

PROJECT

Cellular automata and Evolutionary

Course: Artificial Intelligence

Lecturer: Mohammad Ziyad Kagdi

Friona Pocari

I.Introduction:

Cellular Automata

- About cellular Automata
- Source Code Explanation
- Experiments with 5 different rulesets:
 - o Rule 184-
 - o Rule 110-
 - o Rule 90-
 - o Rule 30-
 - o Rule 150-custom

Cellular automata are one of the most interesting computational science investigations of emergent behaviours from simpler interactions. Cellular automata (CA-s for short) are discrete, abstract computing systems that have shown utility in many scientific domains as more focused representations of non-linear dynamics as well as generic models of complexity. Firstly, CA are (typically) spatially and temporally **discrete**^[1]. They consist of atoms or cells, which are a finite or denumerable set of homogenous, simple units. The cells instantiate one of a finite number of states at each time unit. They obey dynamical transition rules or state update functions as they evolve in parallel at discrete time steps: the update of a cell state is obtained by considering the states of cells within its immediate vicinity. Secondly, CA are **abstract**: they can be specified in purely mathematical terms and physical structures can implement them.^[1] Thirdly, CA are **computational systems**: they can compute functions and solve algorithmic problems.^[1] Even though CA operates differently from conventional, Turing machine-like devices, it can simulate a universal Turing machine by using the right rules. As a result, it can compute anything that can be computed, given Turing's thesis.

Simpliest form of CA:

Elementary CA is the most basic type of CA. It functions on a one-dimensional grid with just two states, often 1 or 0, on or off. Every cell that creates the initial configuration first chooses a state, and then they use a set rule to advance to the next generation. Since there $2 \times 2 \times 2 = 2^3 = 8$ possible binary states for the three cells neighbouring a given cell, there are a total of $2^8 = 256$ elementary cellular automata, each of which can be indexed with an 8-bit binary number (Wolfram 1983, 2002).[2]

Specifically on our Project:

When working with cellular automata (CA), particularly in the context of Rule 30, Rule 90, Rule 110, Rule 150, and Rule 184, there are several important considerations, including boundary conditions and Classes of behaviour.

Boundary Conditions:

Periodic Boundary Conditions:

In a CA, the cells often form a one-dimensional grid. Periodic boundary conditions mean that the leftmost and rightmost cells are considered neighbours, creating a circular grid.

Fixed Boundary Conditions:

Fixed boundary conditions assume that the outer cells have fixed values and do not change during the evolution of the CA.

Reflective Boundary Conditions:

Reflective boundary conditions mimic a mirror-like behaviour, where the outer cells reflect the state of their neighbouring cells.

General Classes of Cellular Automata:

Class 1 (Homogeneous and Stable):

-In class 1 automata, the initial pattern tends to stabilize quickly, forming homogeneous and unchanging structures.

Class 2 (Repetitive and Self-replicating: exhibit repetitive and self-replicating patterns. They can generate complex structures but tend to stabilize over time.

Class 3 (Chaos and Complexity):

-Class 3 automata, showcases chaotic and complex behaviour. They produce patterns that are unpredictable and lack long-term stability.

Class 4 (Universality):

Class 4 automata, exemplified by Rule 110, are capable of universal computation. They can simulate any Turing machine, demonstrating high computational complexity.

Rule 30, 90, 110, 150, 184:

Rule 90 is a **class 2** automaton known for self-replicating and recursive patterns, such as the Sierpinski triangle. It exhibits order and predictability.

Rule 110 is a **class 4** automaton, demonstrating universality and computational complexity. It can simulate any computable function, making it Turing complete.

Rule 150 (Symmetric Patterns): a **class 2** automaton. Its patterns are ordered and display a diagonal symmetry.

Rule 184: class 2. and it creates patterns resembling the movement of cars on a highway.

Additional Considerations:

Initial Conditions:

The choice of the initial state significantly impacts the evolution of the CA. Some initial configurations may lead to Garden of Eden patterns (Rule 110-we tried it but we did not get any results).

Neighbourhood Size: influences the complexity and behaviour of the automaton.

