NaCo 22/23 assignment 2 report, group 11

Wouter Ykema, Yvo Hu

Leiden Institute of Advanced Computer Science, The Netherlands

1 Introduction

In this paper, we will implement a model for cellular automata in python, and complement it with a genetic algorithm to try and solve the inverting problem. The common goal of this problem is to find an unknown starting state C^0, given a known final state C^T on which T time steps have occurred.
To calculate this unknown state, we make use of an objective function to evaluate our current solution (upon which our CA will run with a given rule with T time steps), and compare it with the given final state C^T.

A genetic algorithm from Holland (1975) [1] will help us with modifying our existing solution using mutations to find the best possible solution. This genetic algorithm will make use of an objective function to calculate the similarity between the currently calculated $C^{'}$ and the given C^T . This will be the measure upon which our genetic algorithm decides which mutations to the current best solution are the best to create an even better solution.

We will experiment with different kinds of objective function and find out which one is the best for our testcases. We will create fixed-target plots and fixed-budget plots of the results and conclude which objective function performs the best.

2 Problem Cellular Automata

In this section we'll describe the implementation of a basic 1-dimensional Cellular Automata in python of size N (non-connected boundaries, fixed to 0), which can follow a transition rule ϕ, with a neighbourhood size r = 1. Additionally the number of possible values k for each cell in our CA may be either 2 or 3.

We first define a general CellularAutomata class, from which to instantiate our CA. The instance requires an argument rule_number, which configures what wolfram rule we'll use for this specific instance, and an optional argument k for the number of possible values per cell (defaulted to 2).

The argument rule_number will be passed down to a secondary function getRuleSetDict(), which will initialise a dictionary with all transitional states with r neighbours. We calculate how many unique dict values are needed for a given k and r. The numbers will serve as keys in this dictionary, with a number between 0 and k as value.

After instantiating this class,we initialise the solution with C^0 as the starting state. We may loop through it T times, and generate a new C' after each iteration based on the configured rule and previous C'. This C' will ultimately be compared with the given C^T using an objective function. The starting state C^0 will be repeatedly modified by the genetic algorithm based on the value of the objective function, to approach the solution to the inverting problem.

3 Algorithm Description

A genetic algorithm uses the concept of a *population* to solve the problem. At each iteration, the fitness function is evaluated on each *individual* of the population. In our case, the individuals are starting states of the cellular automaton, and the fitness function is a similarity measure, as explained in section 4. In general, on each iteration, the following steps are performed.

step 1. Create the next generation using the concept of *mating selection*. This consists of selecting parents out of the population and performing recombination on them to create offspring. The amount of offspring is usually larger than the original population size.

step 2. Apply *variation* to each of the individuals. Small changes to the individuals increase the diversity of the population.

step 3. Evaluate the fitness function on each individual and store the resulting fitness values.

step 4. Apply the concept of *environmental selection* according to the fitness values, similar to how *natural selection* works in nature. The amount of individuals to keep is equal to the original population size.

step 5. Stop when the *termination criteria* are met. Otherwise, go back to step 1.

The termination criteria can include a fixed *budget* of allowed function evaluations, or some fixed *target* fitness to be reached.

Notation Let M denote the original population size, and let N be the amount of offspring to create during mating selection. Let n be the dimensionality of the search space, which is the length of the starting state. Each cellular state $\boldsymbol{x}$ will be denoted by $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$, where each $x_k \in \{0, 1\}$ for $k \in \{1, \dots, n\}$. Let $f : \mathbb{Z}^n \to \mathbb{R}$ be the fitness function as explained in section 4, sending all states to real numbers measuring how close they are to the target.

3.1 Applying it to the problem

We have used the following operators:

◇ Two-parent uniform crossover, with exponential parent-selection.[1]

◇ Changed version of flip-bit mutation. This new version adds a random number as mutation modulo the base (so that it works now for all bases) instead of toggling the bit, (which only works properly in base 2).[2]

◇ Truncation selection works just fine.

◇ The termination criteria are whether the optimum has been found or the number of evaluations exceeds the budget.

◇ The population initialization is uniformly random.

[1] described in the previous report [3].

[2] It also works for other bases, but the mutated loci would only ever get 0 or 1, which means that there need to be enough 2's in the initial population for this to be succesful in base 3, for example.

