Programming and Algorithms

Evolving Discrete Cellular Automata with Genetic Algorithms

Gautham GANESH

Submitted on 27 January 2022

Introduction

1 One-dimensional cellular automata

1.1 Transition rule

A basic neighbour-sum transition rule was used for the one-dimensional CA; the state of the cell in the next time step would be entirely dependent upon the number of neighbours that were ON in the current state. Since each cell has two immediate neighbours, the possible values for the neighbour sum would be 0, 1, and 2 (neighbour-sum $n_{sum} = n+1$).

This transition rule was represented using an array of the form $\begin{pmatrix}1 & 1 & 1\end{pmatrix}$, in which the the indices and the elements of the array correspond to the neighbour sum in the current generation and the state of the current cell in the next generation respectively. A population of these rules were used in the genetic algorithm to evolve the CA and solve the majority problem.

Thus, the function that defines the transition rule was simple - it provided the state of the current cell in the following generation by returning `rule[neighbours]`.

1.2 The majority problem

A grid space of one hundred cells was initialised using a NumPy array in the OFF state. Ten cells in the centre were then randomly assigned either of the two binary states. This resulted in an array of the form $\begin{pmatrix}0 & 0 & ... & 1 & 0 & 1 & 1 & ... & 0 & 0\end{pmatrix}_{100}$.

1.2.1 Genetic algorithm

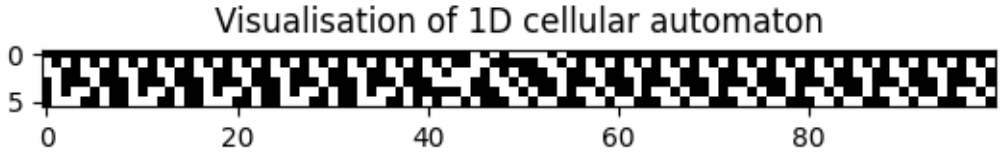
Initialisation of the population A population of eight rule arrays of size three were initialised randomly as chromosomes using the NumPy module.

Calculation of fitness The fitness for each chromosome was calculated by summing the ON states of the grid using `calculate_fitness1(population, iterations, size, space)` since a more sophisticated function was not required for the convergence of the genetic algorithm to a solution.

Pool selection, crossover and mutation The top six fittest individuals were picked to be parents that were carried on to the following generation. Out of these six individuals, two offspring were produced by combining halves of the i^{th} and $(n-i)^{th}$ chromosomes, *i.e.*, a process equivalent to selection and crossing over between the fittest and least fit individuals in the parent set (`crossover1(parents, number)`). Each of these offspring were also potentially subject to a process of mutation using the function `mutation1(offspring)`; in this case, if the probability of a trial $p_{mut} > 0.9$, then a switch mutation (*i.e.*, 0 to 1 and vice versa) would occur in a randomly-selected bit of the array. The parents and resultant offspring were then mixed to produce a new population of chromosomes.

1.2.2 Results

The experiment converged to a solution within one to six generations almost every time because of the simple transition rule space and the dimensionality of the grid. The most commonly occurring solution was $\begin{pmatrix}1 & 0 & 1\end{pmatrix}$ closely followed by $\begin{pmatrix}1 & 0 & 0\end{pmatrix}$. Sometimes, depending upon the initial set of randomised states, a solution wasn't possible; for instance, if there are no cells that are ON in the initial grid, then the goal cannot be achieved.



2 Two-dimensional cellular automata

2.1 Transition rules

Two different transition functions were used for different problems in the two-dimensional CA. For the 50-50 problem, populations of an eight neighbour-sum rule were evolved. This rule was represented using a similar array as that of the one-dimensional CA; however, it consisted of nine elements $\begin{pmatrix}0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0\end{pmatrix}$ for eight possible neighbours instead. Since $n_{sum} = n+1$, the neighbour-sum could take on values between 0 and 9. Therefore, the transition function described for the one-dimensional CA `rule[neighbours]` was used here as well.

On the other hand, two fixed non-evolving rules were used for the directed dynamics and maze-like layout generator problems. These transition rules were two-dimensional and were represented using an array of the form

$$\begin{pmatrix}0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0\\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1\end{pmatrix}$$

Here, each row index corresponds to the state of the cell in the current generation and the one-dimensional array at these indices corresponds to the neighbour-sum rule. Thus, the elements of the first row ($row = 0$) would provide the state of the cell in the next generation if it was OFF (*i. e.*, 0) in the current generation. Similarly, the elements of the second row ($row = 1$) would provide the state of the cell in the next generation if it was ON (*i. e.*, 1) in the current generation. The transition function `transition2(neighbours, rule, current)` used for these types of rules returned `rule[current][neighbours]`.

2.2 The 50-50 problem

A grid space of size 50×50 was initialised using a NumPy array in the OFF state; a 10×10 block of cells in the centre were then randomly assigned either of the two binary states.

2.2.1 Genetic algorithm

Initialisation of the population A population of eighty rule arrays of size nine were initialised randomly as chromosomes using the NumPy module.

