Instituto Politécnico Nacional
Escuela Superior de Cómputo
Evolutionary Computing
9 Cellular Automata
Martínez Coronel Brayan Yosafat
Rosas Trigueros Jorge Luis
12/10/2021
23/11/2021

Theorical framework

A cellular automaton consists of a regular grid of cells, each in one of a finite number of states, such as on and off (in contrast to a coupled map lattice). The grid can be in any finite number of dimensions. For each cell, a set of cells called its neighborhood is defined relative to the specified cell. An initial state (time t = 0) is selected by assigning a state for each cell. A new generation is created (advancing t by 1), according to some fixed rule (generally, a mathematical function) that determines the new state of each cell in terms of the current state of the cell and the states of the cells in its neighborhood. Typically, the rule for updating the state of cells is the same for each cell and does not change over time, and is applied to the whole grid simultaneously, though exceptions are known, such as the stochastic cellular automaton and asynchronous cellular automaton [1].

The concept was originally discovered in the 1940s by Stanislaw Ulam and John von Neumann while they were contemporaries at Los Alamos National Laboratory. While studied by some throughout the 1950s and 1960s, it was not until the 1970s and Conway's Game of Life, a two-dimensional cellular automaton, that interest in the subject expanded beyond academia. In the 1980s, Stephen Wolfram engaged in a systematic study of one-dimensional cellular automata, or what he calls

elementary cellular automata; his research assistant Matthew Cook showed that one of these rules is Turing-complete [1].

The primary classifications of cellular automata, as outlined by Wolfram, are numbered one to four. They are, in order, automata in which patterns generally stabilize into homogeneity, automata in which patterns evolve into mostly stable or oscillating structures, automata in which patterns evolve in a seemingly chaotic fashion, and automata in which patterns become extremely complex and may last for a long time, with stable local structures. This last class is thought to be computationally universal, or capable of simulating a Turing machine. Special types of cellular automata are reversible, where only a single configuration leads directly to a subsequent one, and totalistic, in which the future value of individual cells only depends on the total value of a group of neighboring cells. Cellular automata can simulate a variety of real-world systems, including biological and chemical ones [1].

Material and equipment

In this practice, Google Colaboratory with Jupiter Notebooks was used for the development.

Practice development

In this practice, we have two automata to design, both instructions are described in the assignment and looks like this:

Consider a 2D CA defined as follows:

S= infinite rectangular grid.
N={closest neighbors} U {self}
Q={0,1,2}

δ= (b1,b2,s1,s2,u1,u2,r1,r2)

- If the cell is in state 0, and the number of closest neighbors in state 1 is between b1 and b2 (b stands for born) the state of the cell will change to 1.
- If the cell is in state 1,
    - If the number of closest neighbors in state 1 is between s1 and s2 (s stands for survival) the state of the cell will remain in 1.
    - Else: If the number of closest neighbors in state 2 is between u1 and u2 (u stands for upgrade) the state of the cell will change to 2.
- If the cell is in state 2,
    - If the number of closest neighbors in state 2 is between r1 and r2 (r stands for remain), the state of the cell will remain in 2.
- Otherwise, the state of the cell will change to 0.

Find either an oscillator, a still life or a glider that visits all three states and uses in δ at least two digits in your student number.

Consider a 1D CA defined as follows:

S= Infinite 1-D grid.
N= {First 2 layers of closest neighbors} U { self } (|N|=5)
Q={0,1}

Provide a graphical representation of the evolution of this CA with

δ = Wxy0
and
δ = Wxy0R

where x,y are nonzero digits from your student number. Provide evolutions that start with a small number of cells in 1 and with random configuration.

First problem

So, we just need to define b1, b2, s1, s2, u1, u2, r1, r2, it would be harder to define these numbers and then search for a glider or an oscillator. So, rather than doing so, we first give an initial configuration, just as follows:

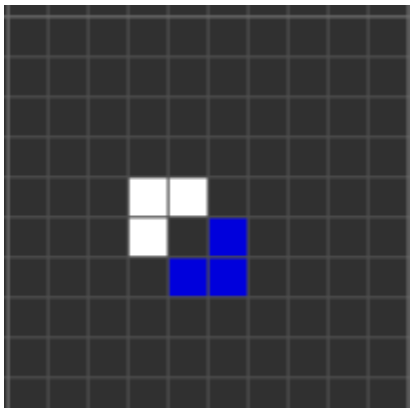

Fig. 1 Initial configuration

Now, we say it is already a still life, and we want to know which numbers use.

Let white be state 2, blue state 1, and grey state 0. First, we do not need to create new state 1 cells, so, let define b1 as 4 (which is part of my student number), and 6 (also from my student number), and this follows the rule at the end. We want state 1 and 2 to stay at it, so, we just pick r1 = s1 = 2 = r2 = s2. So, all cells stay at its state. Finally, u1 and u2 could be (almost) any integer. This solves this part of the problem.

