Title: Final Project

Author: Josef Lazar

Background/Introduction:

In this lab we explore the behavior of Conway's Game of Life (CGoL) and relationship between starting cell arrangements and their outcomes. We will start by creating a program that can run CGoL on an expandable grid. We will then create an algorithm that can detect if the game is in a state where its behavior is regular and predictable. The next stage of the project will involve creating an algorithm that can compute every possible combination of living and dead cells within given grid parameters. After we have completed these two things, we will simulate every possible cell composition within a three by three grid until its behavior becomes predictable, or it reaches the boundaries of our computational capabilities. We will store the initial cell composition and its outcome cell composition, along with the type of repetitive behavior we detected and the amount of evolutions it took it to get there.

Our prediction is that most cell compositions will evolve into an empty grid or a simple still life or oscillator. Though they will probably be a minority, we also believe that there will be quite a few big complicated cell combinations that will reach our program's limits before their behavior becomes predictable. There may also be some grids that evolve into a combination of still lives or oscillators, and spaceships.

By undertaking the steps listed above, we will test our hypothesis and search for unexpected results. CGoL follows relatively simple logical rules, but is able to do endlessly complex things. This project will conduct tests on simple cell formations in CGoL and analyze the results, with the hope that this will broaden our knowledge of CGoL's behaviors and patterns more generally.

Methods:

- The program contains a main class, a GameOfLife class, a GameOfLifeMap class and a Button Class.

GameOfLife class

- The GameOfLife class stores a two dimensional boolean array named grid. In this report we will be referring to two dimensional boolean arrays more generally as grids. The first index of the array indicates the column, and the second

indicates the row of the grid. A false value symbolises a dead cell and a true value signifies a living cell.

- The GameOfLife class has two evolve() methods, one evolves a GameOfLife object's own grid, and the other takes a grid as an input, evolves it, and returns it. The evolution process is the standard CGoL evolution process, where a living cell with two or three living neighboring cells survives, a dead cell with exactly three living neighbors is revived, and all other cells die or stay dead.
- The GameOfLife class has a getGrid() method, which takes a grid and x, y, w, h coordinates as input, and returns a grid containing the area within the imputed grid marked by the coordinates. If the coordinates reach outside the grid, then all the outlying values are set to false.
- The GameOfLife class has a minimalize() method, which takes a grid as input and takes away rows and columns from the edges of the grid if they don't contain any living cells, until there is at least one living cell along all four edges. It then adds a dead column and row to each side and returns the grid. This is done so that all living cells have eight neighbors, but the grid isn't unnecessarily large. If the grid has no living cells, a grid containing just one dead cell is returned.
- The GameOfLife class has a gridsEqual() method, which takes two grids as input and checks if they are the same.
- The GameOfLife class has a stillLife() method, which checks if a GameOfLife object's grid is a still life. It does this by checking if its grid is equal to itself after one evolution.
- The GameOfLife class has an oscillating() method, which checks if a GameOfLife object's grid is an oscillator. It does this by checking if its grid is equal to any of its 20 most recent past selves.
- The GameOfLife class has a spaceShip() method, which checks if a GameOfLife object's grid is a spaceship. It does this by checking if its grid, when minimalized, is equal to any of its 15 most recent past selves, when minimalized. Minimization puts the living cells in the middle of the grid, so the difference in location between a spaceship currently, and when it had the same shape last, is eliminated, hence the grids become equal. To make sure that the location has indeed changed, the grid must also specifically not be oscillating.

GameOfLifeMap class

- The GameOfLifeMap class is used to store the results of our simulations. Its Attributes include two grids named gridInput and gridOutput, which store the initial cell configuration and the resulting cell configuration, respectively. They also include a string named type, which is used to indicate the repetitive behavior that the algorithm detected in the resulting cell configuration, and they include an

integer named evolutions, which stores how many generations it took to get from the initial configuration to the resulting configuration.
- The GameOfLifeMap class has a displayGrid() method, which takes a grid and x, y, w, h coordinates as input, and displays the imputed grid in the imputed coordinates. It colores dead cells black and living cells white.
- The GameOfLifeMap class has a display() method, which displays its gridInput and gridOutput grids on the left and right sides of the screen, respectively. If gridOutput is an oscillator or a spaceship, it is displayed evolving. It also displays how many evolutions it took to get from one to the other.