The only thing we really needed to change was the mutation. The rest of the operators we had are all still fine for this new problem. Luckily it was very easy to just add a new factory function to our code and use it in the algorithm, because the code is very flexible and modular.

4 Solving the Inverting Problem

In this section, we solve the so-called inverting problem using a genetic algorithm.

4.1 Problem statement

The cellular automaton calculates C', which is the output state after T time steps, for a given input state C^0. The goal of the inverting problem is to find C^0 given C^T.

4.2 Problem solution

We use our genetic algorithm [3] to solve this problem.

For this, we have written some new objective functions, that, given a configuration of the cellular automaton, calculate the similarity between an input state as suggested by the genetic algorithm and the actual input state of the cellular automaton.

We first tried using the levenshtein similarity measure and the hamming similarity measure. The levenshtein measure does so by counting the number of times you would have to insert, delete or substitute a character from string 1 to make it like string 2. The hamming measure is the number of positions at which the corresponding characters are different. The following experiments were performed.

4.3 Experiments

In this section we will perform various experiments, and compare the CA with different configurations of the GA. We will give a detailed description of each of these experiments and observe the results.
Each of the experiments will contain 2 instances, and each of these instances will process a CSV file containing 10 different configurations. Each configuration will be subsequently be run 5 times and averaged for us to have a more accurate generalization, and rely less on the randomness of singular runs.

Specifically, the fixed-target and fixed-budget metrics will be evaluated in these experiments.
The fixed-budget range is 0 - 10000

The fixed-target range is 0.32 - 1

The outcomes of the experiments are plotted using IOH-Experimenter.

Experiment 1 This experiment compares the fixed-budget and fixed-target metrics between a CA configured with the Hamming similarity measure, and one with the Levenshtein similarity measure.

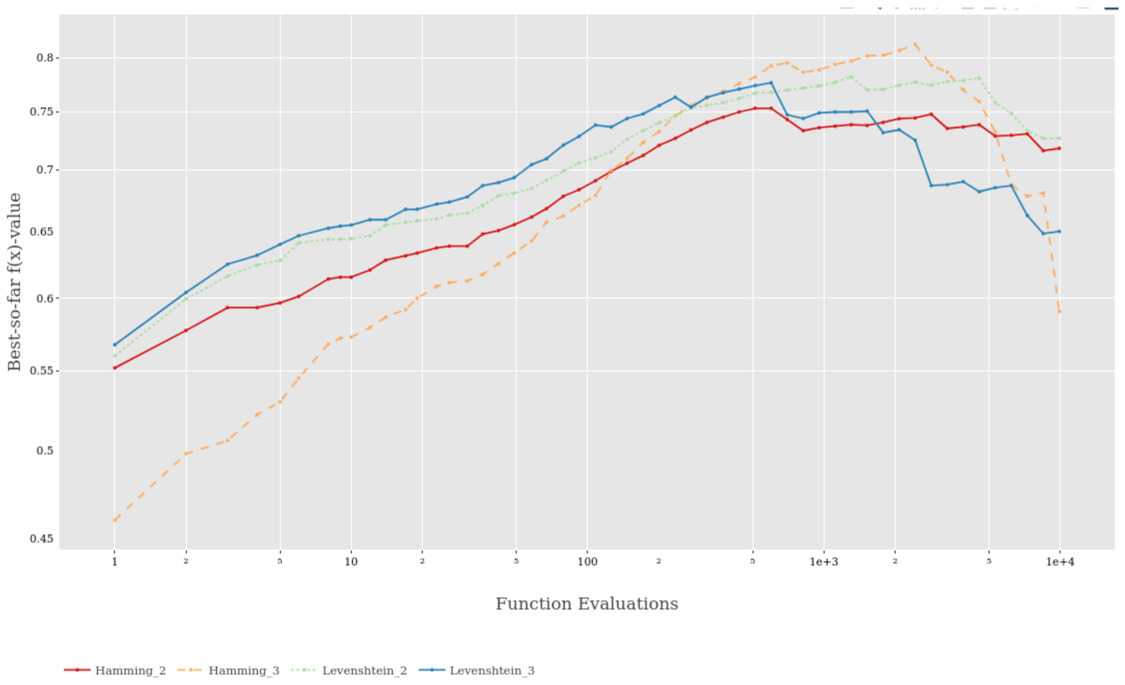


Fig. 1

From figures 1 and 2 we see the following. Firstly, we would expect both plots to be non-decreasing, but the first one does not. We think that's weird. In the second plot we see that Hamming uses more function evaluations to get to a best-so far value in the range 0.4 to about 0.6. But then Levenshtein uses more evaluations if the target is very close to 1.

