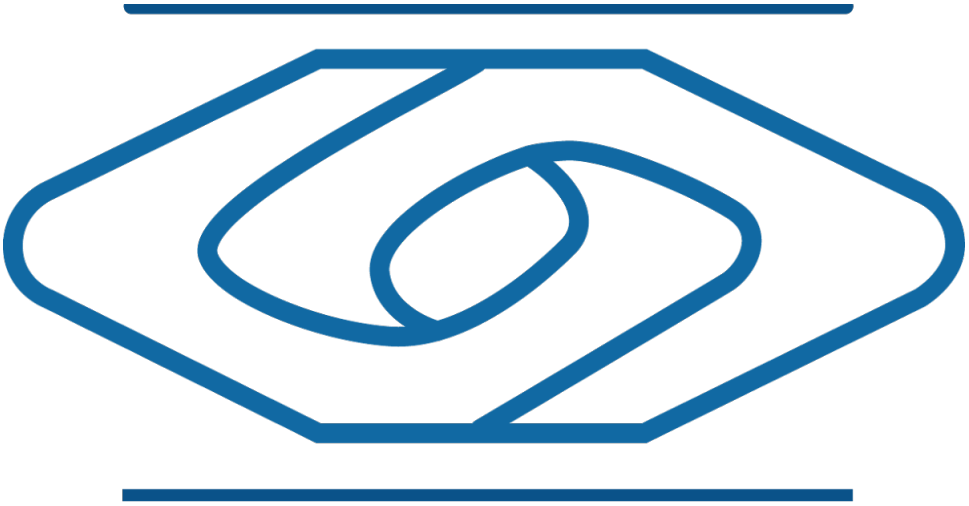


Instituto Politecnico Nacional

Escuela Superior de Computo

Cellular Automata

Submited to:Genaro Juárez Martínez
Submited By: Meza Madrid Raúl Damián
For the class: Computer Selected Topics: Complex Systems

Contents

1 Introduction

As Turing developed the Turing machine, around 1936, John Von Neumann was working on the problem of self-replicating systems[1], a notion of a robot building another robot. Through a reductionist approach, he implemented a method where while trying to calculate the liquid motion, he implemented a method where each unit would behave based on the behavior of the units next to it, this is the beginning of the *Cellular Automata*.
40 Year later, in 1970, John Horton Conway created the *Game of Life*, a zero-player game ,which maybe the reason why people sometimes refer to it as just *life*, which best shows how the complex systems work; a set of elements with simple individual behavior act in a complex way.

2 Development

All the source Code for this and other projects for this class can be found on: https://github.com/asdf1234Damian/ComputerSelectedTopics. To implement the Cellular Automaton, *The Game of Life*, it was decided to first start with the cellular automaton itself instead of the GUI. Since the limits for the automaton where set at 1,000x1,000 cells, the implementation started with a well known by the author programming language, C++.

2.1 C++ implementation

To implement good performance graphics the Automata.cpp and Automata.h files were created. Here we created an class Automata, class that inherits the the methods from the class Window from SFML, a C++ API used to render graphics for 2d games. Each instance of the Automata class is called with the size of the grid and the probability of cells being born at the beggining.
The next is the Automata.h file with comments for each function. This comments are not in the Git repository.

```
#include <SFML/Graphics.hpp>
#include <vector>
class Automata{
public:
  unsigned int size,p;
  Automata(unsigned int,short int );
```

```
  sf::Color alive= sf::Color(250, 250,250);
  sf::Color dead= sf::Color(0,0,0);
  int ls=2,us=3,lb=3,ub=3,zoom=1,viewx=0,viewy=0;
  bool running=false;
  //Starts the programm, handled by SFML
  void run();
  //Random values for the starting grid, based in the given probability
  void randomStart();
  //Starts with a Exploder, the name was never changed
  void gliderStart();
  // applys setcell to all cells and draws the cells in the grid
  void update();
  //changes what part of the grid is being focused
  void updateView();
private:
  sf::RenderWindow grid;
  sf::View view;
  std::vector<sf::Vertex> cells;//Current state
  std::vector<sf::Vertex> cellsNext;//Next state
  void pollEvent();//Main loop, handled by SFML
  //retrives the value of a cell, in charge of the
  //hadling the toroid,or doughnut, aspect
  short int getValue(float, float);  of the automata.
  //retrives the summed value of all the neighbor cells, Moove Neighbor
  short int neighSum(float, float );
  //checks if the cell stays alive or not
  bool rule(float, float);
  //aux function to acces the vector as a matrix
  size_t getIndex(float , float );
  //change the state of a given cell
  void setCell(float , float );
};
```

After this implementation, the SFML window was expected to be implemented into a GUI made with the QT API, but due to the last update, compatibility wasn't possible anymore, then it was decided to implement the program in python.

