Conway Game of Life – 100 course points

The purpose of this assignment is to practice your understanding of 2D arrays and union-find.

Start your assignment early! You need time to understand the assignment and to answer the many questions that will arise as you read the description and the code provided.

Refer to our Programming Assignments FAQ for instructions on how to install VSCode, how to use the command line and how to submit your assignments.

See this video on how to submit into Autolab.

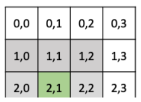
Overview

Conway Game of Life is a cellular discrete model of computation devised by John Horton Conway. The game consists of a game board (grid) of ***n*** *x* ***m*** cells, each in one of two states, *alive* or *dead*.

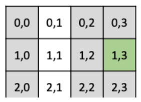
The game starts with an initial pattern, then it will change what cells are alive or dead from one generation to the next depending on a set of rules. As the Game of Life continues, the game will keep making a new generation (based on the preceding one) until it reaches one of three states.

Cells

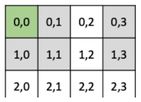
Each cell can be in one of two states, alive or dead, and it has 8 neighboring cells which are the cells that are horizontally, vertically, or diagonally adjacent. The figure below exemplifies neighboring cells on a game board of 4 x 4 cells.



0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3



0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3



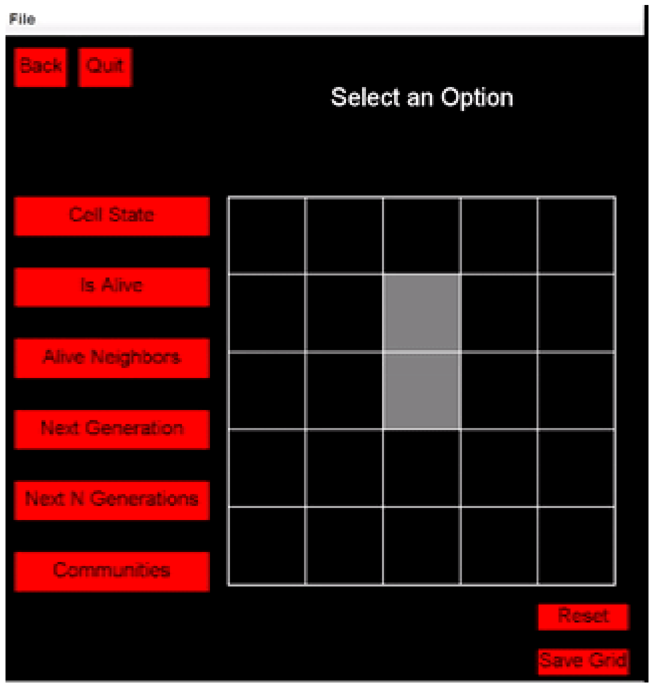
0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

Rules of the Game

The game starts with a initial set of alive cells (this is the first generation). The next generation evolves from applying the following rules simultaneously to <u>every</u> cell on the game board, i.e. births and deaths happen simultaneously. Afterwards, the rules are iteratively applied to create future generations. Each generation depends exclusively on the preceding one.

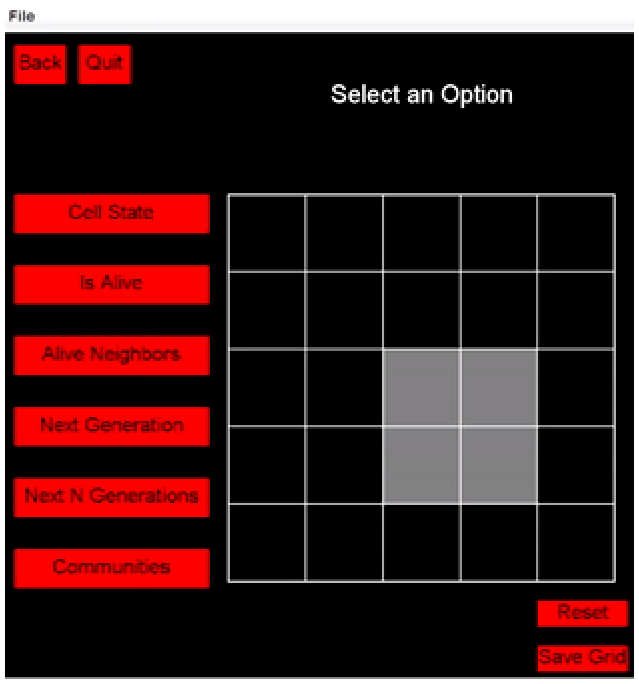
Rule 1

Alive cells with no neighbors or one neighbor die of loneliness.

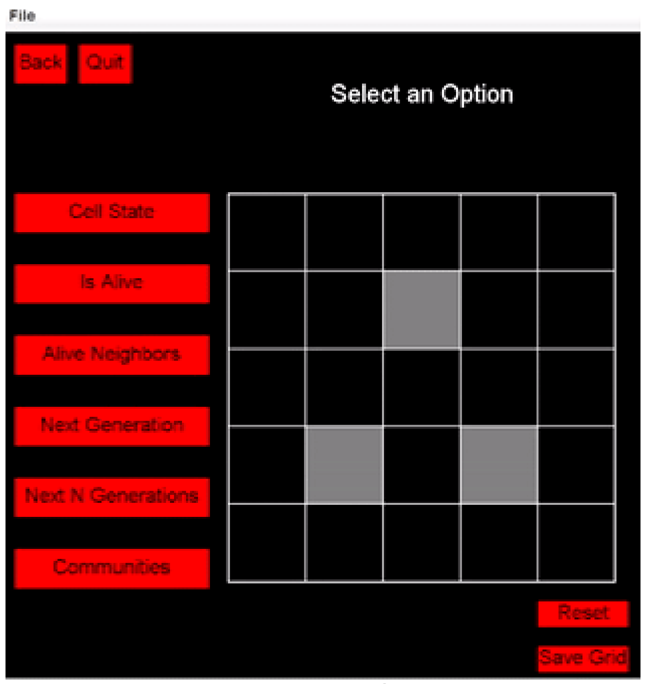


Rule 3

Alive cells with two or three neighbors survive.



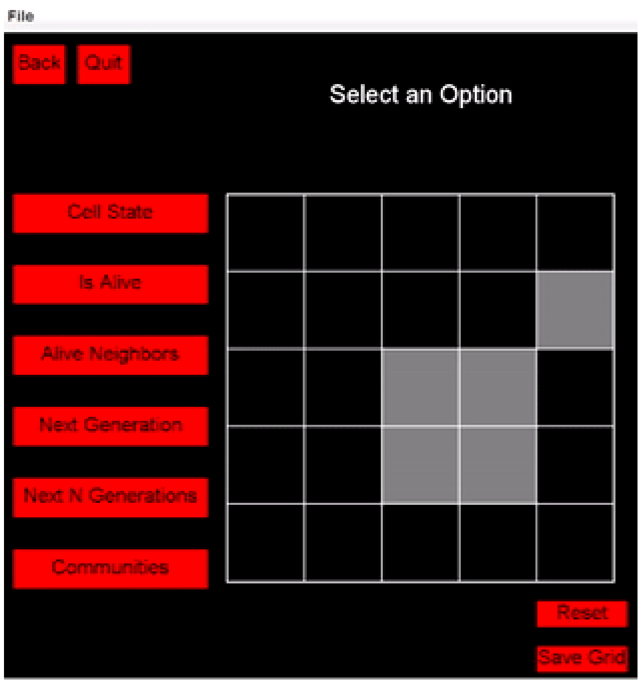
Dead cells with exactly three neighbors become alive by reproduction.



Notice the center cell comes to life but the other three die from Rule 1.

Rule 4

Alive cells with four or more neighbors die of overpopulation.



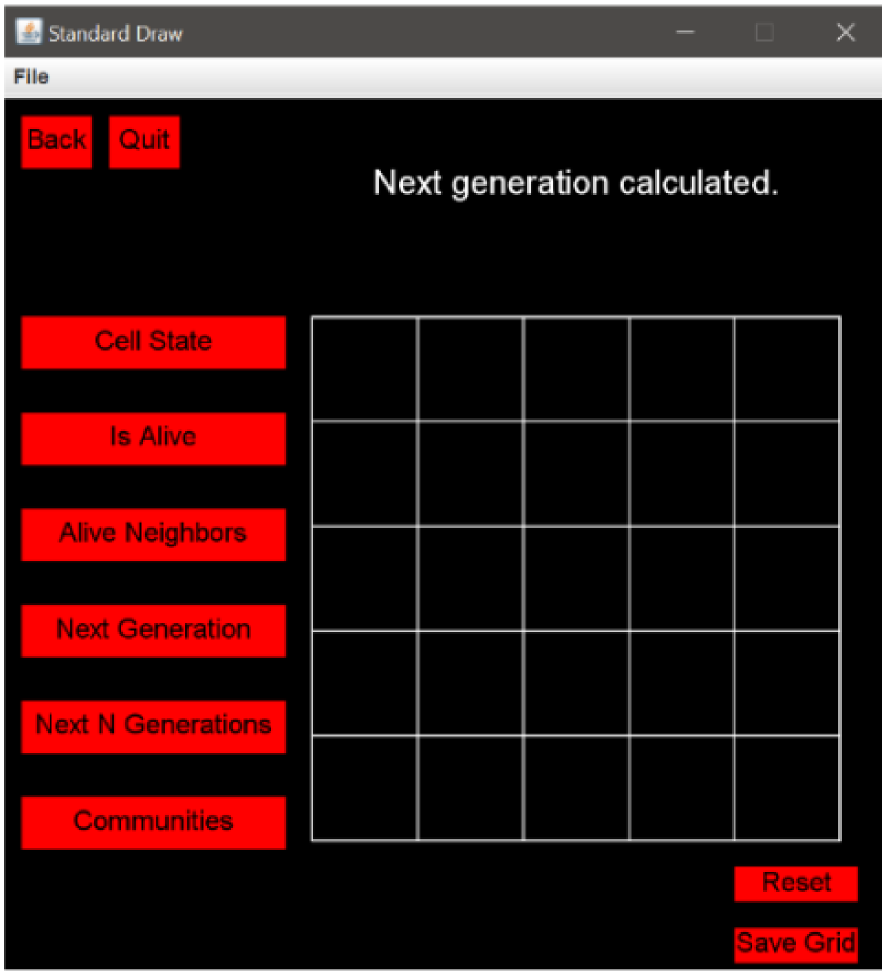
Notice the top right cell has four neighbors and dies from over population, the topmost dies from Rule 1, the other three from the square survive from Rule 3, and two cells come to life from Rule 2.

States for the Game

There are three states that the game can reach in regards to the rules provided.

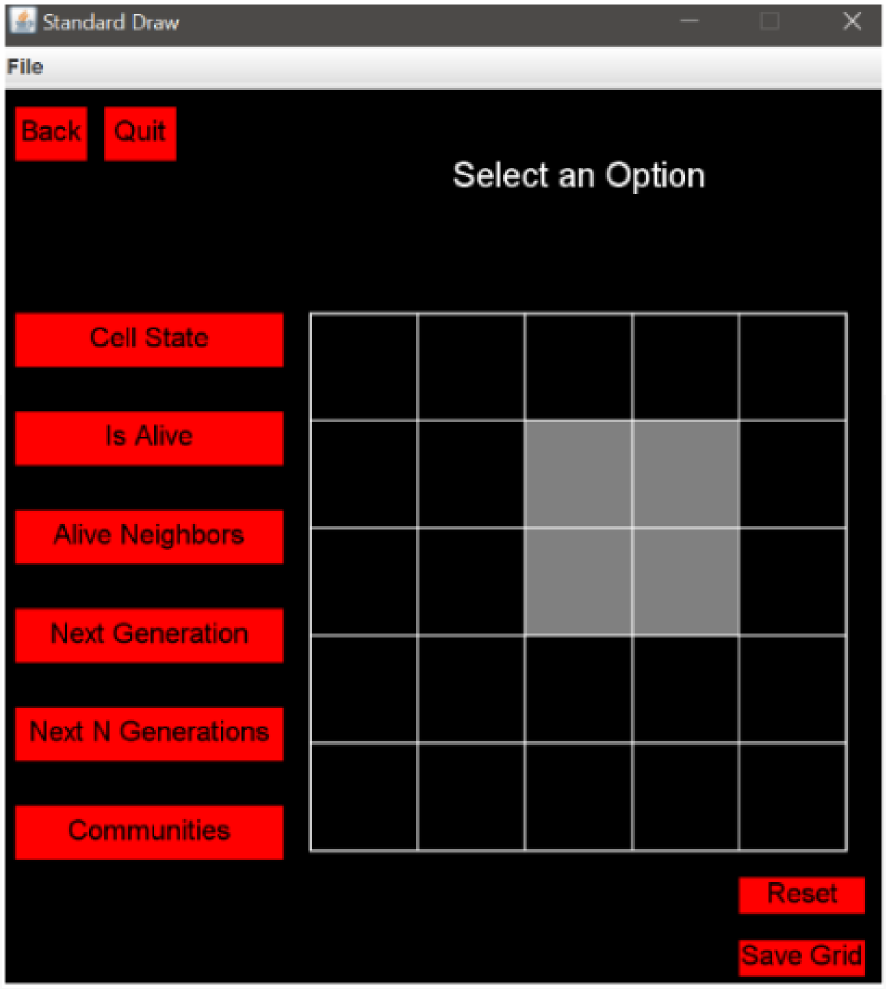
1. There are no more living cells for the next generation.
2. There are living cells, but the next generation is the same as the last (stable game).
3. The next generation infinitely cycles throughout the game.

Note that there are more scenarios for states 2 and 3, these are just one example for each.

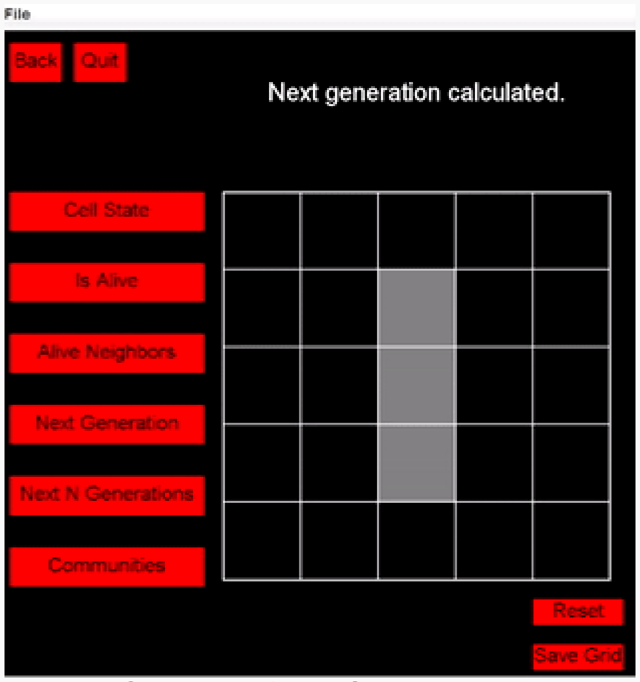


Next generation calculated.

State 1: No Living Cells



State 2: Stable Generation



State 3: Infinite Generations

Implementation

Overview of files provided

- **GameOfLife class** holds all of the methods to be written that will then be tested when running the game. Edit the empty methods with you solution, but **DO NOT edit the provided ones or the methods signatures of any method**. This is the file you submit.
- **WeightedQuickUnionUF** class, which houses the union-find data type. **Do not edit this class**. This newly learned algorithm holds two useful methods to operate on a given data structure. The `find` method will return a parent index of an array given the element you are looking for. The `union` method finds the parent of two elements given and will determine how to join the two elements together. You will need to utilize an instance of this class in the GameOfLife class to compute the number of communities of cells.
- **Driver** class, which is used to launch the Game Of Life and test the various methods. You can run the driver by simply hitting the run button above the main method or clicking the run button in the top right of the IDE (recommended). Alternatively you can type in java Driver in the IDE terminal (just make sure you are in the correct directory with the ls command and make sure everything is compiled, look at "Executing And Debugging" section below for more info). Feel free to edit this class, as it is provided only to help you test. It is not submitted and it is not used to grade your code.
- **Board, Button, Page, Rectangle, Text, and StdDraw** classes hold various methods to support the interactive driver.
- **StdIn** and **StdOut**, which are used by the driver. **Do not edit these classes.**.
- Multiple text files (grid0.txt, grid1.txt, grid2.txt, grid3.txt, grid4.txt, grid5.txt, grid6.txt, grid7.txt) which contain the initial pattern, and are read by the driver as test cases. Feel free to edit them or even make new ones to help test your code. They are not submitted.

GameOfLife.java

- DO NOT add new import statements.
- DO NOT change any of the method's signatures.

Methods to be implemented by you:

1. GameOfLife – One argument constructor

- This method builds your game board (grid) from the input file (initial game pattern).
- You have been provided some input files to test this method (input1.txt, input2.txt, input3.txt). The format is as follows:
    - One line containing the number of rows in the board game
    - One line containing the number of columns in the board game
    - Several lines (one for each row) containing the cell state (false for dead, true for alive) for each column in the board, space separated

Use the **StdIn** library to read from a file:

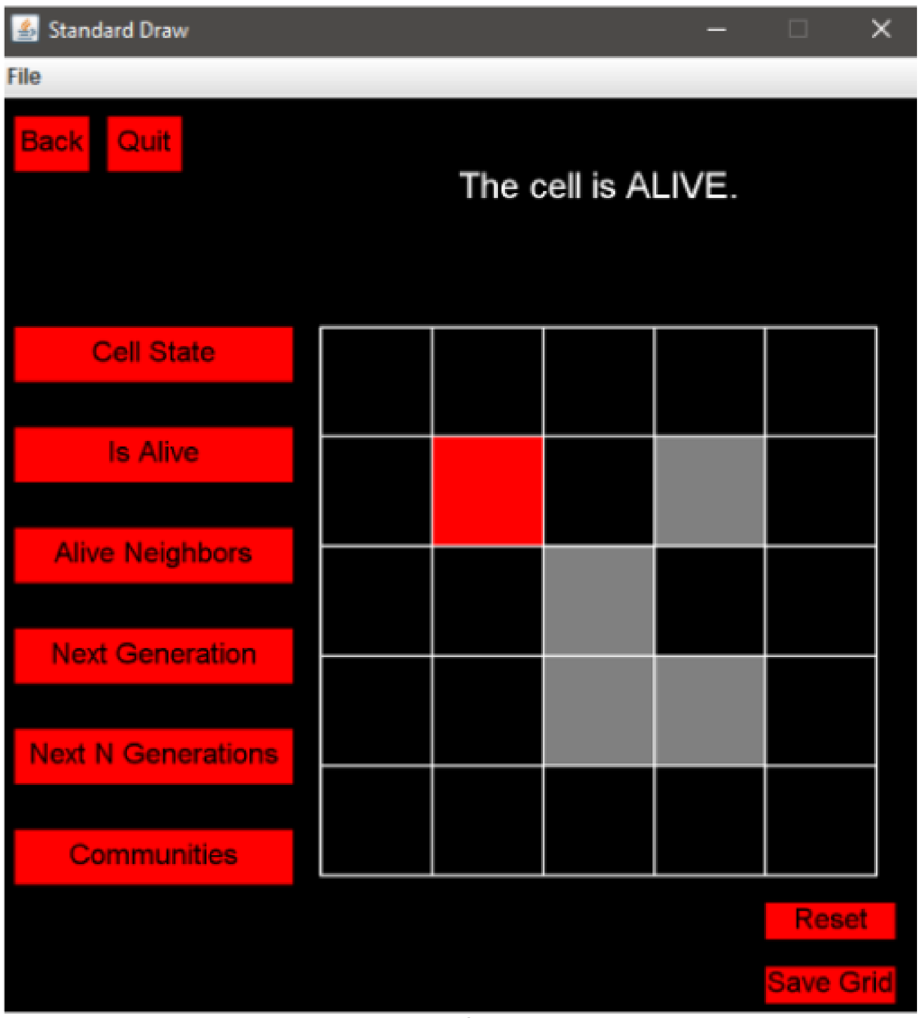
- **`StdIn.setFile(filename)`** opens a file to be read
- **`StdIn.readInt()`** reads the next integer value from the opened file (weather the value is in the current line or in the next line)
- **`StdIn.readBoolean()`** read the next boolean value from the opened file (weather the value is in the current line or in the next line)

2. getCellState

Given two integers representing the cell row and column this method returns true if the cell is alive and false if the cell is dead.

Test this method using the driver:

- upon clicking the button **Cell State** you will be prompted to select a cell
- select a cell by clicking in one square
- the selected cell will light up red and the driver will state whether the cell is living or dead.

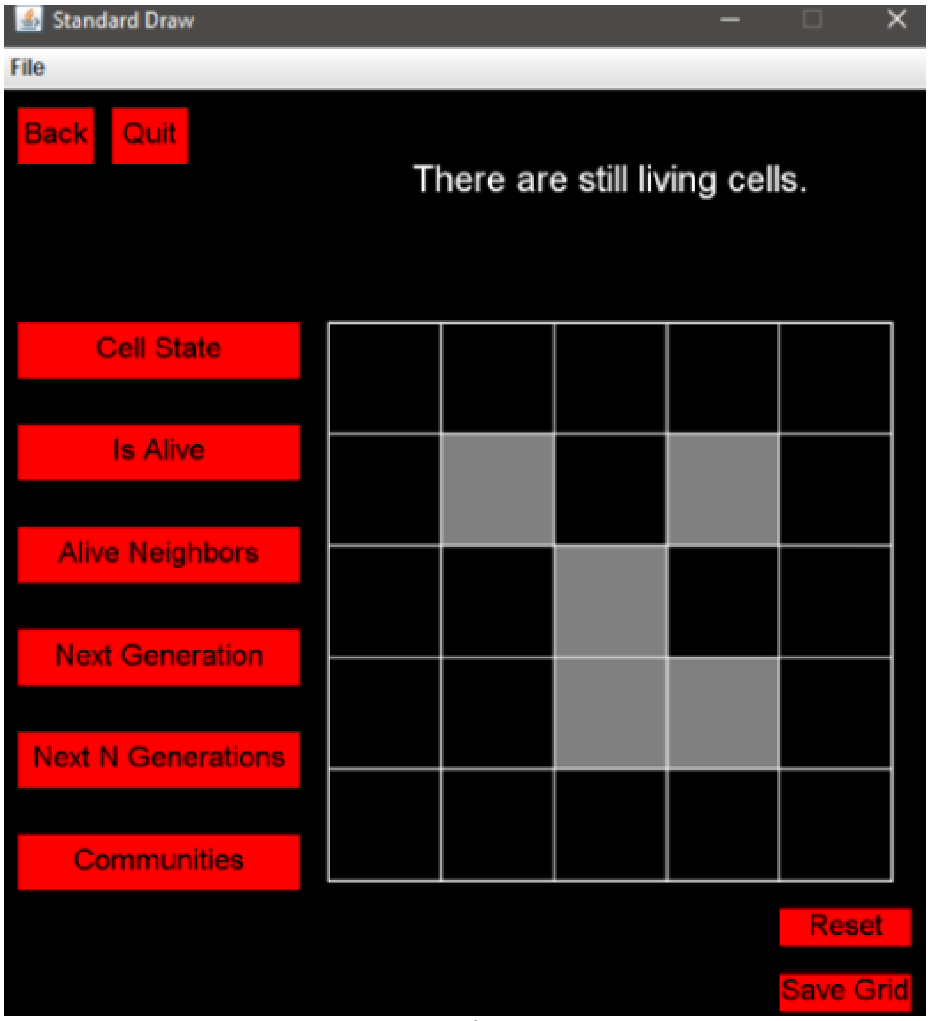


Example with default constructor

3. isAlive

Returns true or false based on whether or not there are living cells within the board (grid)

Test this method using the driver. Click on the **Is Alive** button and the driver will display whether or not there are living cells on the board game.



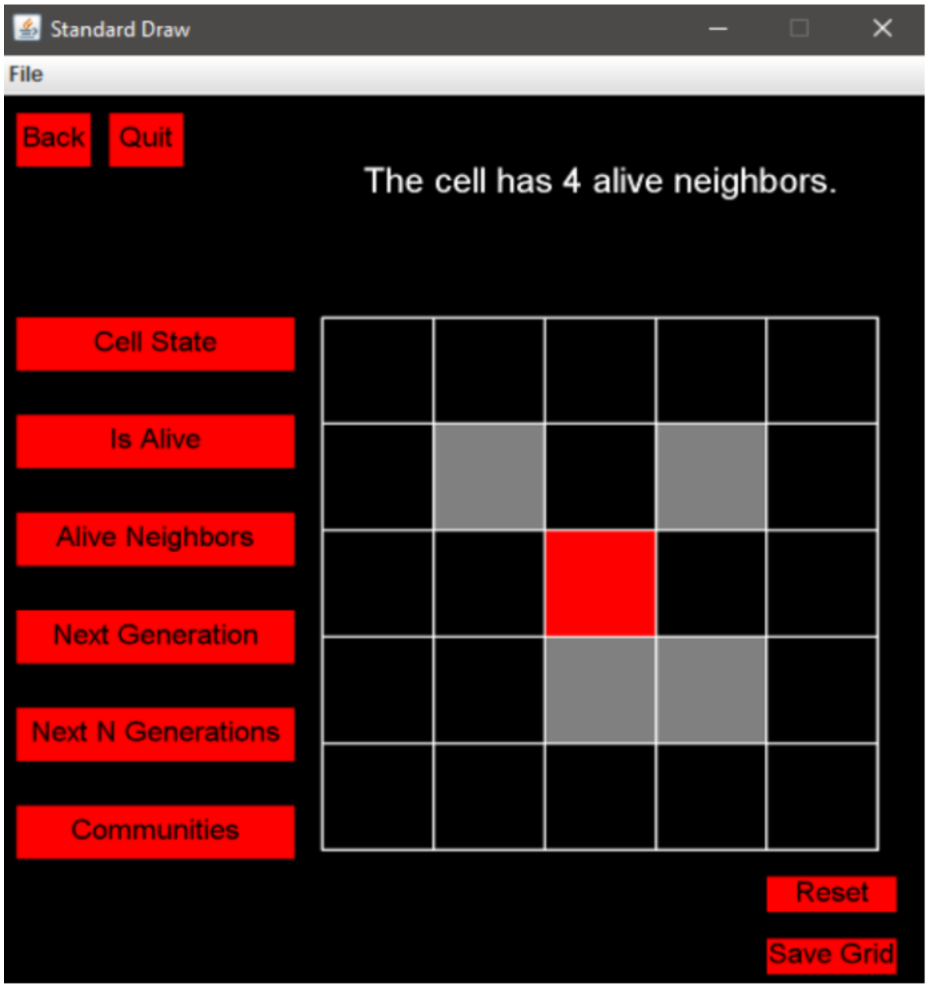
Example with default constructor

4. numOfAliveNeighbors

Given two integers representing the cell row and column this method returns the cell's number of alive neighbors of a maximum of 8 neighbors.

Test this method using the driver:

- upon clicking the button **Alive Neighbors** you will be prompted to select a cell
- select a cell by clicking in one square
- the selected cell will light up red and the driver will state the total number of alive cells around the selected cell



The cell has 4 alive neighbors.

Example with default constructor

5. computeNewGrid

This is where you will be using the rules of the game (stated above) to compute the next generation of cells.

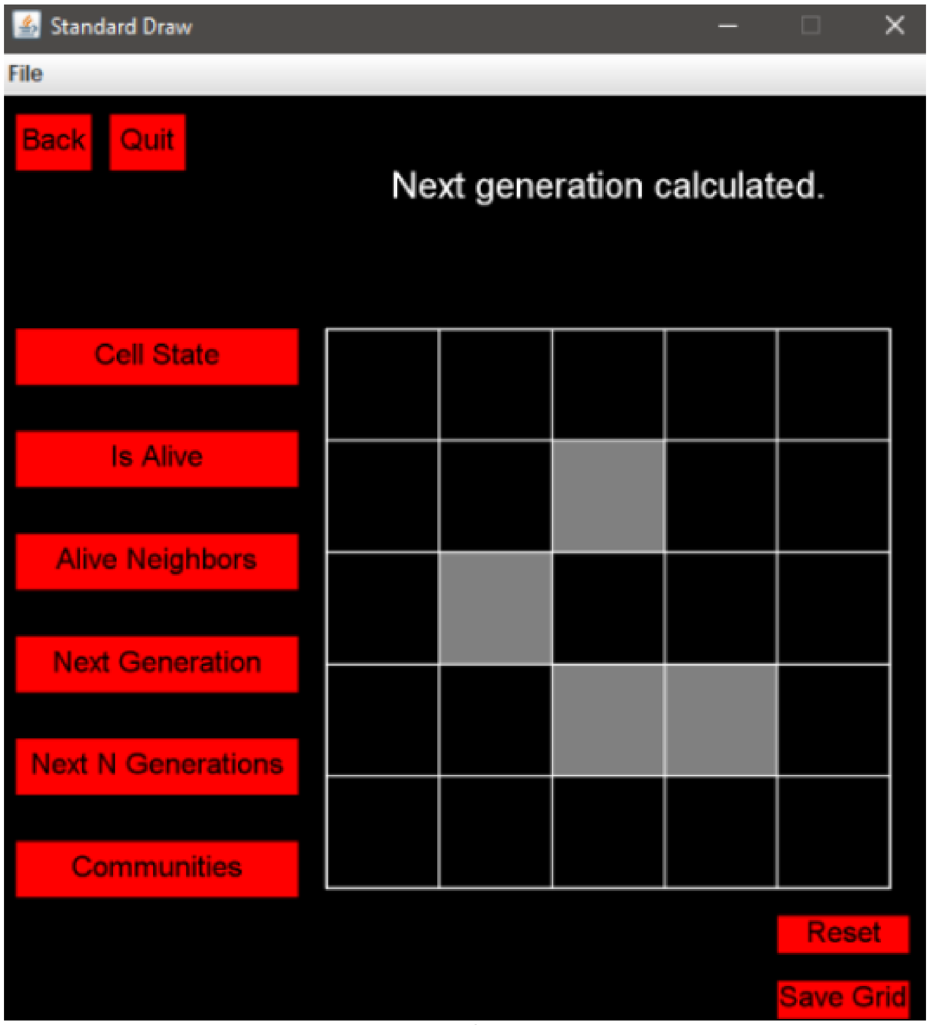
- create a new board (grid) to be returned, representing the new generation
- for each cell, use the numOfAliveNeighbors method to determine how many cells are alive around a cell
- then using the number of alive neighbors with the rules of the game determine if the cell will be set to be alive or dead in the new grid

6. nextGeneration

Update the board (grid) with the board returned by the computeNewGrid method.

Test this and the computeNewGrid methods using the driver by clicking the **Next Generation** button.

The driver will state **Next generation calculated**.



Next generation calculated.

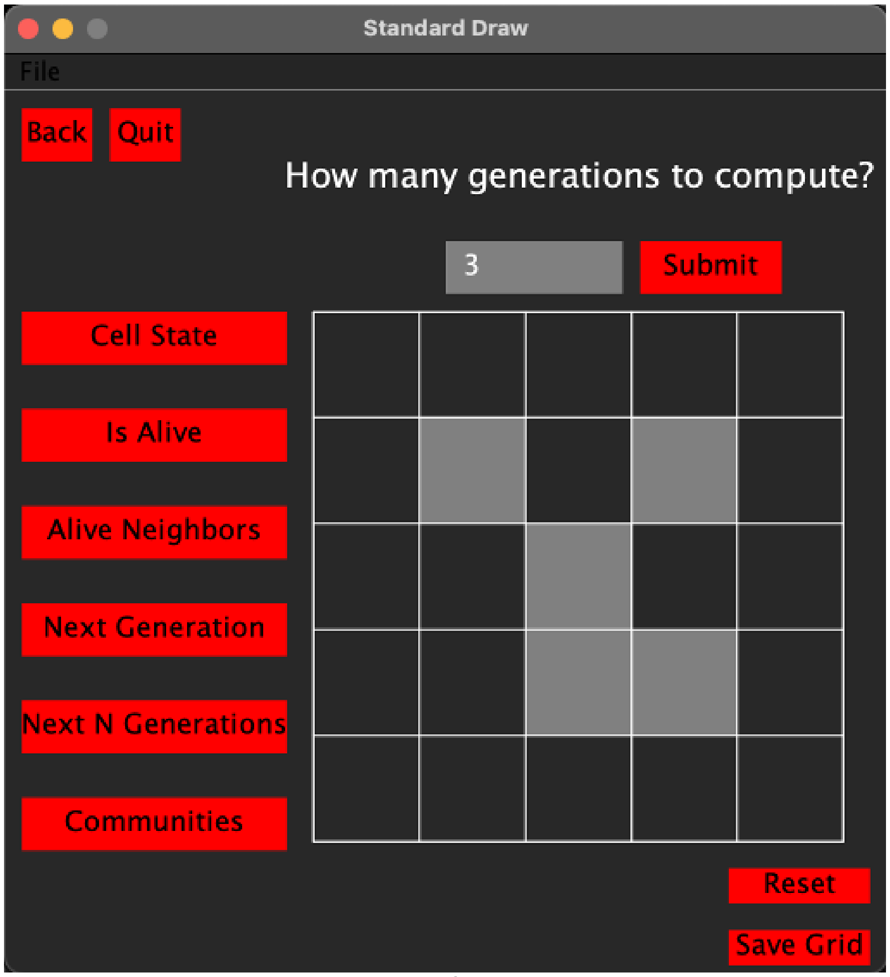
Example with default constructor

7. nextGeneration – One argument

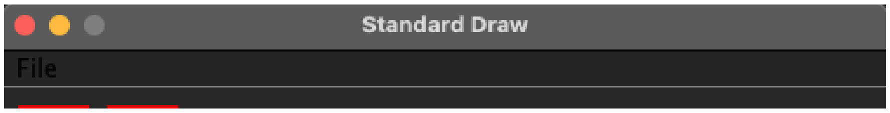
The input integer parameter represents the number of generations to compute (fast forward).

Test this methods using the driver:

- upon clicking the **Next N Generations** button you will be prompted to enter a number
- enter a number and click the **Submit** button
- the driver will state that the board has evolved the number of generations requested



Example with default constructor



8. numberOfCommunities

This method computes and returns the number of separate communities of cells in the board. A community is made of connected cells.

Recall that Union-Find keeps track of connected components. In this assignment each connected component is a community of cells (all the cells in the community are connected).

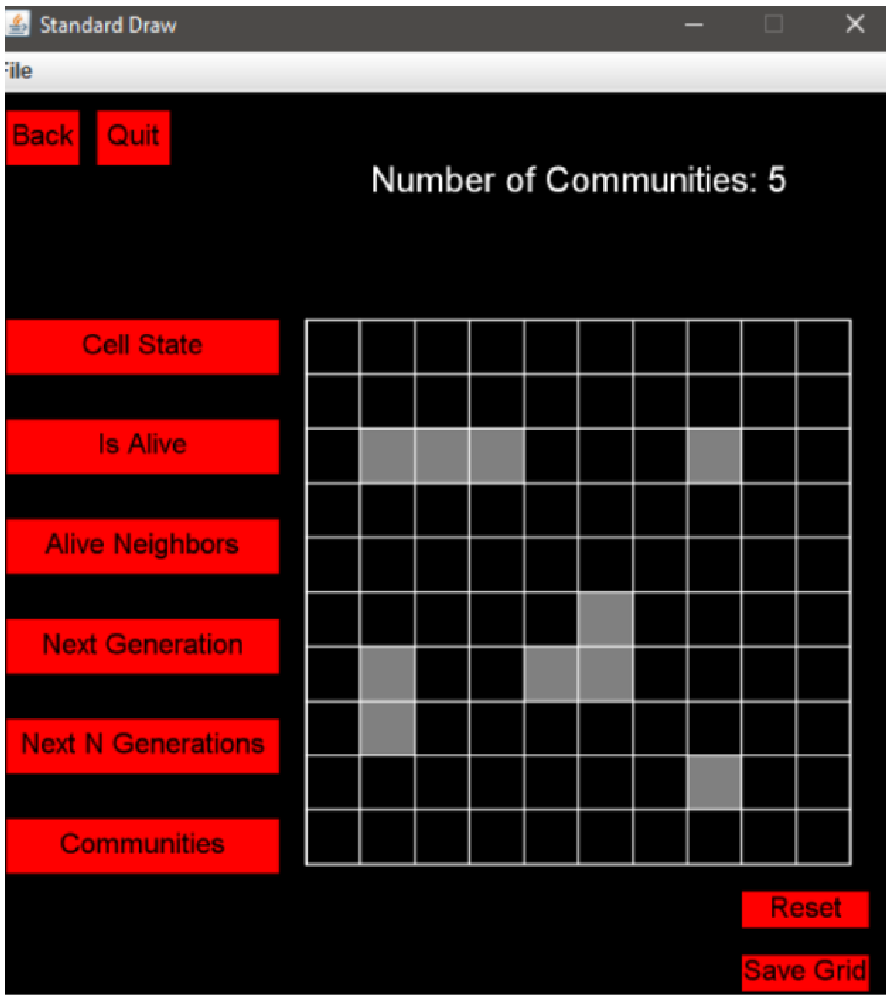
- Cells from separate communities are not connected.

In algorithm Weighted Quick Union UF each community (connected component) is a tree. The community's representative is the root of the tree.

- To connect two cells use the *union* method
- To find the root of the tree a cell belongs to use the *find* method
- To find the number of communities count the number of unique roots of trees

Test this methods using the driver by clicking the **Communities** button.

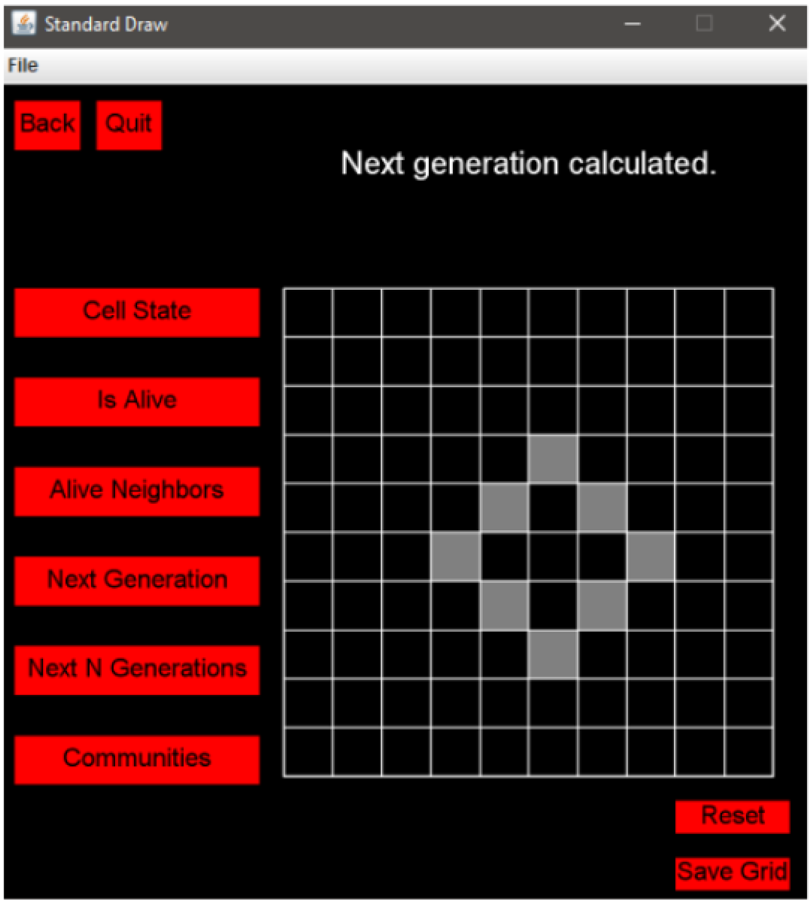
The driver will state the number of communities currently on the board.



Number of Communities: 5

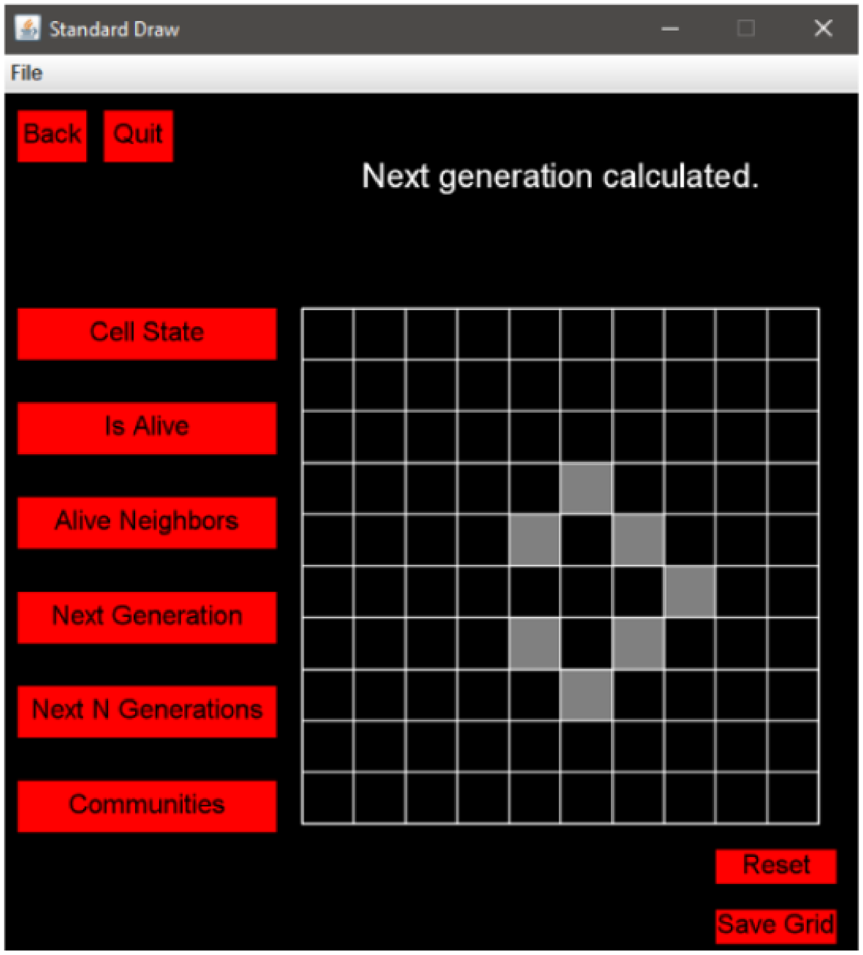
Find below the expected outputs after going through one generation of grids 1-6

Grid 1



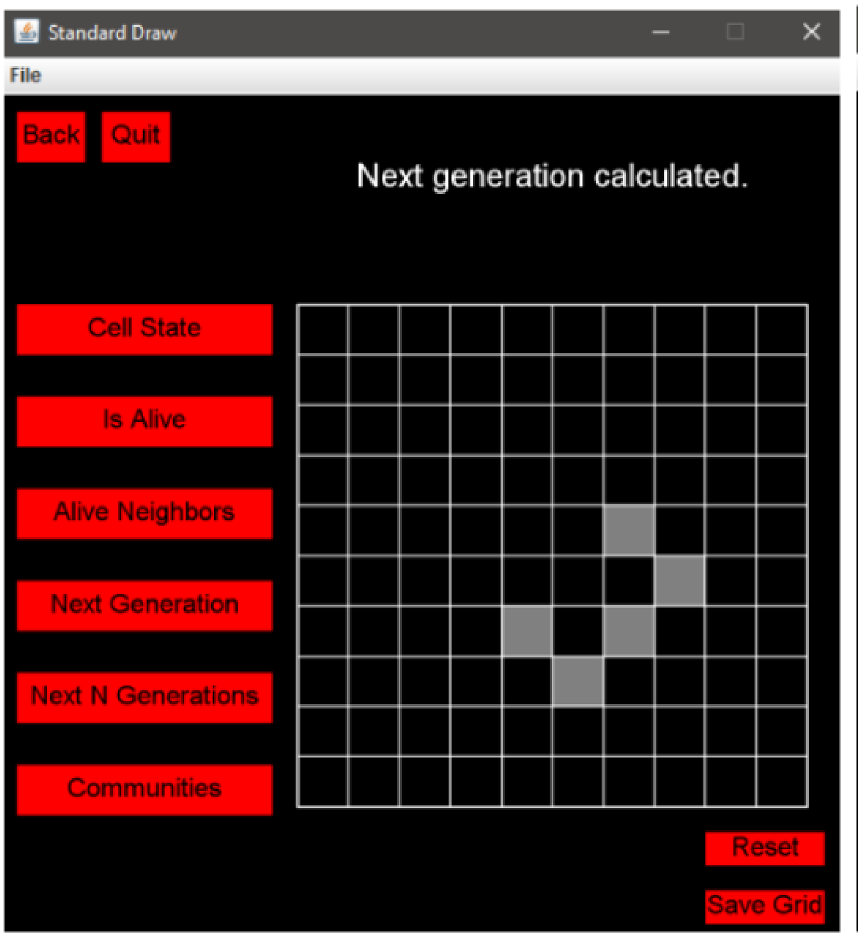
Next generation calculated.

Grid 2



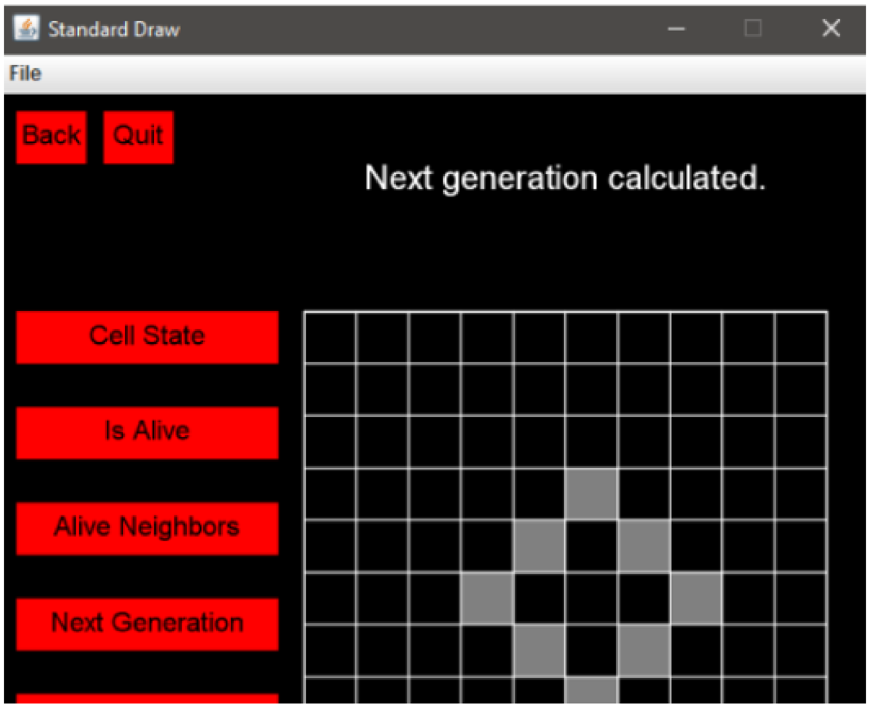
Next generation calculated.

Grid 3



Next generation calculated.

Grid 4



Next generation calculated.

Implementation Notes

- YOU MAY only update the methods GameOfLive(String), getCellState(), isAlive(), numOfAliveNeighbors(), computeNewGrid(), nextGeneration(), nextGeneration(int), and numOfCommunities().
- DO NOT add any instance variables to the GameOfLive class.
- DO NOT add any public methods to the GameOfLive class.
- YOU MAY add private methods to the GameOfLive class.
- **DO NOT** use System.exit()

VSCode Extensions

You can install VSCode extension packs for Java. Take a look at this tutorial. We suggest:

- Extension Pack for Java
- Project Manager for Java
- Debugger for Java

Importing VSCode Project

1. Download GameOfLife.zip from Autolab Attachments.
2. Unzip the file by double clicking.
3. Open VSCode

  - Import the folder to a workspace through **File** > **Open**

Executing and Debugging

- You can run your program through VSCode or you can use the Terminal to compile and execute. We suggest running through VSCode because it will give you the option to debug.
- How to debug your code
- If you choose the Terminal:
  - first navigate to **GameOfLife** directory/folder
    - to compile: **javac -d bin src/conwaygame/*.java**
    - to execute: **java -cp bin conwaygame.Driver**

Before submission

Collaboration policy. Read our collaboration policy here.

Submitting the assignment. Submit *GameOfLife.java* separately via the web submission system called Autolab. To do this, click the *Assignments* link from the course website; click the *Submit* link for that assignment.

Getting help

If anything is unclear, don't hesitate to drop by office hours or post a question on Piazza.

- Find instructors office hours *here*
- Find tutors office hours on Canvas -> Tutoring
- Find head TAs office hours *here*
- In addition to office hours we have the CAVE (Collaborative Academic Versatile Environment), a community space staffed with lab assistants which are undergraduate students further along the CS major to answer questions.

Problem by Seth Kelly and Maxwell Goldberg

SAS Events

Explore SAS

Departments & Degree-Granting Programs

Other Instructional Programs

Majors & Minors

Research Programs, Centers, & Institutes

International Programs

Division of Life Sciences

Explore CS

We are Hiring!

Research

News

Events

Resources

Search CS

Home

Back to Top