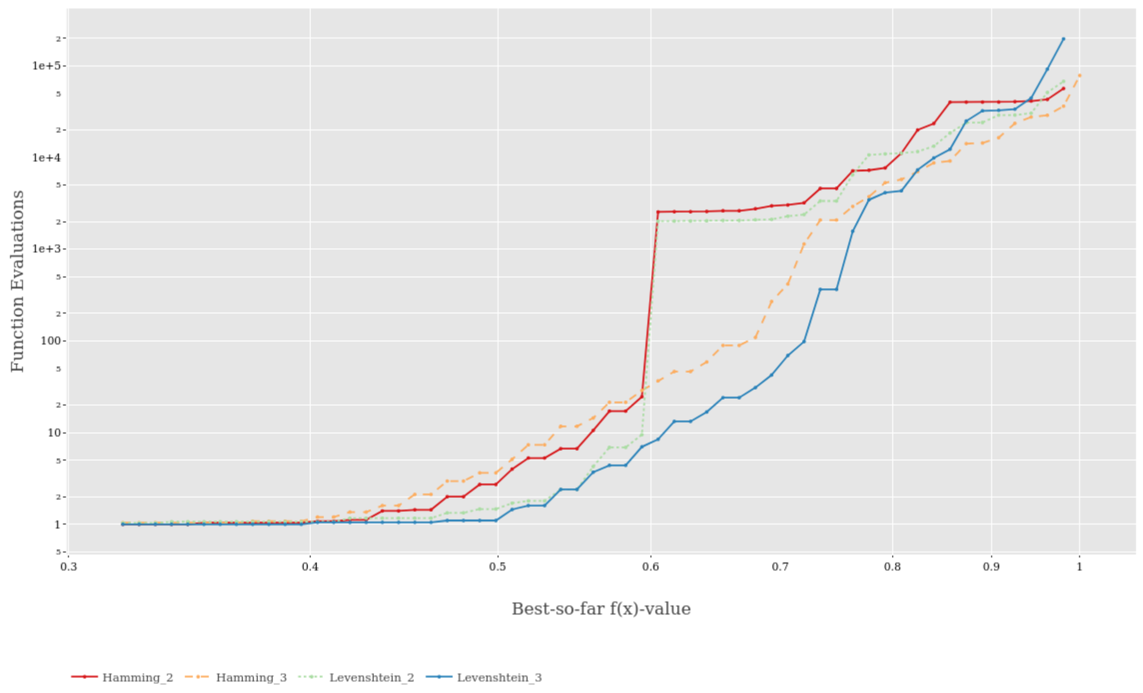

Function Evaluations
2
1e+5
5
2
1e+4
5
2
1e+3
5
2
100
5
2
10
5
2
1
5
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Best-so-far f(x)-value
Hamming_2
Hamming_3
Levenshtein_2
Levenshtein_3

Fig. 2

Experiment 2 This experiment compares the fixed-budget and fixed-target metrics of a CA configured with the Hamming similarity measure with differing GA's. One which uses flipbit mutation, uniform crossover recombination and exponential parent selection operators, and one which uses a random search algorithm.