2.2 Python implementation

Looking for a similar way to implement the same algorithm as in C++, I started with the pyglet library, which offers a similar structure to the SFML API, both of them using the vertex object. After around 50 lines of code, I realized that the same problem as with SFML existed. I couldn't implement a tkinter GUI and the pyglet graphics in the same thread without messing with unknown things. Due to the time left, I restarted the file Automata.py, available at the Git repository, and Started the new GameOfLife.py file, using only tkinter canvas as the grid for the Automata. With an easy to make GUI from the tkinter library in python, the only problem was performance. Hoping that the same algorithm wouldn't be affected, the Python implementation was made using canvas, but performance was awfully slow.
Here is the implementation, nonetheless.

```
from  tkinter  import *
import threading
import random as rng

class Automata:
    toggle=False#toggle between which array use
    running=False#toggle between running and stopped
    cellCount=0
    gridSize=100
    cellsOn=[0]*(gridSize*gridSize)
    cellsOff=[0]*(gridSize*gridSize)
    cellSize=1000.0/gridSize

    def __init__(self):
        self.tk=Tk()

        self.canvas=Canvas(self.tk,width=1000,height=1000,bg="black")
        self.canvas.pack(side=RIGHT)

        self.sizeLbl=Label(self.tk,text="Size of the grid")
        self.sizeLbl.pack(side=TOP, padx=10, pady=(200,5))

        self.sizeIn=Entry(self.tk,width=10,justify="center")
        self.sizeIn.insert(END,"400")
        self.sizeIn.pack(side=TOP, padx=10, pady=10)

        self.sizeLbl=Label(self.tk,text="Probability")
```

```
        self.sizeLbl.pack(side=TOP, padx=10, pady=(10,5))

        self.sizeIn=Entry(self.tk,width=10,justify="center")
        self.sizeIn.insert(END,"10")
        self.sizeIn.pack(side=TOP, padx=10, pady=10)

        self.startBttn= Button(self.tk, text="Run",state="active",
        command=self.run)
        self.startBttn.pack(side=TOP, padx=10, pady=10)

        self.stopBttn= Button(self.tk, text="Stop",state="active",
        command=self.stop)
        self.stopBttn.pack(side=TOP, padx=10, pady=10)

        self.stepBttn=Button(self.tk, text="Step",state="active",
        command=self.step)
        self.stepBttn.pack(side=TOP, padx=10, pady=10)
        #TODO
        self.loadBttn=Button(self.tk, text="Step",state="active",
        command=self.step)
        self.loadBttn.pack(side=TOP, padx=10, pady=10)
        #TODO
        self.sizeIn=Entry(self.tk,width=10,text="400")
        self.sizeIn.pack(side=TOP, padx=10, pady=10)
        #TODO
        self.saveBttn=Button(self.tk, text="Step",state="active",
        command=self.step)
        self.saveBttn.pack(side=TOP, padx=10, pady=10)
        #TODO
        self.clearBttn=Button(self.tk, text="Clear",state="active",
        command=self.step)
        self.clearBttn.pack(side=TOP, padx=10, pady=10)

        self.setRNG()

        #

    def start(self):
        self.tk.mainloop()
```

```
def run(self):
    self.startBttn(state="DISABLED")
    threading.Timer(1.0/30.0, self.run).start()

def stop(self):
    self.stopBttn(state="disabled")
    self.runnign=False
    return

def step(self):
    #print('something')
    self.updateCells()
    self.draw()
    #print('something else')

def getIndex(self,x,y):
    return (y*self.gridSize)+x

def getCVal(self,x,y):#

    if not self.toggle:
        return self.cellsOn[self.getIndex(x,y)]
    else:
        return self.cellsOff[self.getIndex(x,y)]

def getNVal(self,x,y):#
    if x<0:
        x=self.gridSize-1
    elif x==self.gridSize:
        x=0
    if y<0:
        y=self.gridSize-1
    elif y==self.gridSize:
        y=0

    if self.toggle:
        return self.cellsOn[self.getIndex(x,y)]
    else:
        return self.cellsOff[self.getIndex(x,y)]
```

```
def getNeigh(self,x,y):
    #print("NewNeight",x,",",y)
    return (self.getNVal(x-1,y-1)+self.getNVal(x-1,y)+
    self.getNVal(x-1,y+1)+self.getNVal(x,y-1)+
    self.getNVal(x,y+1)+self.getNVal(x+1,y-1)+
    self.getNVal(x+1,y)+self.getNVal(x+1,y+1))


def setRNG(self):
    for y in range(self.gridSize):
        for x in range(self.gridSize):
            if rng.random()<.10:
                self.cellsOff[self.getIndex(x,y)]=1;
                self.canvas.create_rectangle(x*self.cellSize,
                y*self.cellSize,(x+1)*self.cellSize,
                (y+1)*self.cellSize, fill="white")
            else:
                self.cellsOff[self.getIndex(x,y)]=0;
                self.canvas.create_rectangle(x*self.cellSize,
                y*self.cellSize,(x+1)*self.cellSize,
                (y+1)*self.cellSize, fill="black")


def rule(self,x,y):
    sum=self.getNeigh(x,y)
    if(self.getNVal(x,y)):# TODO: Change to current val
        if(sum>=2 and sum<=3):
            return 1
    elif(sum>=3 and sum<=3):
        return 1
    else:
        return 0

def setNext(self,x,y):
    if self.toggle:
        if self.rule(x,y):
            self.cellsOff[self.getIndex(x,y)]=1
        else:
            self.cellsOff[self.getIndex(x,y)]=0
    else:
```

```
            if self.rule(x,y):
                self.cellsOn[self.getIndex(x,y)]=1
            else:
                self.cellsOn[self.getIndex(x,y)]=0

    #Iterates through the array
    def updateCells(self):
        for x in range(0,self.gridSize):
            for y in range(0,self.gridSize):
                self.setNext(x,y)
        self.toggle=not self.toggle


    def draw(self):
        if self.toggle:
            for y in range(self.gridSize):
                for x in range(self.gridSize):
                    if self.cellsOn[self.getIndex(x,y)]==1:
                        self.canvas.create_rectangle(x*self.cellSize,
                        y*self.cellSize,
                        (x+1)*self.cellSize,
                        (y+1)*self.cellSize,
                        fill="white")
                    else:
                        self.canvas.create_rectangle(x*self.cellSize
                        ,y*self.cellSize,
                        (x+1)*self.cellSize,
                        (y+1)*self.cellSize,
                        fill="black")
        else:
            for y in range(self.gridSize):
                for x in range(self.gridSize):
                    if self.cellsOff[self.getIndex(x,y)]==1:
                        self.canvas.create_rectangle(x*self.cellSize,
                        y*self.cellSize,
                        (x+1)*self.cellSize,
                        (y+1)*self.cellSize,
                        fill="white")
                    else:
                        self.canvas.create_rectangle(x*self.cellSize,
```

