Conway's Game of Life

Muneeb Azfar Nafees

March 2025

Contents

1 Abstract

In this project, I simulated Conway's Game of Life using object-oriented programming in Java. I used arrays to create the grid that stores `Cell` objects, which are the basic units of life in the game. In addition, the `getNeighbors` method uses `ArrayList` to handle a variable number of neighboring cells dynamically, thus optimizing memory usage when compared to fixed-size arrays. I further extended my implementation by developing a graphical user interface using `JFrame` and `JPanel`. My interface prompts the user for simulation parameters and provides interactive controls for starting, pausing, and resetting the simulation.

My experiments reveal that grid dimensions and initial cell density critically influence the simulation's evolution. Moderate initial probabilities tend to yield stable patterns, while extreme values lead to quick extinction of life. Key findings also show that the proportion of surviving cells stabilizes over time, regardless of absolute grid size.

2 Results

2.1 Visualization of Simulation States

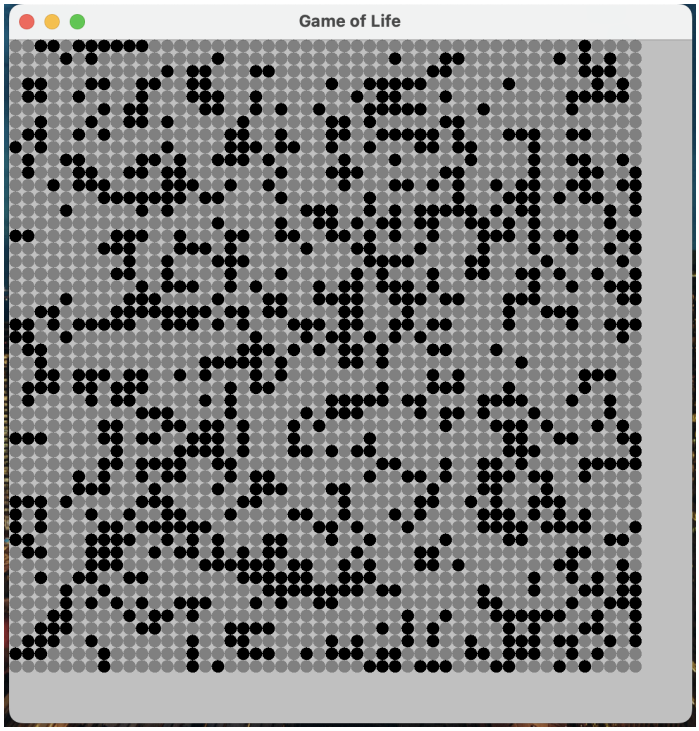


Figure 1: Initial state of the simulation with given conditions.

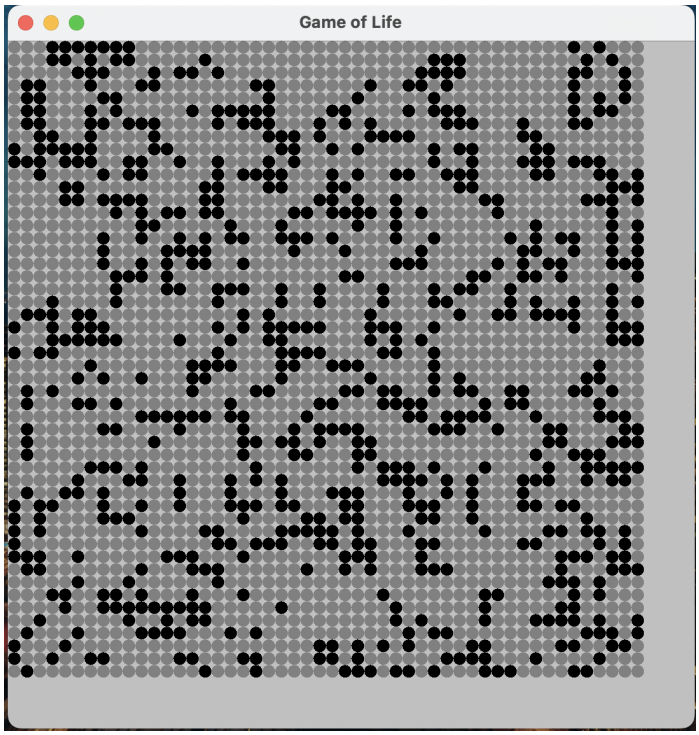


Figure 2: Board state after one simulation step (advance call) showing the updated cell states.