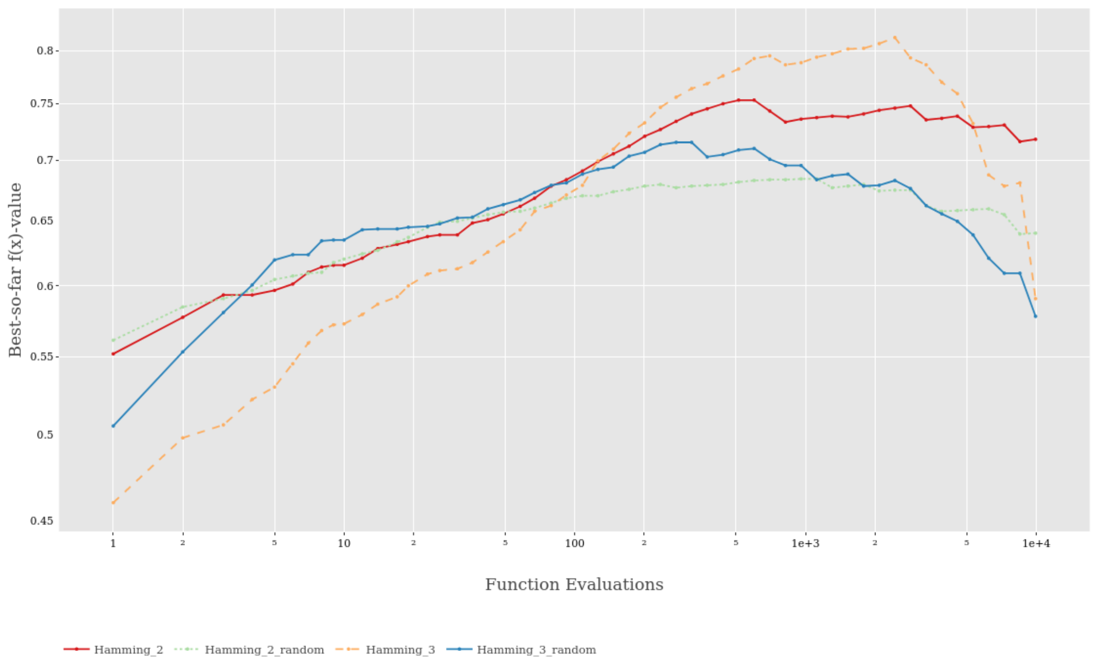


Fig. 3

In figure 3 and 4 we see how our algorithm performs against a random search, both using the Hamming objective function. It is very clear that our algorithm performs better than random search. This can be seen in figure 3 because the green and blue lines have a lower function value, and it can be seen in figure 4 because the green and blue lines use more function evaluations (for the same fixed target) than the red and yellow lines. Of course the different lines overtake (and catch up to) each other at several different points, so the conclusion should not be relied upon too heavily.

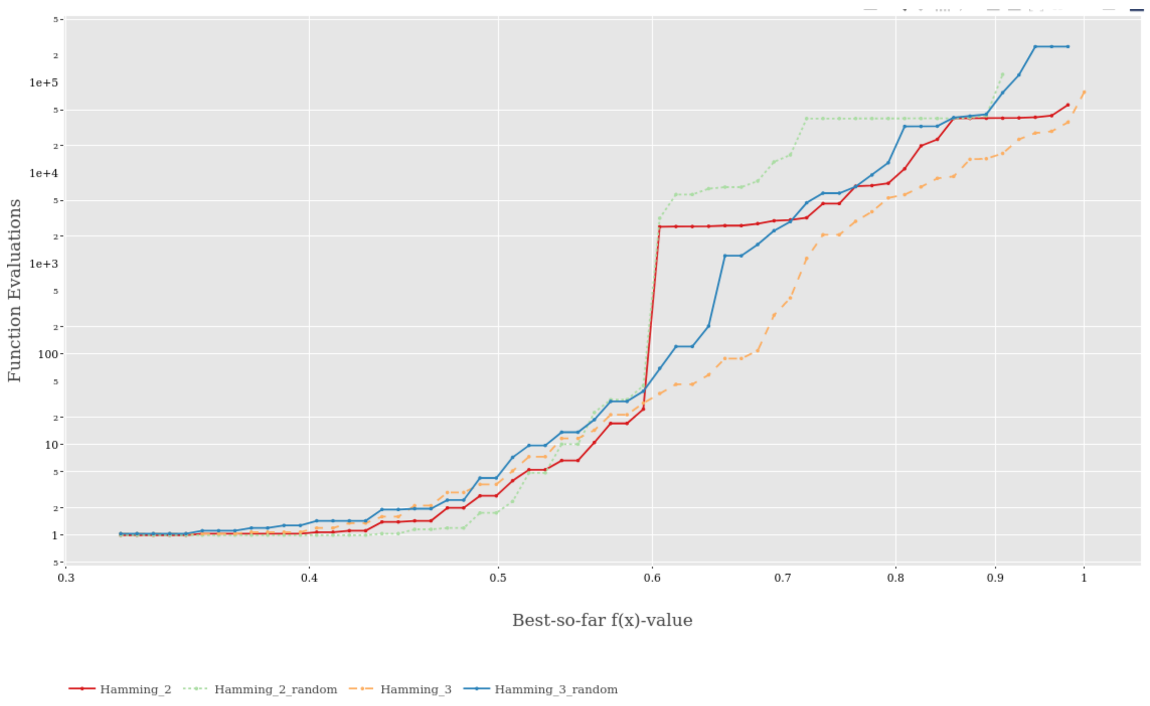

Function Evaluations
Best-so-far f(x)-value
Hamming_2
Hamming_2_random
Hamming_3
Hamming_3_random
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1e+5
1e+4
1e+3
100
10
1