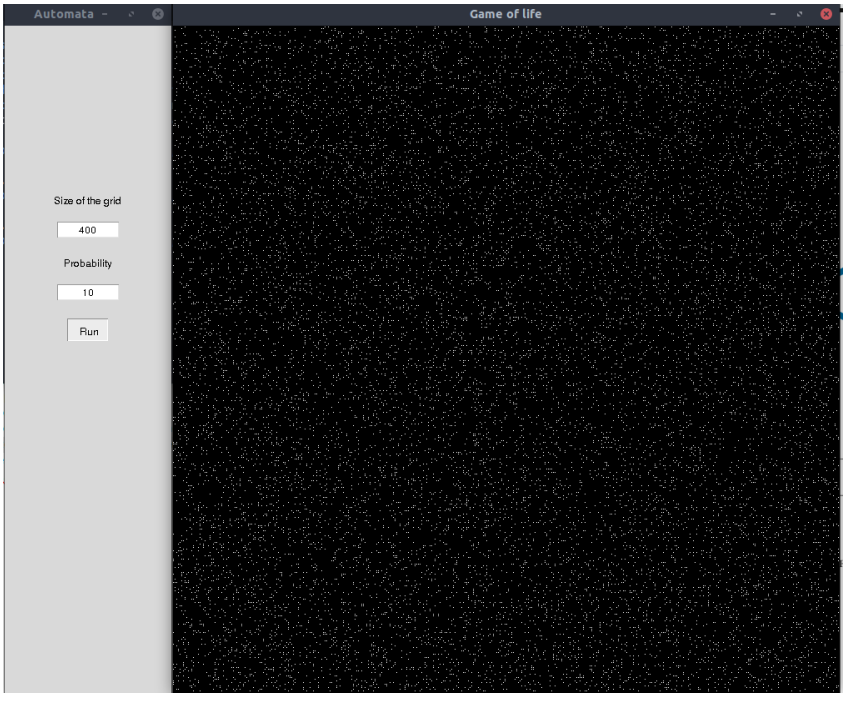
```
                y*self.cellSize,
                (x+1)*self.cellSize,
                (y+1)*self.cellSize,
                fill="black")


GOL = Automata()
GOL.start()
```

