Cellular Automata

New Mexico

Supercomputing Challenge

Final Report

April 6, 2019

Team #74

Sarracino Middle School

Team Members

- Sky Sessions
- Angelica Jaquez
- Elena Prieto

Teachers

- Lauri Capps
- Theresa Apodaca

Project Mentors

- Alexander Benson
- Amy Knowles
- Rio Sessions

Acknowledgements

We would like to give a huge thanks to these people for helping us accomplish our project. Thank you to our sponsoring teachers Lauri Capps and Theresa Apodaca. Thank you to Jorge-Silvia Roman for reviewing our project and giving us a clearer view of Cellular Automata. A big thanks to Rio Sessions, a high school student, for taking the time to help us through the project. Thank you to Amy Knowles for helping us with the original idea for Cellular Automata. We would also like to give a gigantic shout out to our college mentor Alexander Benson for coming to the middle school to teach us how to code and guiding us on our project. His commitment is greatly appreciated.

Table of Contents

Executive Summary	4
Statement of problem	5
Introduction	6
Model description	7
Code	8
Reversing & Testing program	9
Results & Conclusion	10-11
Bibliography	12
Appendix A: Game of life rules	13
Appendix B: Cellular Automata Code	14-19
Appendix C: Reversed Cellular Automata Code	20-25
Appendix D: Table of figures	26
Appendix E: Picture	27

Executive Summary

Imagine there is a living system in your computer program. What if each cell in your system is an independent computational unit? If each cell runs its own program you can call it an Automated Cell, or Cellular Automaton (a model studied in computer science, mathematics, physics, complexity science, theoretical biology, and microstructure modeling). Each cell runs its own program to update the next generation using the game of life rules.

In this project we tried to find a seed or origin of a team member's drawing to recreate it using the game of life algorithm. The game of life uses rules to update the next generation. However, when we used the algorithm, it didn't work because the rules state that if there are three living neighbors the state of a cell is alive (in other words the cell is born). The rule also states that if there are two or three living cells that are neighbors then the cell will stay alive. In our program we could not reverse cellular automata since we couldn't tell whether the cell was being born or staying constant.

Problem Statement

Cellular automata is quite complicated and interesting, as they can be as complicated as the program that runs your phone, or as simple as the “hello world” program built in python. So how do we solve them?

First, we need to know how you can code with cellular automata. In this project we tried to find a seed or origin of a team member’s illustration to recreate it using ”The Game of Life” algorithm. The Game of Life uses rules to update the next generation. Our problem that we wanted to solve is coding with cellular automata in reverse to create an illustration designed by one of our team members.

Introduction

The project is about cellular automata (automaton). We are trying to find a seed which is the origin of the illustration using the game of life algorithm to replicate the drawing illustrated by a student. According to Wolfram Mathworld, the definition of cellular automata is “a cellular automaton is a collection of ‘colored’ cells on a grid of specified shapes that evolve through a number of discrete time steps according to a set of rules based on the states of neighboring cells. The rules are then applied interactively for as many time steps as described, for example cellular automata can be the pattern of leaves of geometric shapes on a grid.”

Model Description

Because the Game of Life is built on a grid of nine squares, every cell has eight neighboring cells, as shown in figure 0.

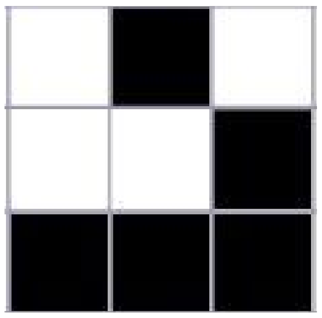
Figure 0: An example of a cellular automaton and its neighbors

A given cell (i, j) in the simulation is accessed on a grid [i][j], where i and j are the row and column indices, respectively. The value of a given cell at a given instant of time depends on the state of its neighbors at the previous time step. Conway's Game of Life has four rules which are as follows:

If a cell is ON and has fewer than two neighbors that are ON, it turns OFF

If a cell is ON and has either two or three neighbors that are ON, it remains ON.

If a cell is ON and has more than three neighbors that are ON, it turns OFF.

If a cell is OFF and has exactly three neighbors that are ON, it turns ON.

Game-of-Life-diagram

So since we know the rules, the next thing we need to figure it out is how to make them work in the program.

Code

For our project, we initially wanted to program in NetLogo, but decided that Python would be a better language to program in. We use John Zelle's graphics module for our project.