Second problem

This part is harder than the last problem, let us show some code for this. There is a class definition called Automaton, based on the code given by the professor, it helps us to visualize this kind of automata:

```
# Original Title: SecondOrder1dCA.ipynb
# Original Author: Jorge Luis Rosas Trigueros
# Visualizes the evolution of a 1d CA with 1st and 2nd order rules
# Last modification: 10 nov 21 20:23

import cv2
import numpy as np
from IPython import display as display
import ipywidgets as ipw
import PIL
from io import BytesIO
```

```
from random import random
```

Code 1. Imports and credit to the original author

```
class Automaton:
    def __init__(self, grid_width, neighborhood_width, rule, random_in
it: False, is_second_order: False):
        self.grid_width = grid_width
        self.neighborhood_width = neighborhood_width
        self.is_second_order = is_second_order

        self.last_iteration = grid_width * [0]
        if random_init:
            self.current_iteration = [0 if random()>0.5 else 1 for _ i
n range(grid_width)]
        else:
            self.current_iteration = grid_width * [0]
            self.current_iteration[50] = 1
        self.new_iteration = grid_width * [0]

        width = 2*neighborhood_width + 1
        aux = (2**width - len(rule)) * [0]
        aux.extend(rule)
        self.rule = aux[::-1]   # Se pone al revés

        self.x0 = 0
        self.y0 = 0
        self.maxX = 500
        self.maxY = 500
        self.color = (255,255,255)
        self.margin = 1
        self.stride = (self.maxX - 2*self.margin) / self.grid_width
        self.wIm = ipw.Image()
        display.display(self.wIm)
        self.img = np.zeros((500, 500, 3), dtype="uint8")
        self.graph_cells(self.img, self.current_iteration)
        self.y0 += self.stride

        print(self.rule)


    def apply_rule(self, neighborhood):
        total_sum = 0

        for index in range(len(neighborhood)):
            cell = neighborhood[-(index + 1)]
```

```
            total_sum += (2 ** index) * cell

        return self.rule[total_sum]


    def iterate(self):
        while self.y0 <= self.maxY:
            if self.is_second_order:
                self.second_order_evaluation()
            else:
                self.first_order_evaluation()

            self.graph_cells(self.img, self.last_iteration)
            self.y0 += self.stride

        pilIm = PIL.Image.fromarray(self.img, mode="RGB")
        with BytesIO() as fOut:
            pilIm.save(fOut, format="png")
            byPng = fOut.getvalue()

        self.wIm.value = byPng


    def first_order_evaluation(self):
        for i in range(self.neighborhood_width, self.grid_width-
self.neighborhood_width):
            self.last_iteration[i] = self.apply_rule(self.current_iter
ation[i-self.neighborhood_width : i+self.neighborhood_width])
        self.current_iteration[:] = self.last_iteration[:]


    def second_order_evaluation(self):
        for i in range(self.neighborhood_width, self.grid_width-
self.neighborhood_width):
            self.last_iteration[i] = self.apply_rule(self.current_iter
ation[i-self.neighborhood_width : i+self.neighborhood_width])
            self.last_iteration[i] = 0 if self.last_iteration[i] == se
lf.new_iteration[i] else 1

        self.new_iteration[:] = self.current_iteration[:]
        self.current_iteration[:] = self.last_iteration[:]


    def graph_cells(self, img, cells):
        for i in range(self.grid_width):
            if cells[i]:
```

```
                start = (int(self.x0 + self.stride*i + self.margin), i
nt(self.y0 + self.margin))
                end = (int(self.x0 + self.stride*(i+1) - self.margin),
 int(self.y0 + self.stride-self.margin))
                cv2.rectangle(img, start, end, self.color, -1)
```

Code 2. Automaton class definition

```
GRID_WIDTH = 100
NEIGHBORHOOD_WIDTH = 2
RULE = [0,1,0,0,0,0,0,1,0,0]          # Está escrita al revés

suma = 0
potencia = 0
for i in reversed(RULE):
    suma += 2**potencia * i
    potencia += 1

print(suma)

automaton = Automaton(GRID_WIDTH, NEIGHBORHOOD_WIDTH, RULE, random_ini
t=False, is_second_order=True)
automaton.iterate()
```

Code 3. Iteration over time

One rule that I picked (because was easy to put in the array) is 260, this follows the rule about Wxy0, in this case, W260, now, let' s see how it evolved over time, the non-random initialization means that at position 50 it has a 1, and the rest are 0:

Fig. 2 and 3: Non-random initialization, W260 (left) and W260r (right)

Fig. 4 and 5: Random initialization, W260 (left) and W260r (right)

Conclusions and recommendations

References

[1] Wikipedia. Cellular automaton. 4, Nov. 2021. Wikipedia. Accessed on: Nov. 20, 2021. [Online] Available: https://en.wikipedia.org/wiki/Cellular_automaton