Button class
- The Button class is used to make buttons that the user can click to switch between which GameOfLifeMap objects they want to display. For example there is a button to view all grids, or to view grids that evolved into spaceships, or to view grids that evolved into still lives, etc.. Importantly, the Button class also stores an array list of GameOfLifeMap objects which contains the maps that get displayed when the button is pressed.
- The Button class has an addMap() method, which takes a GameOfLifeMap object as input and adds it to the array list. In the program, this method is called every time the simulator finds a new resulting grid which fits the description of the button. (for example if it is an oscillators button, and an oscillating grid is found)
- The Button class has a display() method, which displays the button at its location, and makes it change colors depending on if it is pressed, if it's maps are being displayed and if the mouse cursor is on it.

Main class
- The Main class has a function called nextGrid(), which is used to discover all possible cell combinations within a grid of given parameters. It takes a grid as input, marks its width, and converts it to a one dimensional boolean array by stacking all its rows into a single long row. If the row is made up of dead cells, it revives its first cell. If it contains at least one living cell, it takes its last living cell and moves it one spot further to the back. If its last living cell is in the last position, it moves the second to last living cell back one spot and moves the last living cell behind it. Similarly if the second to last living cell is in the second to last spot it moves the third to last living cell back one and pushes the last two cells up behind it. If all living cells are pressed against the end of the row, it revives the first cell and pushes all the cells at the end up behind the first cell. If all the cells in the row are alive, the row is unchanged. Once the row has been updated based on these rules, its initial width is used to convert back to a grid. This grid is returned. Inputting an empty grid and inputting its output indefinitely, will

ultimately cycle through all possible cell combinations in the grid, and will result in a grid where all the cells are alive.

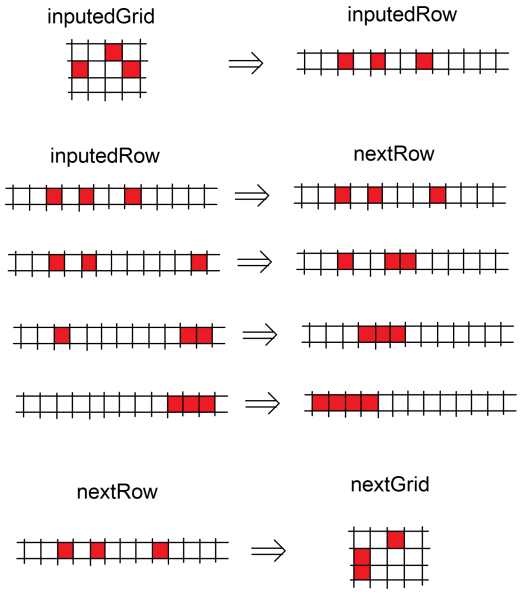

Figure 1: Shows an example of how nextGrid transforms an imputed grid, and how it processes cell rows.

- The Main class contains an array list of grids called allGrids3x3, whose elements are every possible cell formation in a three by three grid. This does not include grids whose living cell formation is already included in a previous grid but in a different location. For example, a grid with one living cell in the top right corner will not be included if the list already contains a grid with one living cell in the top left corner, because even though they are different grids, their living cells (or cell in this case) make up the same formation.
- The Main class contains a different button to view each of the following grid outcomes: all outcomes, empty grids, still lifes, oscillators, grids that are too big for the program's constraints, and spaceships.
- The Main class's draw function by default takes the first grid in allGrids3x3 and evolves it until its behavior is repetitive and predictable, or until its width or hight are larger than 150, in which case it is considered too big. At this point it adds it to the appropriate buttons, and proceeds to the next grid in allGrids3x3. After it finishes simulating all the grids, it changes to display mode. During the simulation, the user can press 's' to skip to display mode, though only the grids that are done simulating will be displayed. If they do this, they can continue the simulation by pressing 'c'. While in display mode, the user can scroll through the maps with the arrow keys, and can change the map category by pressing the buttons at the bottom of the screen.