Cellular Automata is carried out by creating a three dimensional list that represents the grid. The neighboring cells are determined as the surrounding eight cells. A new three dimensional list is created that is able to store the value for the next generation of cells. As the program is looped to evaluate each and every cell, the new state is determined by the next generation list based on the Game of Life rules (Appendix 1). Each cell is given a value 0 (dead) or 1 (alive). The graphic is also given a color, either grey (dead) or yellow (alive).

Reversing Program (Appendix C)

The program that we decided to use to reverse the rules to try to get a seed to run a reverse of the cellular automata rules, below.

1. If a cell is ON and has fewer than two neighbors that are ON, it remains ON
2. If a cell is ON and has either two or three neighbors that are ON, it turns OFF.
3. If a cell is ON and has more than three neighbors that are ON, it turns ON.
4. If a cell is OFF and has exactly three neighbors that are ON, it remains OFF.

Testing Program

Once we found the end result of the reversing program, we entered the coordinates of all of the live cells (made much easier using a function "live", that would make a cell's value one and turn it yellow). We used the original cellular automata rules and code (Appendix B) afterward, and we were then able to generate the final image.

Results & Conclusion

For the process we used, we generated a random picture (Figure 1).

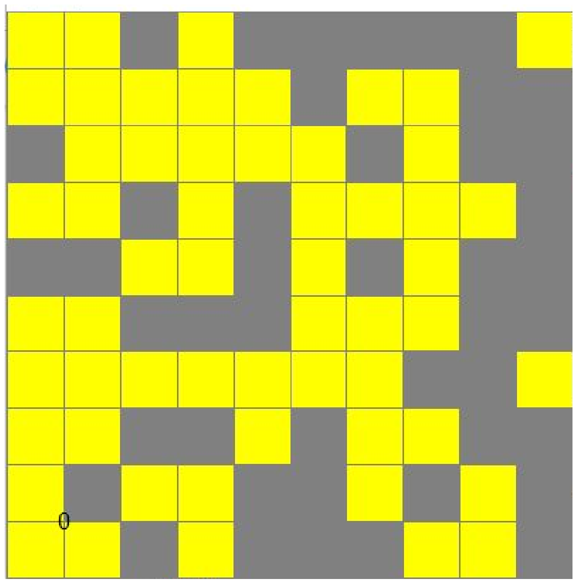

Figure 1: Randomly generated picture that was used to reverse cellular automata.

We then reversed the rules of the game of life (shown in the above “Code” section). The result that would supposedly be the seed (Figure 2) took four frames to come to.

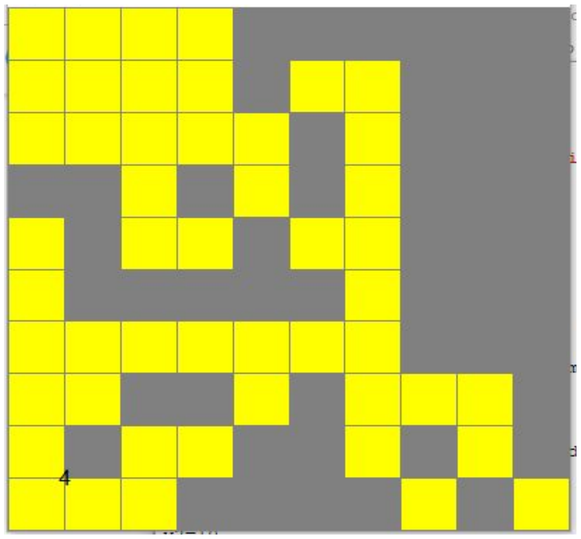

Figure 2: The end result that would be the seed.

After getting this image, we put all of the “live” cells into another program that would perform the game of life to see if we get the randomly generated image back. The very different result (Figure 3) took 27 frames to follow through.

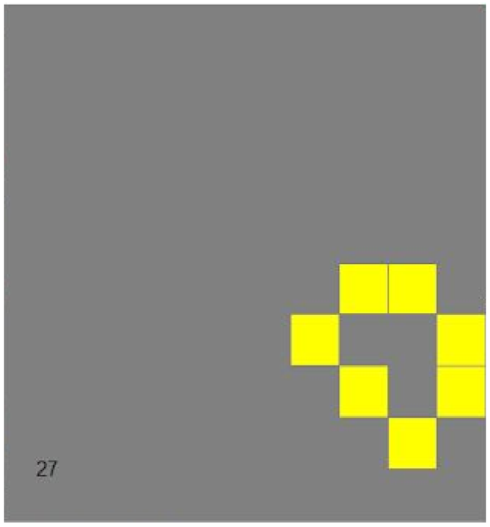

Figure 3: The final result after finishing the game of life.