Figure 1 shows the initial state of the simulation board. In this image, the board is generated using following command:

```
java LifeSimulation 50 50 0.3 2
```

Figure 2 displays the board after one simulation time step (one call to `advance()`). This image shows how the board evolves from its initial configuration, confirming that the simulation updates cell states correctly.

2.2 Hypothesis

I hypothesize that:

1. For very low initial probabilities (near 0.0), the grid will quickly become extinct because too few cells start alive.
2. For very high initial probabilities (near 1.0), overcrowding will cause most cells to die off in the first few rounds, also leading to near extinction.
3. Moderate initial probabilities (roughly 0.2-0.6) will retain a significant number of living cells after 1000 rounds.

4. Increasing the grid size (with the same initial probability) will lead to a larger absolute number of survivors but a similar fraction of living cells relative to the total number of cells.

2.3 Experiments Conducted

To investigate the effect of grid dimensions and initial cell probability on the final number of living cells after 1000 simulation rounds, I conducted several experiments using the `LifeSimulation` code. The simulation was run using the following command-line format:

```
java LifeSimulation <rows> <cols> <chance> <iterations>
```

For example, one experiment was executed as follows:

```
java LifeSimulation 100 100 0.25 1000
```

In my study, I varied the initial probability from 0.0 to 1.0 on a 100×100 grid and also examined the effect of grid size at a fixed probability (0.3). This approach enabled me to observe how both parameters influence the evolution of the system over 1000 rounds.

2.4 Metrics and Experimental Setup

For each simulation run, I recorded the number of living cells (cells that are `true`) both before and after the simulation. The key metric is the final count of living cells after 1000 rounds, which provides insight into the system's working. The experiments included:

- **Varying Initial Probability on a 100×100 Grid:** The simulation was run with probabilities ranging from 0.0 to 1.0 (in increments of 0.1) using:

```
java LifeSimulation 100 100 <chance> 1000
```

- **Varying Grid Dimensions at a Fixed Probability:** Simulations were performed with grid sizes of 50×50, 100×100, 200×200, 400×400, and 800×800 using a fixed probability of 0.3:

```
java LifeSimulation 50 50 0.3 1000
java LifeSimulation 100 100 0.3 1000
java LifeSimulation 200 200 0.3 1000
java LifeSimulation 400 400 0.3 1000
java LifeSimulation 800 800 0.3 1000
```

2.5 Key Results

The table below summarizes the results for the 100×100 grid experiments across different initial probabilities:

Initial Probability	Alive Before Simulation	Alive After Simulation
0.0	0	0
0.1	1014	217
0.2	2024	316
0.3	3014	399
0.4	3987	415
0.5	4970	403
0.6	6025	422
0.7	6989	263
0.8	7907	70
0.9	8955	4
1.0	10000	0