Fig. 4

Experiment 3 This experiment compares the fixed-budget and fixed-target metrics of a CA configured with the Levenshtein similarity measure with differing GA's. These GA's differ in the amount of mutation operators they utilize. One only uses the flipbit mutation operator, another uses both the flipbit and swap mutation operator

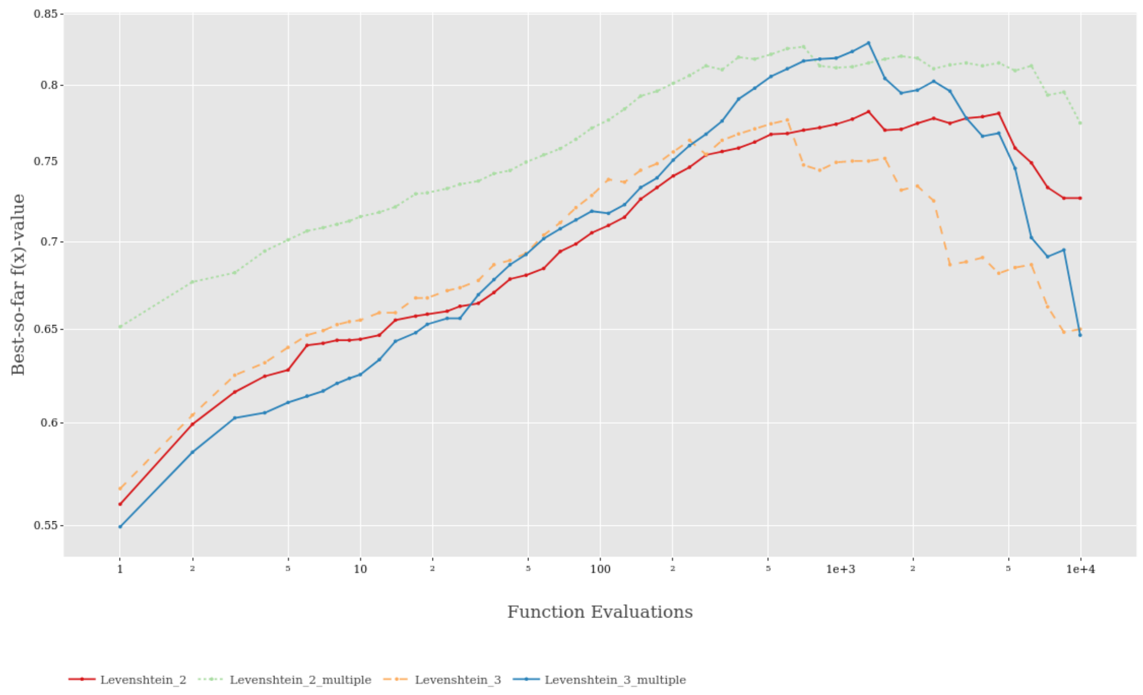
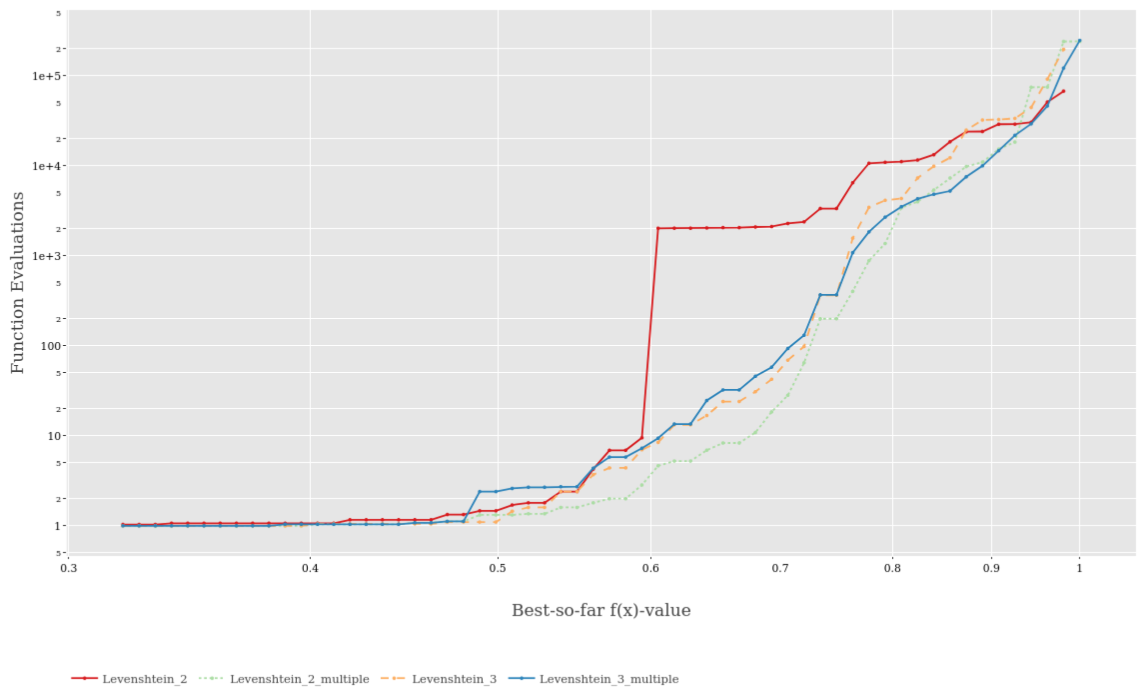