We were then able to determine that the Game of Life is not reversible. The reason for this is that the rules state that if there are three living neighbors, the state is now alive (the cell is born). It also states, that if there are two or three living and neighboring cells, then the cell stays alive. We cannot reverse cellular automata since we cannot tell whether the cell was born or if it had been alive.

Bibliography

Berto, Francesco Tagliabue, Jacopo " Cellular Automata." Stanford Encyclopedia of Philosophy. *Tue Aug 22, 2017*//plato.stanford.edu/entries/cellular-automata/

Weisstein, Eric W. "Cellular Automaton." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/CellularAutomaton.html

Martin, Edwin "John Conway's Game of Life.", https://bitstorm.org/gameoflife/

Lipa, Chris "Chaos and Fractals." Cornell Math Explorers' Club http://pi.math.cornell.edu/~lipa/mec/lesson6.html

The Coding Train. "7.1: Cellular Automata - The Nature of Code" Aug 10, 2015 https://www.youtube.com/watch?v=DKGodqD

The Coding Train. "7.2: Wolfram Elementary Cellular Automata - The Nature of Code" Aug 10, 2015 https://www.youtube.com/watch?v=W1zKu3fDQR8

Appendix A: Game of life rules

1. Births: Each dead cell adjacent to exactly three live neighbors will become live in the next generation.
2. Death by isolation: Each live cell with one or fewer live neighbors will die in the next generation.
3. Death by overcrowding: Each live cell with four or more live neighbors will die in the next generation.
4. Survival: Each live cell with either two or three live neighbors will remain alive for the next generation.

Appendix B: Cellular Automata Code

```
from graphics import *

import sys, time


#custom info, the board width and height, the background color, the block color, and the
wait time between each loop

dimension = 25

bg = "gray"

color = "yellow"

wait = 0.1

generations = 100


#opens the window

win = GraphWin(width = 400, height = 400)

win.setBackground(bg)

win.setCoords(0, 0, dimension*10, dimension*10)



#creates empty displaySquares array

displaySquares = [[0 for j in range(dimension)] for i in range(dimension)]

x1=0

x2=10

y1=0
```

```
y2=10

#creates and displays all the blocks


for i in range(dimension):

    for x in range(dimension):

        mySquare = Rectangle(Point(x1, y1), Point(x2, y2))

        mySquare.setFill(bg)

        mySquare.setOutline(bg)

        mySquare.draw(win)

        x1 = x1+10

        x2 = x2+10

        displaySquares[i][x] = mySquare


    x1=0

    x2=10

    y1=y1+10

    y2=y2+10


#creates two empty 2D Arrays

squareArray = [[0 for i in range (dimension)] for j in range (dimension)]

squareArray2 = [[0 for i in range (dimension)] for j in range (dimension)]



#set initial state
```

```
squareArray[3][3] = 1

displaySquares[3][3].setFill(color)

squareArray[4][4] = 1

displaySquares[4][4].setFill(color)

squareArray[5][4] = 1

displaySquares[5][4].setFill(color)

squareArray[5][3] = 1

displaySquares[5][3].setFill(color)

squareArray[5][2] = 1

displaySquares[5][2].setFill(color)


#run x generations

test = Text(Point(10,10),str(0))

test.draw(win)


for k in range(generations):

    test.setText(str(k))

    #sleep so we can actually see what's going on

    time.sleep(wait)

    #switches between two different 2D arrays, this makes sure we don't change the current
generation

    if(k % 2 == 0):

        #resets the board

        squareArray2 = [[0 for i in range (dimension)] for j in range (dimension)]
        arr = squareArray2

        notarr = squareArray

    else:

        #resets the board

        squareArray = [[0 for i in range (dimension)] for j in range (dimension)]

        arr = squareArray

        notarr = squareArray2


    #iterate through the board

    for i in range (dimension):

        for j in range (dimension):

            #reset the count of neighbors

            count = 0


            #determine the immediate range of neighbors

            pos1 = i

            pos2 = j

            pos1_start = i - 1

            pos1_end = i + 1

            pos2_start = j - 1

            pos2_end = j + 1

            #check bounds of neighbors

            #hedge

            if(pos1 == 0):
```

```
    pos1_start = pos1
if(pos1 == dimension-1):
    pos1_end = pos1
if(pos2 == 0):
    pos2_start = pos2
if(pos2 == dimension-1):
    pos2_end = pos2



#check actual alive nieghbors
for x in range(pos1_start,pos1_end+1):
    for y in range(pos2_start,pos2_end+1):
        #do not check the square i'm in
        if(not(x == pos1) or not(y == pos2)):
            if(notarr[x][y] == 1):
                count += 1

#Rules for game of life
if(notarr[i][j] == 1):
    if(count < 2):
        arr[i][j] = 0
        displaySquares[i][j].setFill(bg)
    elif(count > 3):
        arr[i][j] = 0
```

```
            displaySquares[i][j].setFill(bg)
        elif(count == 2 or count == 3):
            arr[i][j] = 1
            displaySquares[i][j].setFill(color)
    elif(notarr[i][j] == 0):
        if(count == 3):
            arr[i][j] = 1
            displaySquares[i][j].setFill(color)


win.close()
```