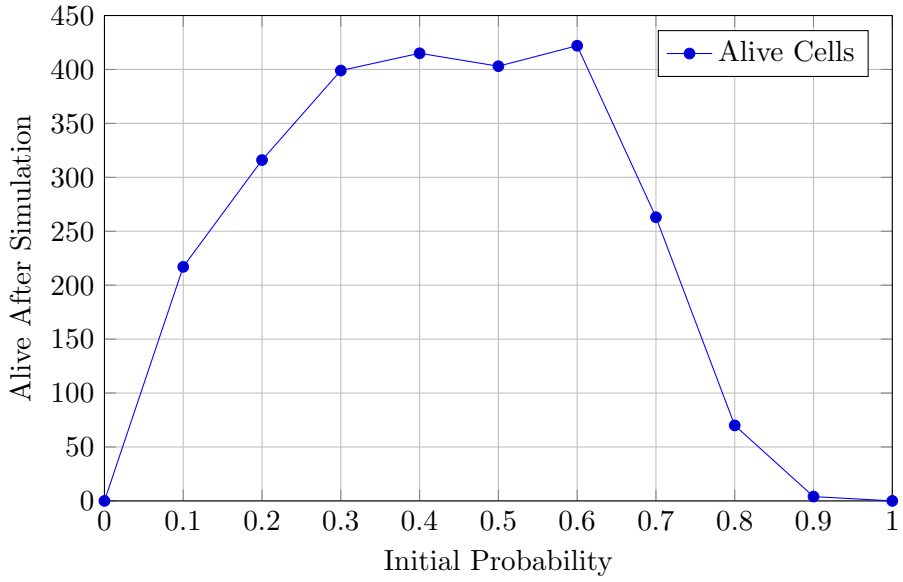
Table 1: Results for 100×100 grid experiments with varying initial probabilities.

Additionally, when exploring the effect of grid size at an initial probability of 0.3, the results were:

Dimensions	Alive Before Sim	Alive After Sim	% of Alive After Sim
50×50	729	87	3.5%
100×100	2987	557	5.6%
200×200	11959	1901	4.8%
400×400	47822	5931	3.7%
800×800	191816	27060	4.2%

Table 2: Results for varying grid dimensions at a fixed initial probability of 0.3, including the fraction of cells alive after 1000 steps relative to the total grid size.

The graph below shows the average number of living cells after 1000 simulation rounds as a function of the initial probability for a 100×100 grid.



2.6 Analysis and Conclusions

From the experiments, we observe that:

- At initial probabilities of 0.0 and 1.0, the final number of living cells is 0, indicating that when the grid is completely empty or completely full, the system quickly evolves to extinction.
- For moderate probabilities (around 0.3 to 0.6), the grid stabilizes to a state with a significant number of surviving cells. In particular, probabilities in this range allow for the emergence of stable or oscillatory patterns.
- When increasing the grid size at a fixed probability of 0.3, the absolute number of surviving cells increases. However, the proportion of living cells relative to the grid size remains comparable.

These results confirm my hypothesis that both grid dimensions and the initial probability strongly influence the final state of the simulation. The experiments demonstrate that overly sparse or dense initial conditions lead to extinction, while moderate conditions lead to sustained life.

3 Extensions

3.1 Command-line Parameters for LifeSimulation Class

3.1.1 What did I do and why?

For this extension, I modified the `main` method in the `LifeSimulation.java` class to accept command-line parameters. The purpose was to allow users to control key simulation parameters without having to modify the source code. The parameters introduced are:

1. Number of rows in the grid.
2. Number of columns in the grid.
3. Initial chance that a cell is alive (a decimal between 0 and 1).
4. Number of simulation iterations (time steps).

This change improves usability by enabling dynamic configuration of the simulation from the command line. A usage message is displayed if insufficient parameters are provided, ensuring that users know the required input format.

A `readme.txt` file is also provided in the extension folder that describes how to use the command-line parameters to run the `LifeSimulation.java` file.

3.1.2 What was the outcome?

The simulation now initializes the `Landscape` based on user-supplied dimensions and initial alive chance. For example, running:

```
java LifeSimulation 100 100 0.25 1000
```

creates a 100×100 grid where each cell has a 25% chance of starting alive, and the simulation runs for 1000 iterations. Additionally, the program prints the number of living cells before and after the simulation, providing immediate quantitative feedback on the simulation outcome.