Rule Parameterization: determines the transition rules for each neighbourhood configuration.

Visualization and Analysis:

Visualization tools (used the library **matplotlib**) and analysis methods are crucial for understanding the emergent behavior of cellular automata.

This the the Cellular automata code,we have included comments and the source code(aside from the vs-screenshot)

[illegible]

```
def start(self):#start a infinit loop

    while(True):

        for i in self.seed:#This loop prints each cell of the current generation

            if i == "0":

                print(self.off_color, end='')#color black

            else:

                print(self.on_color, end='')#color white

        print("")

        #os.system('clear')

        self.calculateNextState()#after printing calculates the next generation

def calculateNextState(self):

    for i in range(self.noCells):#This loop iterates through each cell of the current generation
and calculates its next state based on the states of its neighbors.

        if i==0:

            prevIndex = self.noCells-1

        else:

prevIndex = i-1

        if i==self.noCells-1:

            nextIndex = 0

        else:

            nextIndex = i+1

        neighbourHood = self.seed[prevIndex] + self.seed[i] + self.seed[nextIndex]

        if neighbourHood == "111":

            self.nextGen += self.ruleSet[0]

        elif neighbourHood == "110":

            self.nextGen += self.ruleSet[1]

        elif neighbourHood == "101":

            self.nextGen += self.ruleSet[2]

        elif neighbourHood == "100":
```

```
        self.nextGen += self.ruleSet[3]

    elif neighbourHood == "011":

        self.nextGen += self.ruleSet[4]

    elif neighbourHood == "010":

        self.nextGen += self.ruleSet[5]

    elif neighbourHood == "001":

        self.nextGen += self.ruleSet[6]

    elif neighbourHood == "000":

        self.nextGen += self.ruleSet[7]

    self.seed = self.nextGen

    self.nextGen = ""

    #`neighbourHood` represents the state of the current cell's neighborhood (its left, center, and right
    neighbors).

    ca=odca()

    ca.start()
```

[illegible]


```

C: > Users > frion > AppData > Local > Temp > 96c8baf8-087d-4738-be23-bfc00fc1c5ef_ca.py.zip.5ef > ca.py > ...
2  class odca:
25
26  def calculateNextState(self):
27      for i in range(self.noCells):#This loop iterates through each cell of the current generation and calculates its next state based on the states of its neighbors
28          if i==0:
29              prevIndex = self.noCells-1
30          else:
31              prevIndex = i-1
32          if i==self.noCells-1:
33              nextIndex = 0
34          else:
35              nextIndex = i+1
36          neighbourHood = self.seed[prevIndex] + self.seed[i] + self.seed[nextIndex]
37          if neighbourHood == "111":
38              self.nextGen += self.ruleSet[0]
39          elif neighbourHood == "110":
40              self.nextGen += self.ruleSet[1]
41          elif neighbourHood == "101":
42              self.nextGen += self.ruleSet[2]
43          elif neighbourHood == "100":
44              self.nextGen += self.ruleSet[3]
45          elif neighbourHood == "011":
46              self.nextGen += self.ruleSet[4]
47          elif neighbourHood == "010":
48              self.nextGen += self.ruleSet[5]
49          elif neighbourHood == "001":
50              self.nextGen += self.ruleSet[6]
51          elif neighbourHood == "000":
52              self.nextGen += self.ruleSet[7]
53          self.seed = self.nextGen
54          self.nextGen = ""
55      #`neighbourHood` represents the state of the current cell's neighborhood (its left, center, and right neighbors).
56
57  ca=odca()
58  ca.start()

```

Ln 55, Col 116 Tab Size: 4 UTF-8 LF MagicPython 3.10.0 64

Rule 30 :