Results:

We created a program that computes all possible three by three grids in Conway's Game of Life, and then evolves them until their behavior becomes repetitive and predictable, or they become too big. We discovered that there are 401 different three by three grids, which includes the grid containing only dead cells, and grids that are mirror flips or 90 degree rotations of other grids. There were however only 13 different grids that they all evolved into: the empty grid, the blinker, the block, the bee-hive, 4 blinkers forming a cross, the pond, the tub, the loaf, the glider, the boat, the ship, a large oscillator formed out of blinkers, ponds and blocks, and a larger grid that exceed our programs limitations before it became repetitive. Though there were more than 13 different resulting grids, when mirror flips and 90 degree rotations were counted separately, this is still a surprisingly low amount of variation. All the initial grids that resulted in the grid that exceeded our program's limitations (marked 'Big grid' in Figure 2) took 349-351 evolutions to get there, the initial grids that resulted in the Big oscillator took 175-177 evolutions to get there, and all other initial grids became repetitive within 15 evolutions.

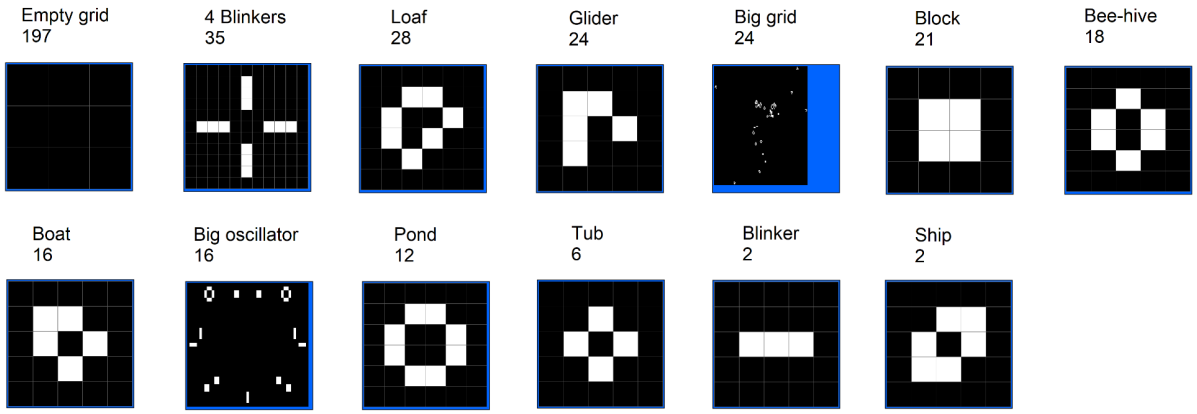

Figure 2: Shows all the possible outcomes of a three by three grid in Conway's Game of life. Below the name of the grid is a number which shows how many such grids evolve into it.

Our hypothesis that most grids will evolve into an empty grid or a simple still life or oscillator proved to be true. The empty grids made up almost half the outputs (197 out of 401), and simple still lifes and oscillators combined (which excludes Glider, Big grid and Big oscillator from Figure 2) to 140. Hence, most (337 out of 401) grids were empty grids, or simple still lifes or oscillators.

Our hypothesis that there will be quite a few big complicated cell combinations that will reach our program's limits before their behavior becomes predictable proved to be mostly wrong. Though there were 24 grids, whose resulting grid fit this description, they all evolved into the same grid (or some mirror or 90 degree rotation alteration of it), and our hypothesis suggests that such cell combinations will be both numerous and various. It was highly unexpected that there would be only one resulting cell combination that reached the program's limits, and that there would be 24 initial cell combinations that evolved into this same grid.

We decided to simulate this particular grid to completion and found that after 1402 evolutions it becomes a repetitive and predictable combination of still lifes, oscillators and space ships. Our last hypothesis that there will be some grids that evolve into a combination of still lives or oscillators, and spaceships has therefore been proven true. This is depicted in Figure 3 and Figure 4. In doing this we have also proven that the glider is the only spaceship that can emerge from a cell formation that fits in a three by three grid.