Fig. 5

In figures 5 and 6 we see the effect of the different mutation operators, this time using the Levenshtein objective function. In figure 5 we see that using multiple mutation operators works better for base 2 problems than using only flipbit. For base 3 problems we see that multiple mutation works better in the end, while only using flipbit works better in the beginning. The same can conclusion can be taken from figure 6.



Function Evaluations
Best-so-far f(x)-value
1e+5
1e+4
1e+3
100
10
1
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Levenshtein_2
Levenshtein_2_multiple
Levenshtein_3
Levenshtein_3_multiple

Fig. 6

5 Application of CA in practice

In this section, we'll be discussing a paper called "Modeling epidemics using cellular automata" [2].

5.1 Field and context

The main goal of this work is to introduce a theoretical model based on cellular automata, to simulate the spread of epidemics.
Whilst a single infected host might not be significant, a disease that spreads through a large population yields serious health and economic threats.

Since the first years of the last century, an interdisciplinary effort to study the spreading of a disease in a social system has been made.
In this sense, mathematical epidemiology is concerned with modeling the spread of infectious disease in a population. The aim is generally to understand the time course of the disease with the goal of controlling its spread. Such models are used, for example, to guide policy in vaccination strategies for childhood diseases. The proposed model can serve as a basis for the development of other algorithms to simulate real epidemics based on real data.

5.2 Method and alternatives

Traditionally, the majority of existing mathematical models to simulate epidemics are based on ordinary differential equations. These models have serious drawbacks in that they neglect the local characteristics of the spreading process and they do not include variable susceptibility of individuals. Specifically, they fail to simulate in a proper way (1) the individual contact processes, (2) the effects of individual behaviour, (3) the spatial aspects of the epidemic spreading, and (4) the effects of mixing patterns of the individuals.

Cellular automata can overcome these drawbacks and have been used by several researches as an efficient alternative method to simulate epidemic spreading. These are simple models of computation capable to simulate physical, biological or environmental complex phenomena. Several models based on such mathematical objects have been appeared in the literature to simulate growth processes, reaction-diffusion systems, self-reproduction models, epidemic models, forest fire spreading, image processing algorithms, et cetera, thus proving its efficacy in various complex systems.

5.3 Type of CA used

In this section, we introduce the mathematical model based on cellular automata, to simulate the spreading of a general epidemic. It is suppose that the ground where the epidemic is spreading stands for the cellular space of the CA, and it is divided into identical square areas, each of them represent a cell of the CA. Different cells will have different populations: differing densities and different 'across cell' traversal or mobility properties. Moreover, the cellular space is considered to be large enough to ensure that the epidemic spreading affects only to the central region.

