Programming project: Disease

Anni Penttilä, 1020539 Bioinformation technology, 2nd year

CS-A1121 Spring 2023 18.5.2023

General description

The program describes how a disease spreads through a population in an abstract space. In the population the individuals are either susceptible to the disease, spreading it, recovered from it and thus gained immunity or deceased and consequently cannot move anymore. Spreaders will infect susceptible people with a certain probability if they have been nearby each other. After catching the disease, a susceptible individual turns into a spreader. The spreader either recovers or will be deceased after a certain time period which depends on the average disease length. The simulation ends when there are no more spreaders left.

The simulation is set up with certain parameters, part of them determined by the user. A terminal user interface is applied for this purpose. Parameters given by the user such as a certain number of characters and a certain number of spreaders, and other predetermined parameters such as the average duration of the disease and the mortality rate will affect how the simulation will proceed. Giving each character also randomly generated age group increases the uniqueness of the project.

The project is implemented at the medium difficulty level. There are also some additional features in it, such as the text scene in the GUI, which displays the current state of the simulation if it's ongoing and and shows the final statistics if the simulation has ended.

Instructions for the user

The program communicates with the user via terminal/console UI since the wanted parameters are numerical and the interaction with the user is simple to implement this way. The user may set several parameters to preferred values.

The program can be started by running the main module. The user

1. Launch the program by running the main module. This can be done by executing the main module using the Python interpreter or by running the main module file.
2. The program will prompt you to enter simulation parameters using the user_input() function. Enter the required information when prompted, such as the name of the disease, the size of the population, the number of spreaders, the average duration of the disease, and the mortality rate of the disease. The simulation parameters ought to be integers.

Here is an example of the user input prompts:

This is a disease simulation.

Enter the name of the disease: Flu

Enter the size of the population: 180

Enter the number of spreaders: 40

Enter the average duration of the disease (in days): 20

Enter the mortality rate of the disease (%): 35

3. After entering the simulation parameters, the simulation will start automatically, and a simulation window will appear.
4. The GUI will display the current state of the simulation, including the distribution of different age groups and the number of characters in each category (spreaders, susceptible, recovered, deceased).
5. The different character types in the simulation are represented by different colors.
6. You can interact with the simulation by pushing the “Next full turn” button. Consequently, each character will take their turn in the simulation. The locations and colours of the characters and the data displayed next to the grid will be updated each time the button is pressed.
7. If the simulation has ended (no more infected people in the population), the final statistics will be displayed, including the number of infected people, the percentage of recovered and deceased among the infected, and the distribution of recovered and deceased by age group.
8. To exit the program, simply close the simulation window.

See also the attached screenshots in the last chapter.

External libraries

The only external library used in this programming project is the PyQt6 library. It is used for the graphical user interface (GUI) implementation which provides a visual representation of the simulation and allows you to interact with it.

Structure of the program

The class diagram shown below is not quite accurate, but it describes the structure and the main idea behind the program. The different character types, Spreader, Susceptible and Recovered, inherit the character AI class (Status in the diagram, Brain in the program). Each character has its own AI which builds on the methods of the character type. Each character type moves in its own specific way. The fourth character type, Deceased doesn’t have its own class since it doesn’t have a brain anymore nor a moving method.

The Character class serves as the parent class for different character types in a simulation. The class provides the basic functionality and attributes common to all character types in the simulation, allowing them to interact with the world and make decisions based on their assigned brains.

The SimulationWorld class represents the simulation world made of a two-dimensional grid in which characters can interact and move. It provides methods to add characters to the world, manage their turns, and perform actions within the simulation. The main purpose of it is to create and manage the simulation world. It allows for the addition of characters, defines the world's dimensions, and handles the turn-based system for characters to take their actions.

The SimulationWorld class uses the Square class to represent the individual squares within the world. It helps in managing the state of each square, such as getting and setting the character in a square and checking whether a square is empty.

The Coordinates and Direction classes provide means of representing and manipulating positions and directions in the simulation world. The Coordinates class deals with representing specific positions on a grid, while the Direction class represents cardinal directions and provides methods to calculate coordinate changes and determine the next direction of a moving character. The Coordinates class can make use of the Direction class to determine coordinate changes when moving in specific directions.

For describing the GUI of the program, the classes used are the GUI and Python's predefined class CharacterGraphicsItem. The former is responsible for creating and managing the graphical user interface for the simulation. It extends the QtWidgets.QMainWindow class and handles the drawing of the simulation world and interaction with the user. The latter handles the visual representation of a character in a simulation and updates the visualization of the character based on its properties.