Appendix C: Reverse Cellular Automata Code

```
from graphics import *

import sys, time

from random import randint




#custom info, the board width and height, the background color, the block color, and the
wait time between each loop

dimension = 10

bg = "grey"

color = "yellow"

wait = 1.5

generations = 100


#opens the window

win = GraphWin(width = 400, height = 400)

win.setBackground(bg)

win.setCoords(0, 0, dimension*10, dimension*10)




#creates empty displaySquares array

displaySquares = [[0 for j in range(dimension)] for i in range(dimension)]

x1=0
```

```
x2=10

y1=0

y2=10

#creates and displays all the blocks


for i in range(dimension):

    for x in range(dimension):

        mySquare = Rectangle(Point(x1, y1), Point(x2, y2))

        mySquare.setFill(bg)

        mySquare.setOutline(bg)

        mySquare.draw(win)

        x1 = x1+10

        x2 = x2+10

        displaySquares[i][x] = mySquare


    x1=0

    x2=10

    y1=y1+10

    y2=y2+10


#creates two empty 2D Arrays

squareArray = [[0 for i in range (dimension)] for j in range (dimension)]

squareArray2 = [[0 for i in range (dimension)] for j in range (dimension)]
```

```
#set initial state
for i in range(len(squareArray)):
    for j in range(len(squareArray[1])):
        squareArray[i][j] = randint(0, 1)
        if squareArray[i][j] == 1:
            displaySquares[i][j].setFill(bg)
        else:
            displaySquares[i][j].setFill(color)

#run x generations
test = Text(Point(10,10),str(0))
test.draw(win)

for k in range(generations):
    test.setText(str(k))
    #sleep so we can actually see what's going on
    time.sleep(wait)
    #switches between two different 2D arrays, this makes sure we don't change the current
generation
    if(k % 2 == 0):
        #resets the board
        squareArray2 = [[0 for i in range (dimension)] for j in range (dimension)]
        arr = squareArray2
        notarr = squareArray
    else:
        #resets the board
        squareArray = [[0 for i in range (dimension)] for j in range (dimension)]
        arr = squareArray
        notarr = squareArray2

    #iterate through the board
    for i in range (dimension):
        for j in range (dimension):
            #reset the count of neighbors
            count = 0

            #determine the immediate range of neighbors
            pos1 = i
            pos2 = j
            pos1_start = i - 1
            pos1_end = i + 1

            pos2_start = j - 1
            pos2_end = j + 1
            #check bounds of neighbors
            #hedge
            if(pos1 == 0):
```

```
    pos1_start = pos1
if(pos1 == dimension-1):
    pos1_end = pos1
if(pos2 == 0):
    pos2_start = pos2
if(pos2 == dimension-1):
    pos2_end = pos2



#check actual alive nieghbors
for x in range(pos1_start,pos1_end+1):
    for y in range(pos2_start,pos2_end+1):
        #do not check the square i'm in
        if(not(x == pos1) or not(y == pos2)):
            if(notarr[x][y] == 1):
                count += 1

#Rules for game of life
if(notarr[i][j] == 1):
    if(count < 2):
        arr[i][j] = 1
        displaySquares[i][j].setFill(color)
    elif(count > 3):
        arr[i][j] = 1
```

```
                displaySquares[i][j].setFill(color)
            elif(count == 2 or count == 3):
                arr[i][j] = 0
                displaySquares[i][j].setFill(bg)
        elif(notarr[i][j] == 0):
            if(count == 3):
                arr[i][j] = 0
                displaySquares[i][j].setFill(bg)


win.close()
```

Appendix D: Table of Figures

Figure 0: Example of a Cellular Automaton and its neighbors
Figure 1: Randomly Generated Image that We Used To Reverse
Figure 2: End Seed of the Reversed Image
Figure 3: Final Image after running reversed image

Appendix E: Picture