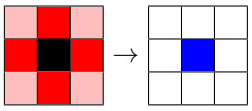
Cellular Automata Program - User Guide (WIP)

Alex Richardson

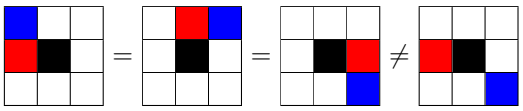
February 1, 2019

1 What is a Cellular Automaton

The Cellular Automata (CA) rules generated and implemented by this program are a generalised version of Conway's Game of Life, with arbitrary numbers of states, and variable cell neighbourhood size. A rule is a mapping between an arrangement of cell states at one time step, to a cell state at the next time step:



Here the state of the blue cell at t_n is determined by the states of all the neighbouring cells (and the state of the blue cell itself) at t_{n-1}. The Rule of a CA is the function that maps all possible permutations of cell states for a given neighbourhood at t_{n-1} to a cell state at t_n. In this implementation, cell state permutations were identical under rotation:



The action performed on 1 cell is applied in parallel on all cells to calculate the next global state of all cells, defined on a grid. This is repeated the desired amount of times, which will be referred to as running a Cellular Automaton.

Although only shown in 2 dimensions here, this was also implemented in 1 dimension, with a row rather than a grid. This can also be generalised to arbitrary dimensions, at computational cost. The rest of this guide will treat the 2D case, as it was explored the most.

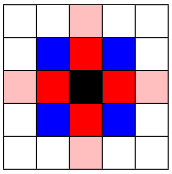
2 CA Rule functions

The implementation of the CA Rule function used here was converting a unique permutation of cell states to a unique index in an array of possible cell states. The method for converting a permutation of cell states in a neighbourhood to an index was kept consistent throughout, the variety of rules were due to varying the array of cell states. As such, any CA Rule can be uniquely identified by this array.

In practice, the CA Rules are randomly generated here, although effort has been made to increase the frequency of generating 'interesting' rules (more on this later). It is worth calculating how many possible CA Rules exist for a given number of states, and cell neighbourhood configuration:

$$\|Rules\| = s^p \tag{1}$$

Where $\|Rules\|$ denotes the size of the set of CA Rules, s denotes the number of possible states of an individual cell, and p denotes the number of unique permutations of cell states in a neighbourhood. The neighbourhood definitions used here are as follows:



Neighbourhood size (N)	Colours
1	Red and Black
2	Blue, Red and Black
3	Pink, Blue, Red and Black

With this definition of cell neighbourhood structures, the number of possible permutations of a cell neighbourhood, p, is given by:

$$p = s + \sum_{x=0}^{N} 4(s-1)s^{x+1} \tag{2}$$

Where N is the numbering of the cell neighbourhood structure. Applying this to a few examples has quite interesting results:

Number of states	Neighbourhood size (N)	$\|Rules\|$
8	1	8^{232}
8	2	8^{2024}
100	1	100^{39700}
10	3	10^{39970}
20	3	20^{639940}

In many cases, the number of CA Rules that can be generated is large enough that it would be impossible to check them all. In fact only a very small fraction of the possible Rules can be generated. This is motivation for developing a technique to limit the set of CA Rules to a set of CA Rules that generate some interesting behaviour, however this is completely non-trivial. Firstly, what defines interesting behaviour? Secondly, if this definition is clear, is there any way to determine if a rule fits this definition without having to run a CA? There are some possible ways to define a rule as 'interesting', which will be explored later.

3 Implementation

The code included is written in Python 2.7, with much use of Numpy, Scipy and MatPlotLib. You will need these to run this code (more details later). The main program asks the user for inputs of: neighbourhood size; number of cell states; size of grid and number of iterations to run the CA for. The code then randomly generates a CA Rule for these parameters, runs the CA and displays the output as a 2D animation.

The CA Rule is applied to the grid of cells using an image convolution function from scipy.signal, which significantly optimises the runtime (compared to a naive method using python for loops).

4 Running the program

Once you have cloned the git repository, navigate into the main directory and type 'python demo.py' to run the demo program. To run the main program, run 'automata_plot.py'. First this will ask the user for CA Rule parameters. Then it will prompt the user with ':'. Simply hit enter to generate a new CA Rule, run it and display an animation. To rerun a rule, enter 'r'. To save a rule, enter 's', and to load a saved rule enter 'l'. There are many more commands that can be entered here, which will be discussed later, but that is all the basics.

4.1 Troubleshooting

If the code fails to run, the most likely cause is you are using python 3. I have written this in python 2.7, but as far as I know the only non compatible parts of the program are the parts that handle user input.

Another possible problem is operating systems. This was written on Ubuntu, if you run this on another operating system, the section of the code that reads/writes CA Rules to files might crash. This has not been tested on other operating systems, but I think that is the only issue that could emerge.

And obviously make sure you have all dependencies installed. The program may take a long time to run, depending on grid size or CA Rule length. If it takes longer than a few minutes, it may eat up RAM.

4.2 Modifying rules

—add later

5 Classifying interesting rules

not really figured out the best way to do this yet

6 RPI

A version of this code designed for a raspberry pi and an RGB LED matrix has been written, although that has not been worked on for some time. More low level programming is needed to have this run efficiently, possibly re-writing some of the code in C. It could make for a very pretty light though.

7 Further modifications

Feel free to modify this as you see fit. I will probably re-factor a lot of it, when I have the time.