Cellular Automata Optimization Using Genetic Algorithms

Initial State Optimization

Eric Peña

A report presented for the course of SSIE 519
Soft Computing — Neural Networks, Fuzzy Logic, and Genetic Algorithms

System Science
State University of New York — Binghamton
December 12, 2019

Contents

List of Figures

Abstract

The mechanism by which nature exhibits emergent patterns and behaviors has been a mystery throughout history. One application that has been developed which tends to mimic nature is Conway's Game of Life — an application in the field of cellular automata. The ability to predict a final state of a system, given an initial state in the context of Game of Life, come as an insurmountable task. In this work, genetic algorithms are explored along with how they may be used to search for initial conditions such that their final outcomes are optimal. Optimal final states may be defined in terms of growth, diversity, and density of the cellular automaton evolution. This may be beneficial in exploring the way in which coupled components interact in mathematical and physical systems.

Chapter 1

Introduction

1.1 Motivation

Many will claim that the ultimate objective of science is to understand and model the natural world. There are many phenomena in nature whose patterns and behavior seem somewhat unpredictable yet these resulting patterns appear highly structured and organized. Scientists and mathematicians have developed techniques such as chaos theory and cellular automata for the attempt to model nature in its truest sense. In this paper we will take an approach to understand how structure stems from randomness in a cellular automata model. A cellular automaton is defined in terms of clear rules on each individual cell and its well defined *neighborhood* of cells that surround it. We will go into detail as to what this means in later chapters but let us begin by thinking about a two dimensional grid of cells that are all identical. We can even analogize this to a simple universe of people who are all the same and only know how to do the same task: become alive or die. Whether they become alive or die depends on the number of people around them who are either alive or dead given clear, unambiguous rules. Every person in this universe obeys the same universal laws—namely, in this context, the cellular automata rules. Given a clear and finite set of cellular automata rules and given a defined initial state, we can compute the state of a future grid—this will tell us which cells are alive and which are dead, after applying the rules onto the grid some predefined number n times. The defined cellular automata rules used in this report are those defined by Conway's Game of Life. The well defined rules for Conway's Game of Life will be explained in section 2.2.

1.2 Thesis Objective

The objective of this project is to understand which initial conditions (initial states), given a set of well-defined cellular automata rules, produce the most optimized final states after n iterations of applying these rules. The variable being optimized is the fitness value where fitness is defined in terms of what I call growth, diversity, and density of the final state grids. These three terms and how they relate to this specific application are further explained in section 4.4. To make the objective clear, I will state it here and repeat it throughout the report to make sure we are on track with achieving it.

OBJECTIVE: Given well-defined cellular automata rules defined by Conway's Game of Life, determine an initial state that produces an optimal final state in terms of growth, diversity, and density after a finite number of iterations.

1.3 Thesis Outline

The report is organized in chapters that describe the major components of this project. The topics covered are the background of the application [**Chapter 2**], an overview of the genetic algorithms and how they are used to optimize initial states [**Chapter 3**], the details of the genetic algorithm implementation [**Chapter 4**], a description of the results [**Chapter 5**], and a few concluding thoughts and considerations for improvements and future work [**Chapter 6**].

Chapter 2

Application Background

2.1 Cellular Automata

In order to provide a comprehensive background on this particular application, we must cover cellular automata and in particular, Conway's Game of Life. Cellular automata is a discrete model used in an array of disparate fields such as computer science, mathematics, complexity science, and many others. It consists of a regular finite grid of cells and each cell can exist in a particular state; typically these states are *on* or *off* (which we can model as a 0 or 1). The *neighborhood* of a cell is a defined set of cells that surround it. There are *rules* that govern the fate of each cell. These rules can be written as mathematical functions.

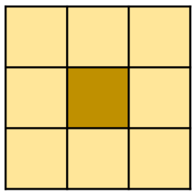


Figure 2.1: Moore neighborhood surrounding a live cell

For example, a deterministic automaton may be represented as a 5—tuple:

$$< Q, \Sigma, \delta, q_0, F >$$

where Q: finite set of states, Σ: finite set of symbols, δ: transition function ($\delta : Q \times \Sigma \to Q$), q_0: initial state, F: final state (or accept states).

