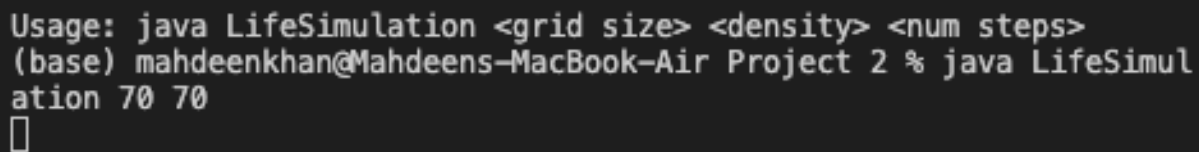
Extension:

Modify the main method in LifeSimulation:

The LifeSimulation class contains a main method to run simulations of Conway's Game of Life. We know that the original code includes code to take in command-line arguments for the

grid size and number of steps to simulate and creates a Landscape with random living cells based on a given probability. The code then animates the simulation using a LandscapeDisplay object.

An extension to the original code we provided, which adds the ability to also pass in the initial density of living cells as a command-line argument. This allows for more fine-grained control over the simulation, as users can specify the initial density of living cells as well as the grid size and number of steps to simulate.



```
Usage: java LifeSimulation <grid size> <density> <num steps>
(base) mahdeenkhan@Mahdeens-MacBook-Air Project 2 % java LifeSimulation 70 70
█
```

Figure 5: Running LifeSimulation class after adding extension.

Modifying the updateState() function in the Cell class to implement different rules:

Now, let's go back to our Cell class. The original updateState() function in the Cell class implements the standard rules of the Game of Life, where a live cell survives if it has 2 or 3 live neighbors, and a dead cell comes to life if it has exactly 3 live neighbors. So, to implement different rules, we modified the updateState() function to check for different conditions before updating the state of the cell. For example, as we wanted to implement rules where a cell survives if it has 1, 2, or 4 live neighbors and a dead cell comes to life if it has exactly 2 or 5 live neighbors, we modify the function as follows:

The first part of the code creates a 3x3 grid of Cell objects and sets some of them as alive or dead. The second part of the code updates the state of each cell by checking its neighboring cells using the updateState() method. In the updated updateState() method, the number of alive neighbors is counted by iterating over the provided list of neighboring cells, and if the current cell is alive, it is not counted as an alive neighbor. Then, based on the number of alive neighbors, the state of the current cell is updated using the rules provided in the method.

```

000
000
000
false
0
true
1
true
1

```

Figure 6: Running Cell Class after modifying the updateState().

In the first example, all cells are initialized as dead, so all outputs are 0 or false. In the second example, a single cell is initialized as alive and the other cells are dead. The output of the `getAlive()` method for the alive cell is true, and its String representation is 1. The state of the first cell is then updated to alive, and the output of its `getAlive()` method and String representation are both true and 1, respectively.

Create more than one type of Cell and give each type different rules:

Now our target is to simulate a game of life by creating a 10x10 grid of cells, with some of the cells being normal and some being immortal. So, first, the initial state of each cell is determined randomly. We make the program then updates the state of each cell based on the rules of Conway's Game of Life. For each cell, the program determines the states of its eight neighbors and updates the cell's state according to those states. If a cell is alive and has either two or three living neighbors, it remains alive; otherwise, it dies. If a cell is dead and has exactly three living neighbors, it comes to life.

After updating the state of each cell, the program prints the new state of the grid to the console. The grid consists of 0's for normal cells and I's for immortal cells. The program checks if a cell is an instance of `ImmortalCell` or not and if it is, prints I for that cell, otherwise prints the current state of the cell. The `ImmortalCell` class extends the `Cell` class and overrides the `updateState()` method to do nothing, ensuring that an `ImmortalCell` remains alive regardless of the state of its neighbors.

```

001I00I100
I01IIII101
II0I00I0II
0III0II0I0
1111I00011
I1I00I0I01
000IIII110
0I0101III0
II0010I00I
0IIIIIIIII

```

Figure 7: Running newCell class.

The output represents a 10x10 grid of cells in which each cell can be either a NormalCell or an ImmortalCell, represented respectively by "0" and "I". The initial state of the grid was randomly filled with a mix of NormalCells and ImmortalCells, with about a 50/50 proportion of each cell type.

The grid was then updated using the updateState() method of each cell based on the states of its neighboring cells, following the rules of Conway's Game of Life. In this variant, ImmortalCells always remain alive, while NormalCells follow the standard Game of Life rules. The resulting grid is printed to the console, with NormalCells represented by "0" and ImmortalCells represented by "I". The pattern that emerges is the result of the evolution of the grid over multiple iterations of applying the updateState() method. The output cannot be fully interpreted without knowing how many iterations were applied to the grid and the initial state of the grid. However, it suggests that the evolution of the grid leads to a relatively stable pattern with some ImmortalCells and NormalCells arranged in clusters, separated by empty spaces.

- **References/Acknowledgements**

Working on a project is never a solo journey! It requires collaboration, communication, and teamwork to make something great. This week was mostly on the topic of ArrayList and I loved it. I had the pleasure of working with a fantastic group of people like Kashef on this project, including the amazing teaching assistants Nafis and instructor Professor Max Bender, who provided us with their expertise, guidance, and support. I also want to extend a special

thanks to my friend Banshee who have taken the course in a previous semester and shared their insights and experiences with me. His advice was invaluable, and I couldn't have done this without him. I learned so much from them and enjoyed every moment of our collaboration.

Sources, imported libraries, and collaborators are cited:

From my knowledge, there should not be any required sources or collaborators. But we can say, we got our instructions from CS231 Project Manual:

<https://cs.colby.edu/courses/S23/cs231B/projects/project1/project1.html>

Except this, while working on the classes, I tried to import “java.util.ArrayList”, “java.util.List”, “java.util.awt”, “java.util.Collections”, “java.util.Random.”