Let $S_{ij}^t \in [0,1]$ be the portion of the healthy individuals of the cell (i,j) who are susceptible to infection at time t; set $I_{ij}^t \in [0,1]$ the portion of the infected population of the cell at time t who can transmit the disease to the healthy ones; and let $R^t_{ij} \in [0,1]$ be the portion of recovered individuals of (i,j) from the disease at time t, that will be permanently immunised. As is stated above, the population of each cell is constant, consequently: $S^t_{ij} + I^t_{ij} + R^t_{ij} = 1$.

These are more precisely defined as the following:

Dimensionality: Two-dimensional

States: Susceptible, Infected and Recovered defined as

$$\left(DS^t_{ij}, DI^t_{ij}, DR^t_{ij}\right) \in Q.$$

where D denotes a discretized variant.

Transition rules:

$$\begin{aligned}
I^t_{ij} &= (1-\varepsilon)\cdot I^{t-1}_{ij} + v\cdot S^{t-1}_{ij}\cdot I^{t-1}_{ij} + S^{t-1}_{ij}\cdot \sum_{(\alpha,\beta)\in V'} \frac{N_{i+\alpha,j+\beta}}{N_{ij}}\cdot \mu^{(i,j)}_{\alpha\beta}\cdot I^{t-1}_{i+\alpha,j+\beta},\\
S^t_{ij} &= S^{t-1}_{ij} - v\cdot S^{t-1}_{ij}\cdot I^{t-1}_{ij} - S^{t-1}_{ij}\cdot \sum_{(\alpha,\beta)\in V'} \cdot\frac{N_{i+\alpha j+\beta}}{N_{ij}}\cdot \mu^{(i,j)}_{\alpha\beta}\cdot I^{t-1}_{i+\alpha,j+\beta},\\
R^t_{ij} &= R^{t-1}_{ij} + \varepsilon\cdot I^{t-1}_{ij}.
\end{aligned}$$

5.4 Interpreting the results

The main characteristic of this model is the definition of the state of each cell as a three-uplet formed by a suitable discretization portion of its population which is susceptible, infected and recovered at each time step, together with the definition of the local transition function involving these parameters.

The simulations obtained using artificially chosen parameters seem to be in agreement with the expected behaviour of a real epidemic. The proposed model can serve as a basis for the development of another algorithms to simulate real epidemics. Further work aimed at testing its performance against real data. Obviously, in real simulations one has to take care with the scale and an appropriate size of the cells must be used in order to obtain an efficient simulation.

5.5 Our opinion on their approach

As was explained in section 5.2 the majority of existing mathematical models to simulate epidemics are based on ordinary differential equations which have serious drawbacks in that they neglect the local characteristics of the spreading process and they do not include variable susceptibility of individuals. This new approach using cellular automata is most appropriate to counter these drawbacks. We are now able to actually simulate these conditions and include them in the model.

5.6 Improve on their setup

The proposed model presents some shortcomings, like the non local effect, which are crucial to model SARS, Foot and mouth disease and Aviation Flu, and seasonality effects which are crucial to model measles.

6 Conclusion

We have learned how to create our own one dimensional Cellular Automata consisting of a static neighbourhood radius, and a varying amount of state values. By implementing these functions ourselves, we have gained an in depth understanding of the processes behind a CA.

From our experiments we can conclude the following.

1. Using the Hamming objective function, the number of evaluations you need to reach a function value of 1 is lower than for the Levenshtein objective function.
2. Our genetic algorithm performs better on the testcases than random search does.
3. Multiple mutation operators work better than only flipbit mutation.

That last sentence is probably true because random mutation is just better than flipbit mutation here, although multiple mutation operators made for a more interesting experiment.

Cellular automata can be used in all kinds of models. In the paper we reviewed, it is used for simulating the spread of epidemics, and it works better than the model using traditional ordinary differential equations, in this case.

A Appendix

References

1. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor, MI (1975), second edition, 1992
2. White, S.H., del Rey, A.M., Sánchez, G.R.: Modeling epidemics using cellular automata. Applied Mathematics and Computation **186**(1), 193–202 (2007). https://doi.org/https://doi.org/10.1016/j.amc.2006.06.126, https://www.sciencedirect.com/science/article/pii/S0096300306009295
3. Wouter, Yvo, B.: Naco 22/23 assignment 1 report, group 11 (2022)