Binary Representation 00011110

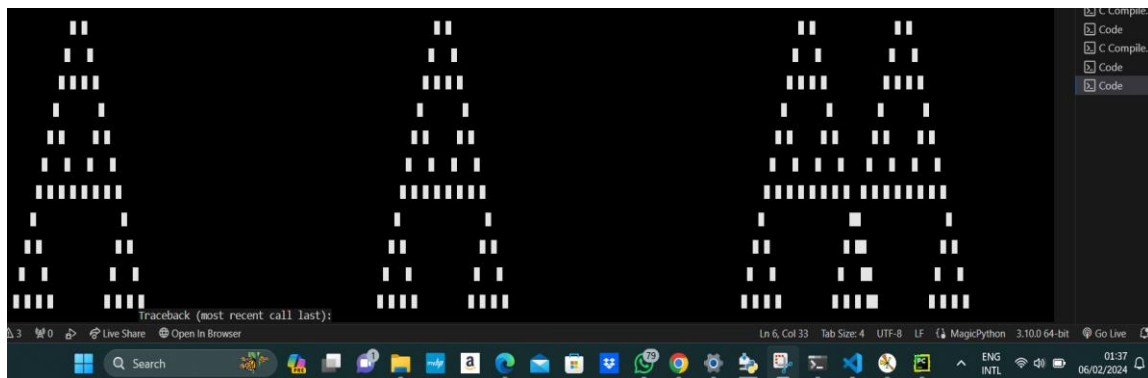
is of special interest because it is chaotic (Wolfram 2002, p. 871), with central column given by 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, ... (OEIS [A051023](#)). In fact, this rule is used as the [random number](#) generator used for large integers in the [Wolfram Language](#).^[1]

Similar to : Sierpinski triangle.

Interesting Features:

The emergence of chaotic patterns and the lack of apparent regularity make Rule 30 a fascinating rule to study.

The rule has been used in cryptography for its pseudo-random properties.



Rule 90:

Binary Representation: 01011010

Special Interest: Rule 90 is not chaotic; it produces a repetitive and symmetric pattern. T

Interesting Features: The ordered and self-replicating nature of Rule 90 makes it a simple yet intriguing rule to study. Its symmetry and repetitive structure are distinct from chaotic rules.

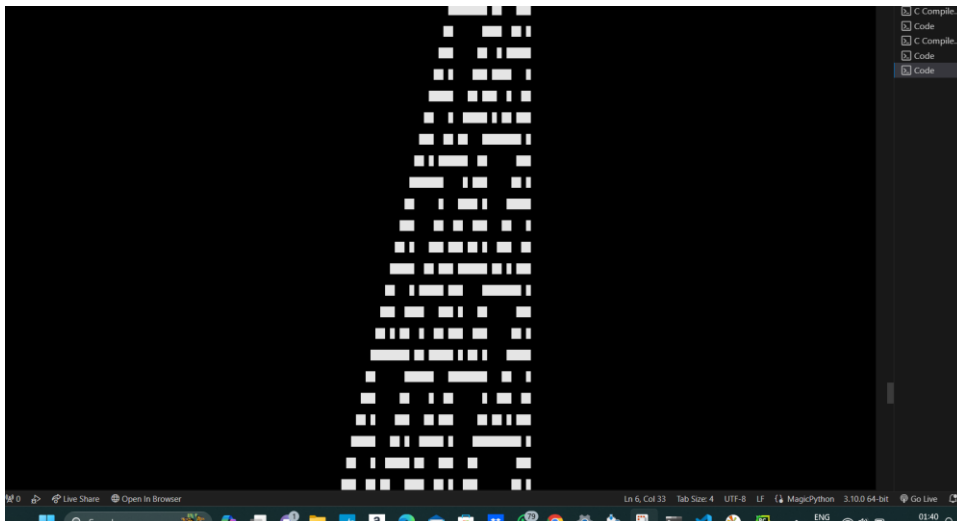


Rule 110:

Binary Representation: 01101110

Special Interest: Rule 110 is considered chaotic, and its behavior is complex. It is known for being Turing complete, meaning it can simulate a universal Turing machine.

Interesting Features: Rule 110 exhibits emergent complexity and is capable of universal computation. It is one of the simplest known universal Turing machines, making it a subject of interest in computer science and complexity theory.