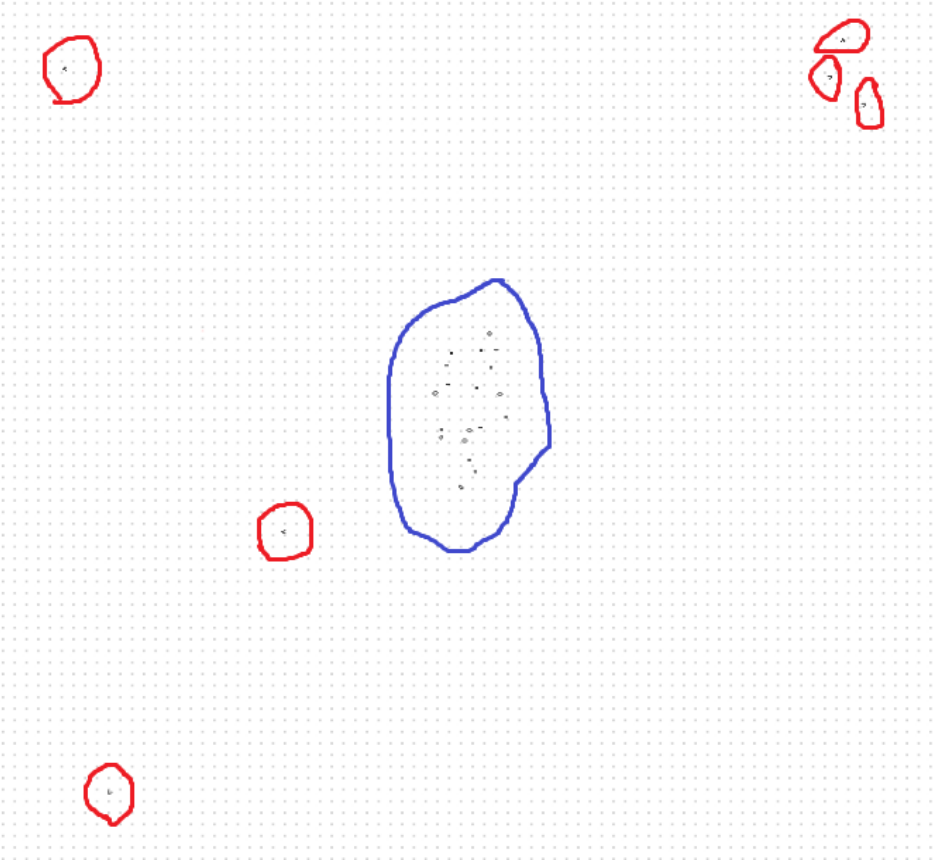

Figure 3: Depicts the grid titled ‘Big grid’ in Figure 2 after 1402 evolutions. The cell formations circled in red are gliders. The cluster of cell formations in the center circled in blue is made up of oscillators and still lives. They are shown in more detail in Figure 4.