Some of the central methods:

Character: move(), get_location(), get_neighbour_location(), is_infected(), is_recovered()…

> These methods provide essential functionality for the Character class, including movement, accessing location and neighbor information and retrieving infection status. Methods infect(), cure() and eliminate() are exclusive methods for infected characters. They are used for infecting susceptible characters and changing the infection status of the spreaders. but are implemented also in this class.

Spreader, Susceptible and Recovered: move_body()

> This method is essential in determining the algorithmic moving method of the character type. It provides the direction the character is willing to move. The method also calls the moving method in the parent class, which eventually executes the movement command if it's possible to move to the given direction.

SimulationWorld: __init__(self, width, height), add_character(), next_full_turn(), next_character_turn(), is_end()…

> The init method initializes the simulation world object, which is of prime importance for the whole program. Next_full_turn() advances the simulation by one full turn, updating the state of

all characters in the world according to their brain behavior. Is_end() checks if the simulation has reached its end state.

Square: get_character(), is_empty(), is_square_wall()…

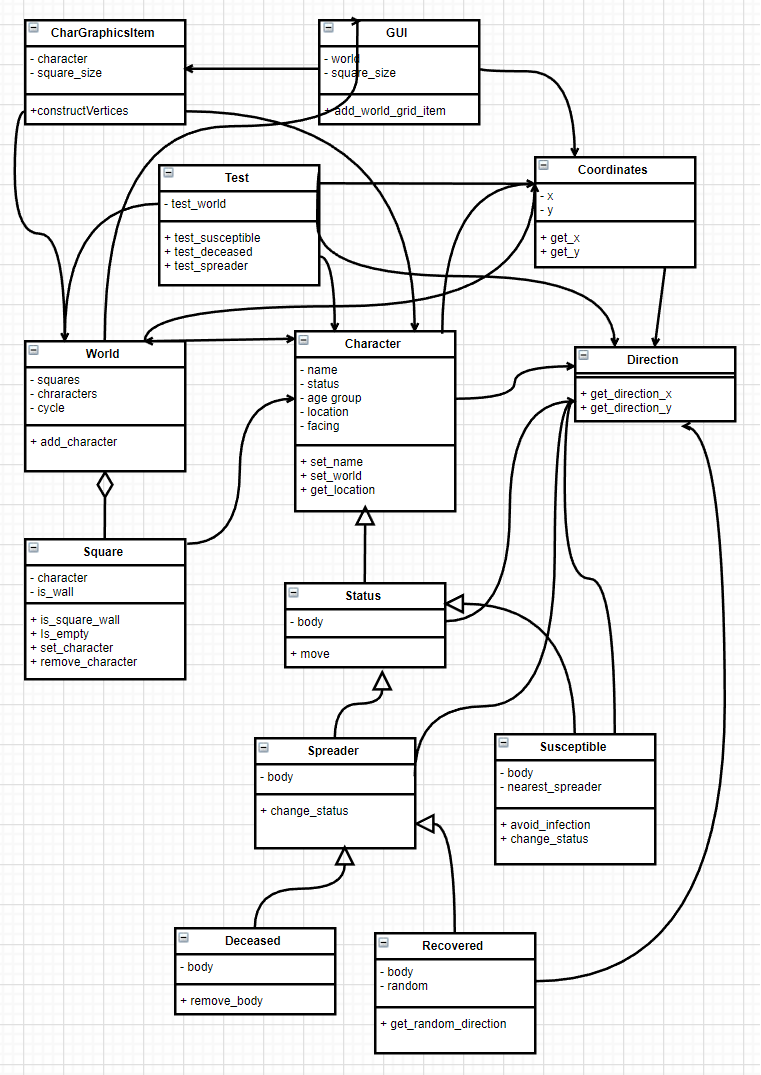
These methods define the current state of each individual square. Each square can contain a character, be empty or be a wall.

Coordinates: get_x(), get_y(), get_neighbor()…

These methods deal with representing specific positions on a grid.

Direction: get_x_step(facing), get_y_step(facing)…

These methods utilize the tuple representation of directions defined in Direction to calculate the coordinate changes. They return the change in the x or y coordinate when moving one step in the given direction.



Algorithms

In this program the mathematical computation regards the movement of the characters and the probabilities of catching the disease and recovering from it. The characters have their own AI, which doesn't base its decisions on randomness unless the individual has recovered from the disease.