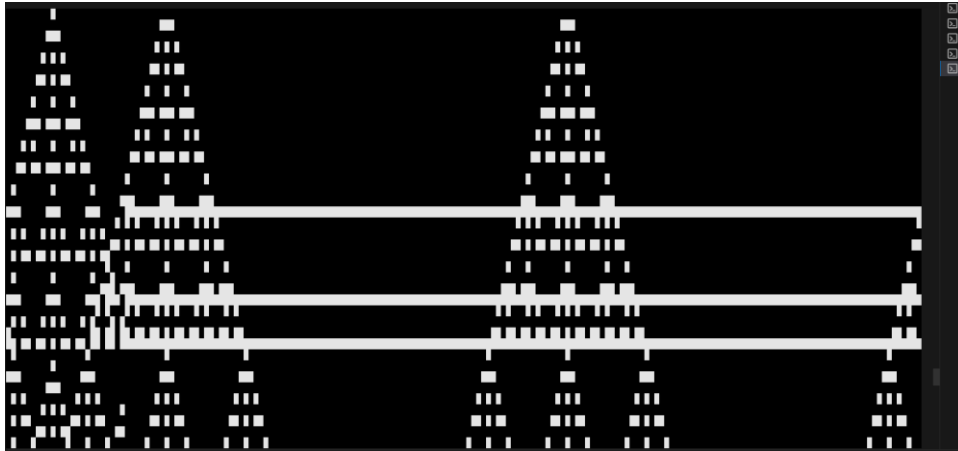


Rule 150:

Binary Representation: 10010110

Special Interest: Rule 150 is not chaotic; it generates a repetitive pattern. It has some resemblance to the behavior of Rule 90.

Interesting Features: The repetitive nature of Rule 150 contrasts with the chaotic behavior of Rule 110. Studying the differences and similarities between these rules provides insights into the diversity of patterns in cellular automata.

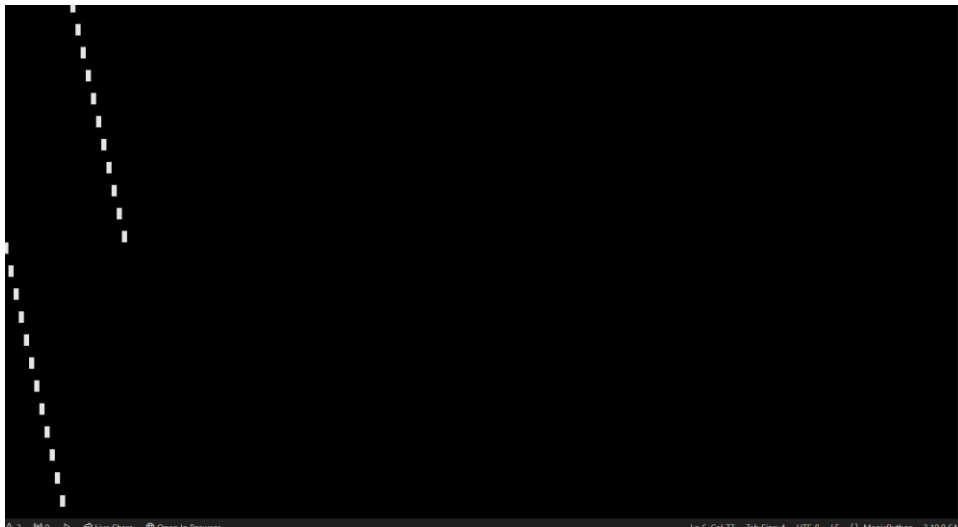


Rule 184:

Binary Representation: 10111000

Special Interest: Rule 184 is not chaotic; it produces a pattern with a distinctive structure. Traffic behaviour similar to cars on a highway.

Interesting Features: Rule 184, with its ordered and structured behaviour, stands in contrast to chaotic rules.



2. Evolutionary Algorithm

Evolutionary Computation (EC), commonly referred to as evolutionary algorithms, is a family of algorithms inspired by nature that seeks to maximise the potential of everything. It's an optimization strategy influenced by Darwinian evolution's survival of the fittest and Mendel's modern genetics.[3] These algorithms are employed to answer problems that cannot be solved in polynomial time, such as NP-Hard issues or any other problems that would take too long to process in its entirety. Evolutionary computation offers roughly ideal answers to challenging issues.