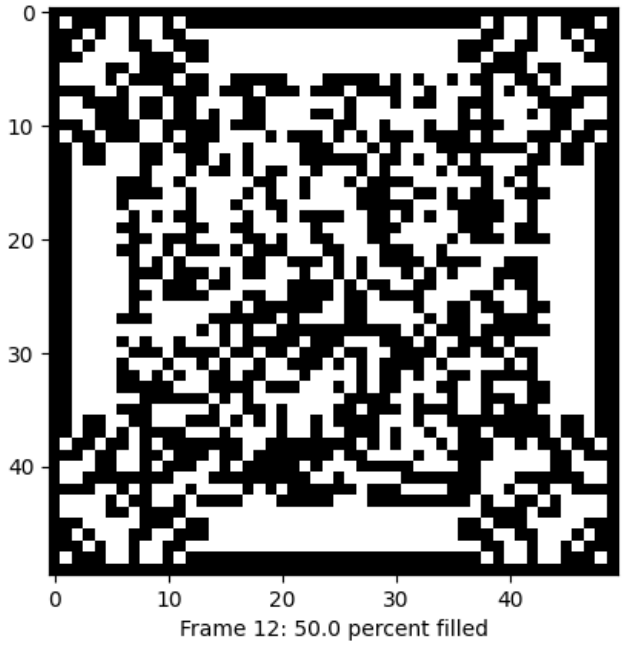
Calculation of fitness The fitness for each chromosome in the population was calculated using the `func_normal(x, sd, mean)` function, which reproduces the bell-shaped curve that commonly occurs in normal distributions, albeit in a much larger scale. This was done to accommodate the large fitness landscape that was required to evaluate each chromosome accurately. The function used was

$$f(x) = 10^3 \frac{\exp(-0.5(\frac{x-\mu}{s})^2)}{\sqrt{2\pi}}$$

where x was the number of cells switched ON, μ was the number of cells required to be ON in the target grid space and s was the standard deviation of the distribution.

Pool selection, crossover and mutation The top sixty-four fittest individuals (called parents) were picked to be carried on to the following generation. Out of these sixty-four individuals, sixteen offspring were produced by crossing over using `crossover1(parents, number)` in the parent set. Each of these offspring were also subject to a process of mutation using the function `mutation1(offspring)`. This was the same function as the one used in the one-dimensional CA. Therefore, a mutation would occur in a randomly-selected bit of the array when $p_{mut} > 0.9$. The parents and resultant offspring were then mixed to produce a new population of chromosomes.

2.2.2 Results



2.3 Directing dynamics in the Game of Life

A grid space of size 40×40 was initialised using a NumPy array with all cells in the OFF state. The objective of this experiment was to discover a set of initial states that would drift towards a random target (or target cells) in John Conway's Game of Life. This is quite different from other experiments in this project, where transition rules were discovered instead.

In the Game of Life, the rules are as follows -

- If a cell is ON, then it shall remain switched on in the next generation only if it has two or three neighbours. Otherwise, it switches off.
- If a cell is OFF, then it shall switch on in the next generation only if it has three neighbours. Otherwise, it shall remain switched off.

The above rules are represented using a specific two-dimensional array -

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

These rules utilise the transition function `transition2(neighbours, rule, current)` to help discover a solution in the following experiment.

2.3.1 Genetic algorithm

Initialisation of the population A population of one hundred 4×4 blocks of cells were initialised randomly as chromosomes using the NumPy module. These blocks correspond to the many sets of initial states that are possible (2^{16} possibilities for this particular block).

Calculation of fitness The fitness for each chromosome in the population was calculated using the `gradient(size, tgt)` function. This function is given by

$$f(x) = \frac{1}{\exp(0.8\sqrt{(i - t_x)^2 + (j - t_y)^2} - 3)}$$

where i, j were the horizontal and vertical component (or indices) of any cell respectively and t_x, t_y were the same components for the target cell respectively.

Pool selection, crossover and mutation The top eighty-four fittest individuals (called parents) were picked to be carried on to the following generation. Out of these eighty-four individuals, sixteen offspring were produced by combining three rows of the i^{th} chromosome and a row of the $(n - i)^{th}$ chromosomes, *i.e.*, a fairly skewed crossing over between the fittest and least fit individuals in favour of the former in the parent set (`crossover2(parents, number)`). Each of these offspring were, as in previous experiments, subject to a process of mutation using the function `mutation2(offspring)`. In this function, if the probability of a trial $p_{mut} > 0.95$, then a switch mutation would occur in a randomly-selected bit of the array. The parents and resultant offspring were then stacked together to produce a new population of chromosomes.

2.3.2 Results

This experiment produced correct results and reached the target cell(s) in only about one out of every ten CA simulations. In most other simulations, the genetic algorithm failed to converge to a solution even after one hundred generations. This could be attributed to the highly-specific nature of the problem, an insufficient or an unsuitable fitness function, and the huge set of initial spaces. A few different fitness functions were explored for this experiment and the best results were produced by the one reported here.

2.4 Generating maze-like layouts