The world where the characters move is a two-dimensional area, a grid formed by squares. During one turn in the simulation, the characters will take one step towards the preferred direction. If the square is occupied, the individual tries to find the nearest empty square. If all possible squares are occupied, the character stays in its current position. One step can be taken into the neighboring squares, but not diagonally. Thus, in one turn, if all the surrounding squares are empty, an individual has four possible directions to choose.

Below I've described some of the algorithms behind the character AI's.

Susceptible: The move_body() function of a susceptible attempts to move the character forward by one square in the direction its currently facing. It checks the next square and if it's empty and not a wall, the character moves forward to that square. Otherwise, the function rotates the character clockwise by 90 degrees and checks the next square in the new facing direction. It repeats this process until an empty square is found or it completes a full rotation. If after a full rotation, the character is unable to find an empty square to move into, it remains in its current position.

Spreader: The moving algorithm of the spreader attempts to move the character away from other characters by calculating the distance between the spreader and its neighbour (the nearest character in the simulation). It moves one square at a time, incrementing either the x or y coordinate based on the smaller distance. If the distances are equal, it increments the x coordinate. The spreader only moves one square per turn and turns to face the direction it is moving in. If the target square of the spreader is a wall, it will randomly choose a new direction and tries to move into the next square in that direction. This prevents the spreaders getting stuck at the edges of the grid.

Infection: The spreader may infect a susceptible person in a neighbouring square. This happens with varying probability depending on the age of the target. The probabilities are predefined (0.25, 0.5 and 0.75). This part of the program could have been even more realistic if the probability of infecting a susceptible person would vary with respect to the duration of the infection that the spreader carries.

Getting recovered / deceased: There's an algorithm implemented in the next_character_turn() method (in SimulationWorld) which checks the infection status of a spreader character and applies a probability-based mechanism to determine if they recover, die, or remain infected based on the duration of infection, average disease length, and mortality rate.

For an infected individual, the probability of recovering or getting deceased increases as the simulation proceeds. In the function I've used an approximation for a coefficient that scales the probability of staying infected in each character turn. The approximation builds on the average length of the disease, which is a parameter given by the user.

Recovered: Since recovered individuals are immune to the disease, they don't need to avoid any individuals. The directions of their steps are just based on randomness.

Deceased: Deceased individuals do not take steps in any direction, since they do not have a brain no more. When a spreader turns into deceased, the colour of the character is switched to black.

Data structures

Most of the data in this program is saved just as class parameters which can be used between various classes mainly based on the connections shown in the class diagram. An important data structure for handling the existing data will be lists that can be read and modified between the classes. For example, in the simulation world, the characters / individuals will be stored in an object list. An access to a certain class parameter will be gained calling the specific object methods. Lists are a handy way of storing data in this program, since they are a useful yet simple tool in OOP. The structures are mainly mutable, which is essential for this dynamic simulation, where a large number of the class parameters change systematically.

Files

This program does not handle any external files. If the program were improved, it could write an external file, where it would record various data about simulations that were run (e.g., how many caught the disease / recovered from it / died from it on each turn, a moving average for the number of cases on each turn etc.

Testing

Testing the program is not its strongest feature. Below I have mentioned some test cases and related tests I could implement to the Test class.

Testing the validity of the input data is important when it comes to testing the functionality of the UI. The try-except blocks and if-clauses are implemented in the user_input() function to tackle these issues.

Some essentials things this program should be able to do flawlessly are storing the data correctly and making sure that the UI works as intended and displays the right information to the user so that the user experience is convenient. These functions could be tested a bit more carefully by comparing expected and actual values and asserting equality.

Since the fundamental purpose of the program is to simulate the spread of a virus in a population, the Coordinates and Direction classes should work properly. It would be important to test whether all the different character types move according to the wanted logic. Some additional tests in the Test class could be useful for this purpose. Another essential thing is that the change of a character's status works problem-free. This could be tested by comparing the expected and actual statuses of concerned characters in different situations.

The functionality of the mathematical calculations was tested during the programming project using print command, running the simulation and doing observations.

The known shortcomings and flaws in the program

- The testing of the program is quite weak. Despite that, the simulation seems to work as intended if the parameters given by the user are reasonable. This could also be investigated with further testing.
- The moving algorithm of the spreader is not too complicated, but it requires calling five methods in two different classes to execute the movement. This maybe could have been done in a bit simpler and more effective manner, maybe using less but more thoughtful methods.
- The program is missing some comments and documentation which would improve the readability of the code.