Genetic Algorithm, a **metaheuristic algorithm** in evolutionary computation, uses the Darwinian principle of natural selection and reproduction to generate optimal results to problems from a large search area.[3]

The One-Max issue (we have on our project) is widely used to demonstrate the concept of the genetic algorithm. The problem asks only one question: what is the maximum sum of a bitstring (a string consisting of only 1s and 0s) of length N ?[4] The amount of potential answers rises exponentially with the length of a bitstring.[4]. In Conclusion rather than going over every possible choice in the search area, the genetic algorithm is employed to locate the best answers.

Genetic Algorithm search of the one max optimization problem

(Source Code)

[1]

```
from numpy.random import randint  
from numpy.random import rand
```

– NumPy library is used for generating random numbers download this library on comand prompt.

[2]

```
def onemax(x):  
    return -sum(x)
```

- explains the "onemax" function, which looks for a binary string that contains the most 1s. The return value will equal the negative sum to accomplish this because the genetic algorithm's basic goal is to minimise the function. The largest count of 1s will be equal to minimising the negative sum.

[3]

```
def selection(pop, scores, k=3):  
    selection_ix = randint(len(pop))  
    for ix in randint(0, len(pop), k-1):  
        if scores[ix] < scores[selection_ix]:  
            selection_ix = ix  
    return pop[selection_ix]
```

-the selection function is needed to perform the tournament selection The population's components and their scores will be entered into the function. It will compare the scores of randomly selected population elements. The element designated as the parent will be the one with the lowest score.

[3]

```
def crossover(p1, p2, r_cross):
```

```
c1, c2 = p1.copy(), p2.copy()

if rand() < r_cross:

    pt = randint(1, len(p1)-2)

    c1 = p1[:pt] + p2[pt:]

    c2 = p2[:pt] + p1[pt:]

    return [c1, c2]
```

The crossover function is needed so it can to combine two separate parents in this case p1 and p2 while taking as an input the crossover rate. This will create two children with the genetic info from the parents. First, the two kids resemble their parents. We look for crossover point selection and recombinations. The two kids will subsequently be returned after the crossover is completed.

[4]

```
def crossover(p1, p2, r_cross):

    c1, c2 = p1.copy(), p2.copy()

    if rand() < r_cross:

        pt = randint(1, len(p1)-2)

        c1 = p1[:pt] + p2[pt:]

        c2 = p2[:pt] + p1[pt:]

        return [c1, c2]
```

Defines the mutation operator that will perform the bit flipping of the binary string. The input consists of a bitstring and a mutation rate. Checks for a mutation and if a mutation is found the bit will be flipped.

[5]

```
def genetic_algorithm(objective, n_bits, n_iter, n_pop, r_cross,
                      r_mut):

    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]

    best, best_eval = 0, objective(pop[0])

    for gen in range(n_iter):
```

```
scores = [objective(c) for c in pop]

for i in range(n_pop):
    if scores[i] < best_eval:
        best, best_eval = pop[i], scores[i]
    print(">%d, new best f(%s) = %.3f" % (gen, pop[i], scores[i]))
    selected = [selection(pop, scores) for _ in range(n_pop)]
    children = list()
    for i in range(0, n_pop, 2):
        p1, p2 = selected[i], selected[i+1]
        for c in crossover(p1, p2, r_cross):
            mutation(c, r_mut)
        children.append(c)
    pop = children
    return [best, best_eval]
```

— The main function take as an input the number of iterations, the length of the bitstring, the population size, the crossover and mutation rate. Start: int. the population of a bitstring with `n_pop` members and a length of `n_bits` for every binary string. Using the `onemax` function, scores of every mem. will be measured. The best solution is highlighted and stored. This will be repeated across the generations, examining every individual in the population. After that, a competition will be held to choose the parents who will raise the next generation. Then after the next generation is formed, the code continues to crossover and mutation. Parents must be chosen in pairs. The loop continues for a predetermined number of generations (10) and then stops. In the code provided by our professor, ten generations will pass.

