The Game of Life simulation

Coursework 3: Life (Pair Programming)

This Report is to describe the function of The Game of Life simulation and the process of game creation by Java.

Jiangjing.Xu

21134158

Jiangjing.Xu@kcl.ac.uk

Hongyuan.Zhao

22022013

Hongyuan.Zhao@kcl.ac.uk

29/02/2024

Description of The Game of Life simulation

Our project builds on the famous Conway's Game of Life, a cellular automaton in a two-dimensional grid, where each cell's fate—survival, death, or change—is determined by a set of fixed rules based on the states of neighboring cells. Our aim was to expand and enhance this biomimetic system. We improved the rules for basic life forms and developed new ones, adding complexity with elements like disease spread and treatment, environmental impacts, non-deterministic cells, and symbiotic relationships. By modifying the rule set, we increased the simulation's complexity and interactivity, making its behavior more diverse and engaging.

Implementation

Our application initiates with a selection interface, offering users a choice between two simulation modes via a JavaFX ChoiceBox: "Simulate all cells" and "Simulate single cell." Selecting "Simulate all cells" displays various cells simultaneously on a single stage, with options to progress through generations either step-by-step or in a continuous sequence, alongside a reset function to clear both the generation count and the canvas. Conversely, "Simulate single cell" prompts a new window, requiring users to select a specific cell for individual simulation.

For the Base Task:

We've enhanced our cell simulation model by introducing an aging mechanism within the Cell class, allowing each cell to track its age. A maximum age threshold has been established, ensuring that cells naturally expire when this limit is reached. This lifecycle management is meticulously executed through the act() method, which now incorporates checkAge() to evaluate if a cell has aged beyond its viability, and updateAge() to increment the cell's age with each simulation step, providing a more dynamic and realistic representation of cellular lifecycles.

1. **Mycoplasma:**
   In the updated Mycoplasma simulation, cell behavior is governed by simple if-else statements reflecting three core rules: cells die from isolation or overcrowding if they have fewer than two or more than three neighbors, survive with two to three neighbors, and dead cells revive with exactly three neighbors.

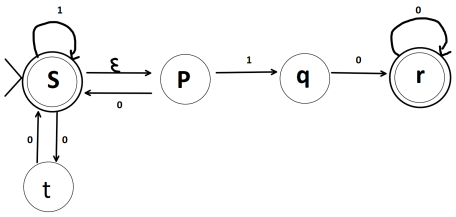
2. **MyFungiChangeColor:**
   We introduced a new class, MyFungiChangeColor, to showcase cells in various colors indicative of their life stages: blue for infancy, green for maturity, and

black for demise. The method checkAge() is employed to ascertain the age, followed by if-else logic to align the color with the cell's age and the count of adjacent neighbors. The setColor() method facilitates the color transition, while setNextState() updates the cell's condition. This life form's color transitions with the age and surrounding live cell count, enriching visual experience and interaction.

3. **MyEvolvingCellChangeBehaviors:**
   We developed a class named MyEvolvingCellChangeBehaviors to demonstrate how cell behaviors evolve over time. This class assigns three distinct behavioral patterns to cells at different stages of their life. For instance, in their youth, cells can survive with one or no neighbors, but as they mature, the necessity for two neighbors to sustain life kicks in. This approach showcases the adaptive strategies of cells in response to their aging process and environmental interactions.

For the Challenge Task:

1. **NonDeterministicCells:**



For non-deterministic cell behavior, we use four classes to accomplish it: NFA, NFAState, NFARun and NonDeterministicCells, which together build the NFA framework. The NFA class defines the structure of a nondeterministic finite automaton, including its set of states, transition functions, initial and acceptance states. It provides methods for adding states and transitions, as well as the ability to use "for-loop" to check whether a specific input string is accepted by the automaton. NFAState signifies an individual automaton state, contains matching input strings and corresponding transitions, and uses the getOrDefault() method to handle unavailable strings. NFARun, the automaton's operational core, executes via the main() method, loading configurations and parsing strings.

The `NonDeterministicCells` class implements cells with non-deterministic behavior by inheriting the act() method of the `Cell` class. The survival and death of these cells depend not only on the number of surrounding neighbors but also on random probabilities. Cells have a certain probability of dying when they have less than 2 or more than 3 neighbors and have a higher probability of surviving when they have 2 neighbors.

2. **Symbiosis:**
   Initially, we developed a Symbiosis class to form a community consisting of hosts and symbionts. Following this, we introduced two subclasses derived from the Symbiosis class: Parasitism and Mutualism. In the parasitic relationship, we utilized getLocation() methods to identify the positions of hosts and symbionts, then add a new method name areNeighbours() in Field class to ascertain if they are neighbors. If so, the host is adversely affected by accelerated aging, benefitting the symbiont through age reduction. In mutualism, we applied a similar approach to ensure both entities benefit from their proximity and interaction.

3. **Disease:**
   We created an class named `Disease` and defined a method called infectCell(), which determines if an infection occurs by generating a random number between 0 and 1, and comparing it to a fixed disease probability threshold (`CELL_DISEASE_PROB`). If infection is confirmed, the behavior is altered using the `infectedAct()` method.
   We add method getDiseasedAliveCells() of Field Class, which can return the number of diseased alive cell, and make a new label to show it.

4. **CellRecovery:**
   We create two class, Temperature class and CellRecovery class.
   For Temperature class, there is a method randomiseTemperature(), which dynamically adjusts the simulation's environmental temperature based on specific criteria, reflecting real-world ecological impacts on cell behavior. Additionally, we add a method getTemperatureCondition() in the Field class which can return a String of temperature description the view.
   The CellRecovery class provides methods for managing the recovery process of diseased cells in a simulation environment. It evaluates the conditions under which a cell can recover from a diseased state and implements the recovery process. Recovery is contingent upon specific conditions such as the cell's ability to contract diseases, its current health state, the presence of mutualistic relationships, and environmental factors like temperature.