3.1.3 How can you replicate my outcome in my code-base?

To replicate this extension:

1. Ensure that the modified `LifeSimulation.java` file is compiled along with the other necessary classes (e.g., `Landscape.java`, `LandscapeDisplay.java`, and `Cell.java`).
2. Run the program from the command line with the required parameters. For instance:
   ```
   java LifeSimulation 100 100 0.25 1000
   ```
3. Verify that the program prints a usage message if parameters are missing and that the simulation window updates accordingly. The console output will show the number of living cells before and after the simulation.

3.2 Glider Preset

3.2.1 What did I do and why?

For this extension, I created a new file called `Glider.java` to test a well-known pattern in Conway's Game of Life: the glider. Instead of initializing the board randomly, I configured the simulation to start with an empty grid (initial chance set to 0) and then manually activated specific cells to form the glider pattern. The coordinates chosen were:

- (2,0)
- (2,1)
- (2,2)
- (1,2)
- (0,1)

This pattern is known to produce a glider that moves diagonally across the grid. The motivation behind this extension was to verify that our simulation correctly evolves a preset pattern, and to explore the dynamic behavior of patterns that are well understood in Conway's Game of Life.

3.2.2 What was the outcome?

Upon running the `Glider.java` file, the glider pattern was observed to move diagonally across a 50×50 grid as expected. The simulation was executed for 10,000 iterations with a 250 millisecond pause between iterations. The preset pattern was clearly visible in the simulation window and evolved according to the rules of the Game of Life, thereby confirming that the simulation correctly handles predetermined configurations.

3.2.3 How can you replicate my outcome in my code-base?

To replicate this extension:

1. Ensure that all necessary classes (`Landscape.java`, `LandscapeDisplay.java`, `Cell.java`) and the `Glider.java` file are compiled.
2. Run the `Glider.java` file from the command line using:

```
java Glider
```

3. Observe the simulation window to verify that the glider pattern is displayed and moves diagonally across the grid.

3.3 Graphical User Interface

3.3.1 What did I do and why?

For this extension, I developed a complete graphical user interface for the Game of Life simulation. I created two new classes: `LifeSimulationGUI` and `LandscapeDisplayGUI`. The goal was to provide an interactive application that first prompts the user for simulation parameters (such as the number of rows, columns, and the initial chance of a cell being alive) via dialog boxes, and then runs the simulation in a single unified window with control buttons. These buttons allow the user to start/resume, pause, and reset the simulation. This extension improves usability by removing the need for command-line parameters and by enabling real-time interaction with the simulation.

3.3.2 What was the outcome?

The resulting application opens a single window containing the simulation canvas and a control panel at the bottom. Upon startup, the user is prompted to input the grid dimensions and the initial alive chance. After entering these parameters, the simulation is displayed in the center of the window. The control panel includes color-coded buttons:

- **Start/Resume** (green) to begin or continue the simulation.
- **Pause** (yellow) to temporarily stop the simulation.
- **Reset** (red) to reinitialize the simulation to its initial state.

This integrated GUI allows the user to control the simulation interactively, providing immediate visual feedback as the simulation evolves.

3.3.3 How can you replicate my outcome in my code-base?

To replicate this extension:

1. Ensure that the following files are compiled and available in my project directory:
   - `Landscape.java` (simulation model)
   - `LandscapeDisplayGUI.java` (custom display panel)
   - `LifeSimulationGUI.java` (the main GUI class that prompts for parameters and provides control buttons)
2. Run the application by executing:

   ```
   java LifeSimulationGUI
   ```

3. Upon startup, the program will display dialog boxes asking for:
   - Number of rows,
   - Number of columns,
   - Initial chance that a cell is alive.
4. After inputting these parameters, the simulation will appear in a single window along with the control buttons at the bottom.
5. Use the buttons to start/resume, pause, and reset the simulation as needed.

4 Acknowledgments

1. Java JFrame - GeeksForGeeks
2. Java Swing – JPanel With Examples - GeekForGeeks
3. Class JFrame - Oracle
4. Class JPanel - Oracle
5. Java ActionListener in AWT - GeekForGeeks
6. Class Timer - Oracle