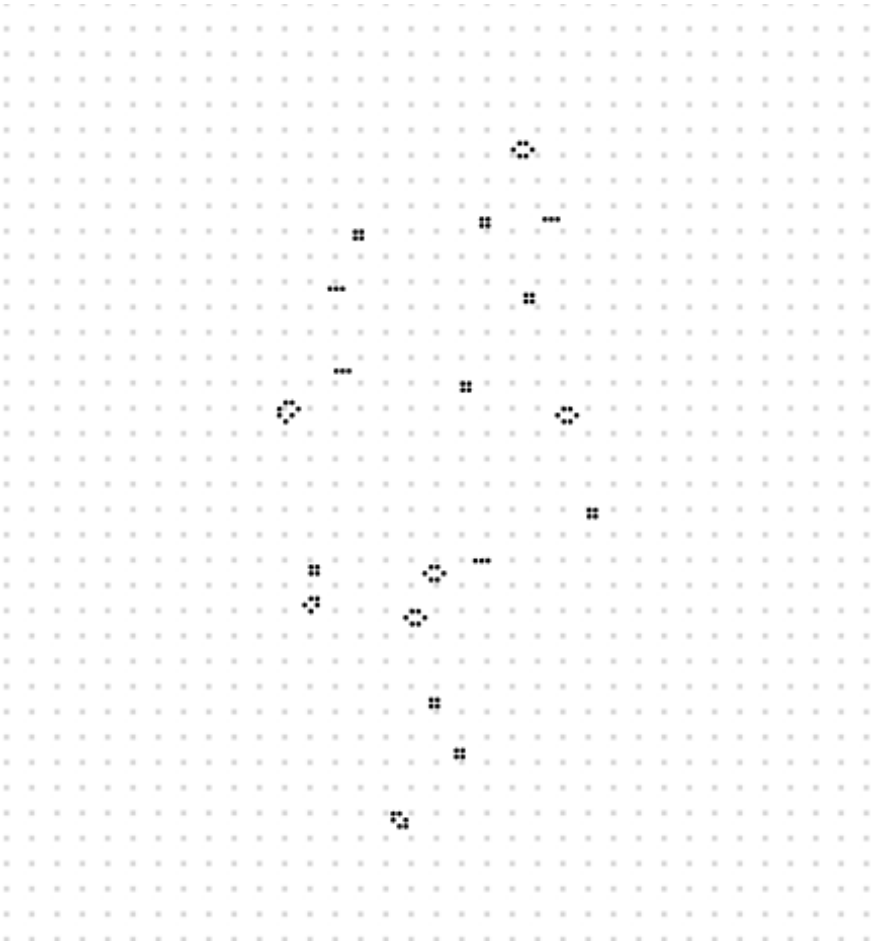

Figure 4: Depicts all the oscillators and still lives that emerged after 1402 evolutions from the grid titled ‘Big grid’ in Figure 2.

Conclusion: We made a program that simulates Conway’s Game of Life and uses a grid generating algorithm to make grids to input into it. The program has proven effective at mapping basic initial grids in CGoL to the stage at which their behavior becomes simple and predictable. Using this tool we discovered that all cell formations in CGoL that fit into a three by three grid behave in a limited and highly predictable way. Specifically there are only 13 distinct patterns that they can produce. Out of these 10 are simple oscillators, still lives or the empty grid, which emerged about half the time. Furthermore 11 out of 13 resulting patterns were always achieved within 15 evolutions. In stark contrast, the two that took longer took up to 177 and 1402 evolutions. Perhaps further mapping could ease the burden of simulations by discovering patterns and overlaps within the complex system that is Conway’s Game of Life.

Credit/Acknowledgements: We used the following websites for this project:
https://www.tutorialkart.com/java/how-to-check-if-arraylist-is-empty-in-java/
https://www.conwaylife.com/wiki/Spaceship
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
https://en.wikipedia.org/wiki/Spaceship_(cellular_automaton)
https://en.wikipedia.org/wiki/Glider_(Conway%27s_Life)
https://beltoforion.de/en/game_of_life/
http://www.cuug.ab.ca/dewara/life/life.html
https://processing.org/reference/for.html
https://processing.org/reference/color_.html
https://processing.org/reference/Array.html
https://processing.org/reference/String.html
https://processing.org/reference/keyCode.html
https://processing.org/reference/max_.html
https://processing.org/reference/ArrayList.html
https://processing.org/reference/Array.html
https://processing.org/reference/strconvert_.html
https://processing.org/reference/String.html
https://processing.org/reference/append_.html
https://processing.org/reference/int.html
https://processing.org/reference/IntList.html

Citations:

How to Check if ArrayList is Empty in Java, TutorialKart.
https://www.tutorialkart.com/java/how-to-check-if-arraylist-is-empty-in-java/.

Spaceship, LifeWiki.
https://www.conwaylife.com/wiki/Spaceship.

Conway's Game of Life, Wikipadia.
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

Spaceship (cellular automaton), Wikipedia.
https://en.wikipedia.org/wiki/Spaceship_(cellular_automaton).

Glider (Conway's Life), Wikipedia.
https://en.wikipedia.org/wiki/Glider_(Conway%27s_Life).

The Game of Life, beltoforion.de.
https://beltoforion.de/en/game_of_life/.

Conway's Game of Life, cuug.ab.ca.
http://www.cuug.ab.ca/dewara/life/life.html.

Reference, Processing Tutorial, D. Shiffman.
https://processing.org/reference.