[6]

```
n_iter = 10
n_bits = 50
n_pop = 100
r_cross = 0.9
r_mut = 1.0 / float(n_bits)

best, score = genetic_algorithm(onemax, n_bits,
                                n_iter, n_pop, r_cross, r_mut)

print('Done!')

print('f(%s) = %f' % (best, score))
```


-The results printed at the end are the best solutions.

Experiments with the program:

In order to determine the best option, we have decided to conduct experiments in which we alter the perimeter values and observe the resulting data.

To provide a better understanding, we have altered the values of the iterations, bits (problem size), population size, crossover, and mutation rate in the future.

Experiment 1:(Low Mutation Rate)

Run 1:

[illegible]

Output Iteration Rate : 36

Run 2:

[illegible]

[illegible]

Output Iteration Rate : 27

Run 2:

[illegible]

Output Iteration Rate : 27

Run 3:

[illegible]

Output Iteration Rate : 23

Experiment 3:(Low Mutation Rate)

Run 1:

[illegible]

Output Iteration Rate :33

Run 2:

[illegible]

Output Iteration Rate :31

Run 3:

[illegible]

Output Iteration Rate :32

Experiment 4:(Low Mutation Rate)

Run 1:

[illegible]

Output Iteration Rate : 38

Run 2:

[illegible]


```
72 # define the total iterations
73 n_iter = 80 #10
74 # bits
75 n_bits = 100 #50
76 # define the population size
77 n_pop = 100 #100
78 # crossover rate
79 r_cross = 0.9
80 # mutation rate
81 r_mut = 10/ 50 #until now we experimented with mutation rate 1/
82 # perform the genetic algorithm search
83 best, score = genetic_algorithm(one_max, n_bits, n_iter, n_pop, r_cross, r_mut)
84 print('Done!')
85 print('f(%) = %f' % (best, score))
```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

[illegible]

Output Iteration Rate :46

Run 2:

[illegible]

Output Iteration Rate :50

Run 3:

[illegible]

Output Iteration Rate :38

Experiment 7: (largest population size, the largest crossover point which is 0.99 and the mutation rate of 20)

Run 1:

```
71 # define the total iterations
72 n_iter = 15 #10
73 # bits
74 n_bits = 20 #50
75 # define the population size
76 n_pop = 220 #100
77 # crossover rate
78 r_cross = 0.99
79 # mutation rate
80 r_mut = 20 #until now we experimented with mutation rate 1/
81 # perform the genetic algorithm search
82 best, score = genetic_algorithm(one_max, n_bits, n_iter, n_pop, r_cross, r_mut)
83 print('Done!')
84 print('f(Ks) = %f' % (best, score))
```

Output Iteration Rate :0

Conclusions:

Rate of Mutation

Significance: Experimenting and diversity in the "one-max" problem is essential for analysing different sets of 0s and 1s. Randomization and the prevention of early convergence as we figured from the data and the experiments are benefits of a moderate to high mutation rate.

(moderate to high mutation rates e.g., 0.05 to 0.2.)

Importance Of The Cross-over Rate: To combine genetic material and create solutions for the "one-max". On the other hand, convergence to suboptimal solutions may result from an overly high crossover rate.

What we Understood: To successfully balance exploration and exploitation, start with a moderate crossover rate, anywhere between 0.6 and 0.9. Consider or taking into account the performance that has been seen.

Here, we tested the biggest population size, the highest crossover point, and the most pronounced mutation.

REFERENCES

[1]

E. W. Weisstein, "Rule 30," *mathworld.wolfram.com*.
<https://mathworld.wolfram.com/Rule30.html>

[2]

Wikipedia contributors, "Elementary cellular automaton," Wikipedia, The Free Encyclopedia, 19-Jun 2023

https://en.wikipedia.org/wiki/Elementary_cellular_automaton

[3]

The, "Introducing the one-max problem," The Pragmatic Programmers, 28-Apr-2021. <https://medium.com/pragmatic-programmers/introducing-the-one-max-problem-c128ebd6b3ee>

[4]

D. Soni, "Introduction to evolutionary algorithms," Towards Data Science, 18-Feb-2018. <https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac>