Raj Mohammad – K20126462

Sam thomas – K21065562

CourseWork 3

Life

**Life Forms**

*Mycoplasma*

Mycoplasma is most basic cell in this simulation. It cannot affect other cell types. However, other cells can influence their living state as well as whether they are infected.

When a Mycoplasma is alive, it stays alive depending on the number of alive neighbours it has, but the number changes based on whether the cell is infected. Normally, the cell continues living if it has 2 or 3 neighbours. However, when infected, it only continues living if there are 2 neighbours. Consequently, a dead mycoplasma cell can come back to life at a particular position when there are 3 living neighbours around it.

Another property that the Mycoplasma has is that it can become infected and is shown by a dark-red colour in the simulation. This has a probability of happening when it is neighbouring a cell that can spread a disease (this is either another infected Mycoplasma cell or a harmful cell). Additionally, once an infected Mycoplasma cell dies and comes back to live, it has a 50% probability of staying infected. Lastly, any infected Mycoplasma cell that neighbours a Healer cell is no longer infected.

Key:

Mycoplasma Cell (not infected)

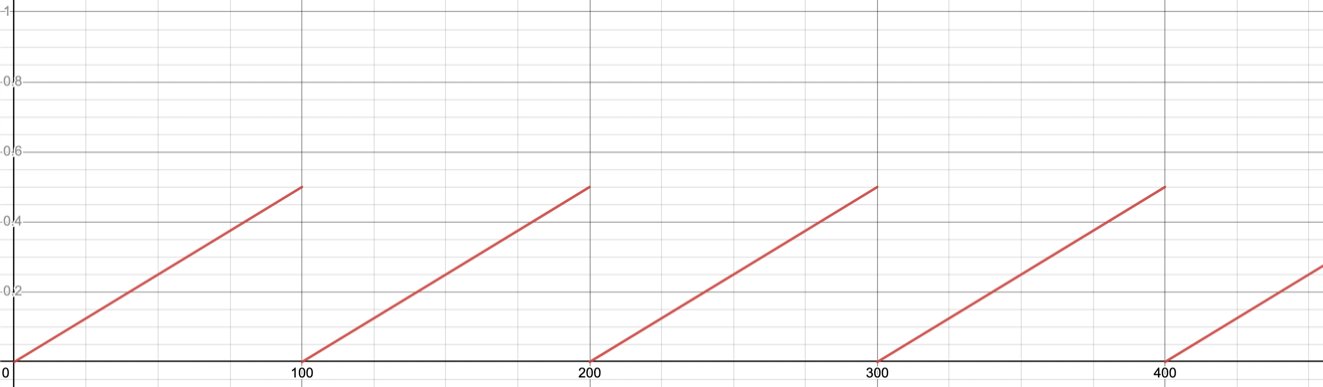
Infected Mycoplasma Cell

*Harmful*

The Harmful cell starts the spread of a disease. It infects Mycoplasma cells which can then spread the disease to other Mycoplasma cells.

To stay alive, the cell needs to have at least 1 living neighbour or be within its life expectancy for each lifecycle. At the start of the program, each harmful cell instance has a predetermined life expectancy which ranges from 50 to 100.

To come back to life, a harmful cell only requires more than 2 alive neighbouring cells.

In order to infect, a check is implemented to see if the number of Harmful cells around a Mycoplasma cell is greater than the number of Protector cells. If so, then there is a possibility of being infected. This depends on the generation count; the graph below represents how the change in generation (x-axis) affects the probability of a Mycoplasma cell being infected (y-axis) under the condition that there are more Harmful than Protector cells surrounding it.

Key:

Harmful Cell

*Protector*

Protector cells prevents the spread of disease by ‘protecting’ uninfected Mycoplasma cells around it from infected Mycoplasma cells or Harmful cells that neighbour. However, they are not always active.

A protector cell can firstly be dead or alive like any other cell. This is determined by their age. Once they reach the age of 100 per lifecycle, they will die. But if there is at least 1 infected or harmful cell around a protector cell, then it will come back to life.

An additional property the protector cell has is whether it is actively protecting which is shown through its colour during the simulation. Initially when the cell comes alive it is in an inactive state, this can be thought similarly to a dormancy period. Then once the cell has an age of 21, it has a probability of 5% to become active from each generation onwards. However, when the cell is active, it fights the infected cells from making uninfected Mycoplasma cells getting the disease. More about the technical implementation is in the Harmful cell section.