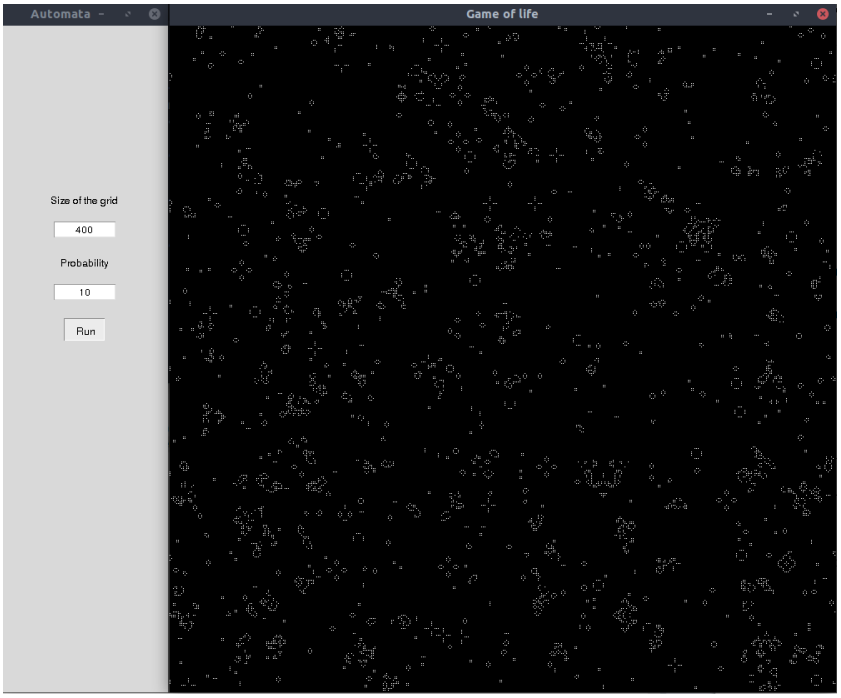
Since this was supposed to be the last implementation of the Automaton, a minor improvement was implemented that allowed to swap which array was being checked or modified instead of having to copy the next state to the current state in each update.
The program was still very slow.