3 best and 3 worst areas

I think one of the most successful features is the realistic progression of the simulation in terms of people getting recovered and deceased. This is due to including the increase in sick days, as well as the average disease duration and mortality rate in the calculation of the infection status probabilities. Thus, for each infected individual, the probabilities change each turn as the simulation proceeds.

Another good feature is the interactivity of the program and the text scene in the GUI, which displays the current state of the simulation when it's ongoing and shows the final statistics if the simulation has ended. The user has a very large influence on what the disease simulation will look like. Interactivity adds to the interest and convenience of the program.

The third good feature is that the program is also easy to customize and develop further because of its basic structure. It provides a good basis for more complex simulations. For example, the randomness-based age grouping is an adequate addition, but I could have used a bit more time with the implementation. Age could also affect the probabilities of getting recovered / deceased. If I continue the project, this is one property I could implement.

Some of the weaknesses I have already mentioned in the chapters above. The testing of the program is weak. I my opinion the cross-use of the class parameters/attributes and their call methods is a bit clumsy way to handle variables and mutable data structures between classes. I guess this could have been done in a more consistent way. Some of the moving algorithms are also quite simplistic. They could be made more interesting by making new modifications, for example by adding a "super spreader" that always moves towards the nearest character.

Changes to the original plan

The implementation of the program is slightly simpler than in the original plan. This was due to problems with my own time management. The area around the spreader is not divided into two infectious zones. Only the four neighbouring squares form the infectious zone. Also, there's no 'nurse' character that was part of the original plan.

On the other hand, the text scene in the GUI was not a part of the original plan either. It makes the program easier to understand and hence the simulation may be a bit more interesting to follow.

When it comes to the schedule, I was busier than I expected in the spring and therefore didn't have as much time to spend on the program (at least not within the primary schedule) as I would have liked. The extra time was really useful, and I managed to put together what I think is a reasonably good project.

Realized order and scheduled

The basic structure of this program is based on the robot world program. From this I continued by editing the character type classes and the predefined methods of Robotworld's classes to better fit my own simulation. Then I created more methods, mainly related to the moving algorithms and the behaviour of the characters. Towards the end of the project, I created a

function that processes user input and creates a simulation world based on it. The programming order was as planned.

Assessment of the final result

The strongest parts of the program are its interactivity, the realistic simulations it provides and the possibility of further development. The weak parts are its unnecessary complexity in relation to the outcome and its clumsiness when moving between classes.

The complexity level of the project might not require so many classes and methods. The various character types could probably be implemented by means of truth values and by modifying the Character class attributes. However, it makes the program's structure suitable for making changes and expanding in the future.

The quality of the program is quite good, the code is mostly well commented and is quite clearly readable. Some of the moving methods could have been somewhat better documented and more consistently implemented.

The multi-class moving methods affect the effectiveness of the program. Some of the solution methods for the probability calculations could also be reconsidered. There could be a formula for calculating the probability of a spreader infecting characters nearby taking the current duration of the infected person's disease into consideration. More tests could be added to assess the adequate functioning of the program.

References

AgentPy – Agent based modeling in Python - https://agentpy.readthedocs.io/en/latest/

Building simulations in Python - https://towardsdatascience.com/building-simulations-in-python-a-complete-walkthrough-3965b2d3ede0

Virus spread simulation - https://agentpy.readthedocs.io/en/latest/agentpy_virus_spread.html

Qt Documentation for Python - https://doc.qt.io/qtforpython-6/

QPolygon - https://doc.qt.io/qtforpython-6/PySide6/QtGui/QPolygon.html

Layout Management in PyQt6, CodersLegacy - Layout Management in PyQt6

Attachments

Here is an example of the user input.

```
This is a disease simulation.
Enter the name of the disease: Flu
Enter the size of the population: 170
Enter the number of spreaders: 44
Enter the average duration of the disease (in days): 20
Enter the mortality rate of the disease (%): 25


The probability of getting infected (when in contact with a spreader) varies between different age groups.
The age groups are:
YOUNG: 0-17 (25% infection rate)
ADULTS: 18-64 (50% infection rate)
ELDERLY: 65+ (75% infection rate)
In the simulation the spreaders are red, susceptible people are yellow, recovered green and deceased black.
```

The graphical user interface is shown below. The user can interact with it by pushing the “Next full turn” button.