Key:

Active Protector Cell

Inactive Protector Cell

*Healer*

Healer cells has the role of making infected Mycoplasma cells around become uninfected.

Like all the other cells, the Healer cell has certain conditions that state whether it stays alive, dies, or comes back to life. To stay alive, the cell must have healed less than the maximum number of heals it can make within its lifetime (this is set to 10 currently), otherwise the cell will die. The healer cell can come back to life if the number of infected cells that neighbour it equal to or greater than 2.

During every generation, Mycoplasma cell checks whether there is a healer cell within its alive neighbouring cells, if so then it becomes uninfected. This is how the healing aspect is implemented.

Key:

Healer Cell

**Challenge Task Implementation**

*Non-deterministic Cells (Protector Cells)*

The protector cells act in a non-deterministic way once it has reached the age of 21 for each of its lives during the simulation. This is because after living for 21 generations, the cell has a 5% probability of becoming active during each generation it is inactive (in which the rule for how it influences Mycoplasma cells can be implemented).

*Symbiosis (Harmful Cells and Mycoplasma)*

The relationship implemented between the Harmful cell and Mycoplasma is a parasitic one. This is because in the relationship, the Harmful cell benefits since it continues living when it has Mycoplasma cells to infect. On the other hand, Mycoplasma suffers because it has a harsher condition for staying life than when it is not infected. (Requires only 2 living neighbours rather than 2 or 3)

*Disease*

There is a probability for Harmful cells to spread disease to Mycoplasma cells. Additionally, this disease can be spread between Mycoplasma cells based on chance.

When a Mycoplasma is infected, the behaviour changes since it has a stricter condition for staying alive. Originally, it required 2 or 3 living neighbours, however after being infected it requires only 2 living neighbours.

*User Interface*

We implemented a GUI to the program allowing the User to have more control over the simulation and allow for better understanding of what is happening in the simulator. The GUI was constructed using a mix of border and gridbag layout. It consists of 5 buttons, multiple labels, a progress bar, and a slider. The buttons are as listed, pause/resume, reset, info, icons and start.

Upon running the simulator, the GUI opens but does not run until start is pressed. Upon creation the pause and reset button are disabled for use until the program starts running. We used runnable threads along with lambdas to allow the start button to function correctly. It also then allowed for the functionality of being able to run the simulator again after finishing by using thread.stop().

The pause/resume button either temporarily halts or resumes the program respectively while reset sets the simulation back to its default state allowing the user to end the simulation early and run another.

Info and Icons buttons both open a new window, showing the basic description of every life form and showing what each colour relates to in the GUI respectively.

The slider allows the user to adjust the speed at which the simulation runs, changing the delay between generations from 0-100.

Finally, we implemented a progress bar which shows the percentage of Mycoplasma that are infected. This was accomplished by adjusting the counter class to track disease count and then adding to the fieldStats class a method to increment said disease count and another method called getDiseasedPercentage() which returns an integer between 1 and 100, which is found by dividing the number of diseased Mycoplasma by the total number of Mycoplasma the progress bar is then set each generation to match this percentage.

**Good Coding Practise**

Throughout the project we prioritised good coding practises. Focusing on creating a readable and maintainable software.

*Coupling*

To assure loose coupling we made sure to pass parameters whenever possible, an example of this is our use of the fieldStats class being used by SimulatorView to access the counter fields instead of calling directly. There are parts of the program with tight coupling but only when necessary, for example Simulator and the SimulatorView class however this is acceptable as one is simply the controller and output for the other making them inherently coupled. Due to our use of super classes for cells we also have coupling between subclass and parents but this we also deemed non important.

*Cohesion*

We focused on creating high levels of cohesion throughout the project by splitting methods down into simple functions which we could break into other helper methods. This allowed for much more readable code. A good example of this would be the harmfullCellCheck() method in the Cell class where the helper methods getHarmfulNeighbours() and getProtectorCellNeighbours() have both been used to make the code much more understandable.

We also made sure to store methods and fields in the relevant classes, one example of this was moving the alive probability of the lifeforms, which were originally stored in the main simulator class, to be instead stored as private fields accessed by get methods within the relevant life form class.

*Responsibility-Driven Design*

Following from cohesion, we focused on responsible design, we did this by assuring all classes handled their own relevant methods. An example of this would be when we work out the percentage of mycoplasma infected, despite being needed in SimulatorView the data needed is found in Counter and handled by FieldStats, it made the most sense therefore for the calculations to be performed within the FieldStats class as that is the class in charge of storing and managing statistics related to the simulation.