2.3 Merging both implementations

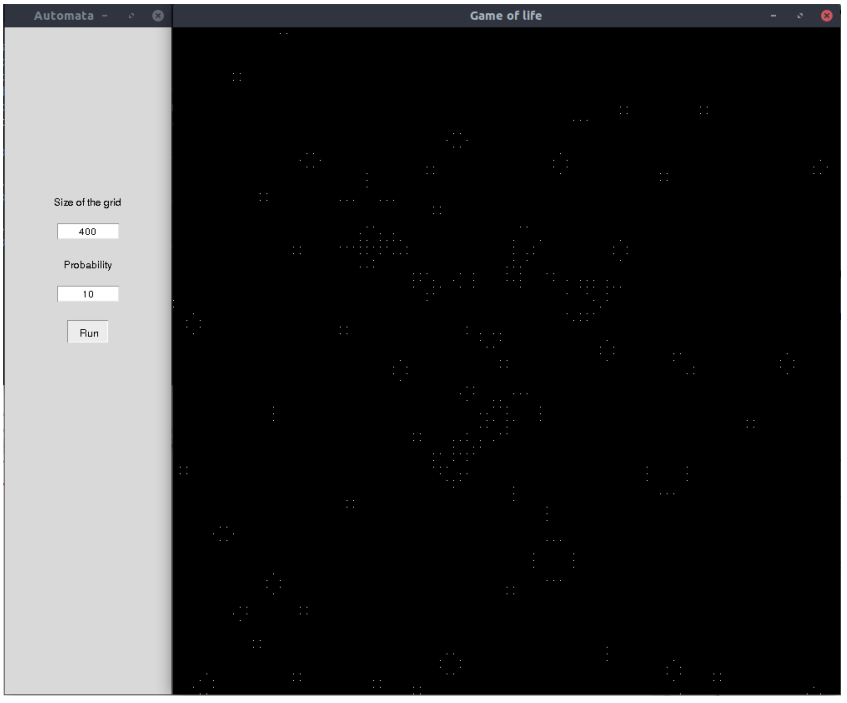
Finally, lacking performance in python and a GUI in C++, I used the subpocess library in python to create a sub-process that would launch the C++, already compiled, program and input parameter from the python GUI.



The Automata Starting with random cells



After some iterations



Zooming in a region

```
from  tkinter  import *
import threading
import random as rng
from subprocess import Popen
class Automata:
    toggle=False#toggle between which array use
    running=False#toggle between running and stopped
    cellCount=0
    gridSize=100
    cellsOn=[0]*(gridSize*gridSize)
    cellsOff=[0]*(gridSize*gridSize)
    cellSize=1000.0/gridSize

    def __init__(self):
        self.tk=Tk()
        self.tk.title("Automata")
        self.tk.geometry("200x800")
        self.sizeLbl=Label(self.tk,text="Size of the grid")
        self.sizeLbl.pack(side=TOP, padx=10, pady=(200,5))
        self.sizeIn=Entry(self.tk,width=10,justify="center")
        self.sizeIn.insert(END,"400")
        self.sizeIn.pack(side=TOP, padx=10, pady=10)

        self.sizeLbl=Label(self.tk,text="Probability")
        self.sizeLbl.pack(side=TOP, padx=10, pady=(10,5))

        self.probIn=Entry(self.tk,width=10,justify="center")
        self.probIn.insert(END,"10")
        self.probIn.pack(side=TOP, padx=10, pady=10)

        self.startBttn= Button(self.tk, text="Run",state="active", command=self.run
        self.startBttn.pack(side=TOP, padx=10, pady=10)
        """
        self.stopBttn= Button(self.tk, text="Stop",state="active", command=self.r
        self.stopBttn.pack(side=TOP, padx=10, pady=10)

        self.stepBttn=Button(self.tk, text="Step",state="active", command=self.ru
        self.stepBttn.pack(side=TOP, padx=10, pady=10)
        """
```

```
    def start(self):
        self.tk.mainloop()

    def run(self):
        automat=Popen(["./Automata",self.sizeIn.get(),self.probIn.get()])
        self.startBttn.config(state="disabled")
        automat.wait()
        self.startBttn.config(state="normal")
        #print(self.sizeIn.get())

GUI = Automata()
GUI.start()
```

3 Restraints

The program was compiled and runned in Ubuntu 18.04.1 LTS, so the C++ compiled program can only be run in Linux. Currently, the program runs *fine*, but some fixes made in the Python implementation could be made to the C++ code. The program doesn't count with a normal distribution RNG nor a plot of the living cells over time.

4 Future

Features like the plot and the normal distribution are ready to be implemented, but not at the time this report is being written. Since the program was made in Ubuntu, there's no support for a GPU, installin SFML in Windows and then re compiling the program should do the job. Alternatively the SFML API has a Python binding called PySFML. At the begginig I planned on using Orthogonal Linked Lists, but wasn't sure about how the rules were going to be applied

5 Conclusion

This is but the foundation for the program. The development will continue during the semester, and more than a project, it will be a chance to learn to deal with both demanding algorithms, optimization and GUI not made in web technologies.

References

[1] Joel L. Schiff. *Cellular Automata: A Discrete View of the World.* Wiley-Interscience, 2008.