On a basic level, they are simply rules that determine the state of a cells depending on the state of cells in their *neighborhood*. The particular cellular automata rule is typically applied to every cell in the grid the same way and does not change in time. This is the case for the application explained in this report. In the 1980's, Stephen Wolfram, a British-American computer scientist, physicists, and businessman, categorized elementary cellular automata into four major classes and are clearly explained in his book, A New Kind of Science[1]. The particular class that is relevant to us is *Class 4* — which states that nearly all initial patterns evolve into structures that interact in complex and interesting ways. One famous example of a *Class 4* is Conway's Game of Life.

2.2 Game of Life

The Game of Life has become the quintessential cellular automaton application and is usually the first example many people are exposed to when learning about cellular automata. It was introduced by the English mathematician John Horton Conway who currently holds the title John von Neumann Professor Emeritus at Princeton University. Professor Conway's initial motivations for developing his Game of Life was to create a cell automaton that was interesting and unpredictable. Some examples of what I mean by the term, *interesting*, are no explosive growth and chaotic outcomes from simple initial patterns. Given particular initial patterns, Game of Life can also exhibit emergent behavior and self-organization. These ideas are closely connected to the idea of spontaneous order from randomness without the presence of a central controller. An interesting question for which this project is largely motivated by is:

Under what initial conditions does Game of Life give us *interesting* behavior and how do we go about finding these patterns?

A defining feature of Class 4 cellular automata is the fact that complex patterns emerge from fairly simple rules. The rules for Game of Life are said to be fairly simple. There are only three rules and they are defined as follows:

1. Any live cell with two or three neighbors survives.
2. Any dead cell with three live neighbors becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.

2.3 Unpredictability and Undecidability

We learned what the overall objective is in section 1.2 — how can we determine which Game of Life initial condition give us the best outcome. Throughout the reading thus far, we have gained an understanding as to why this is an important question in diverse contexts. Another important consideration when exploring this question is how *difficult* is it to answer this question? As is the case in many areas of study, the more generalized the inquiry, the more difficult it is to gain insight. This is because it requires exploring underlying patterns that apply to potentially many phenomena. This question is difficult to answer for a very specific reason and it has to deal with the concept of unpredictability and undecidability. There are problems for which an elegant solution does not exist. In other words, these are problems for which it is impossible to develop an algorithm that leads to the exact correct answer. One of the quintessential examples of an *undecidable* problem is the Halting Problem. The Halting Problem is the problem of determining if a computer program will ever finish running or continue indefinitely given some input. Not only does this theoretical algorithm not exist, it *can't* exist. In 1936, Alan Turing, the founder of computer science, proved that an algorithm to solve the Halting Problem cannot exist which is a profound conclusion that took the science community by surprise. In fact, this proof came after the Austrian logician, mathematician, and philosopher Kurt Gödel published his Imcompleteness Theorems in 1931[2]. These theorems were profound, showing that formal systems, mathematics itself being the most important example, are incomplete and contradictory. You can only imagine hearing that mathematics was broken for the first time. All this is to say that answering the question at hand in this paper is also considered *undecidable*. What this essentially means is the following:

Given an initial pattern and a final pattern, there does not exist an algorithm which will determine whether the final state will ever appear or not.

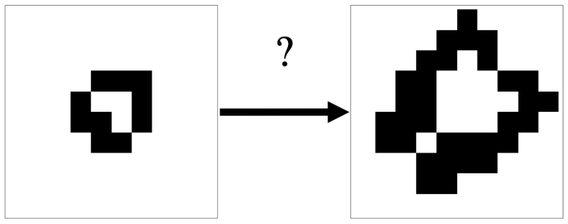


Figure 2.2: Undecidability of Conway's Game of Life

This statement about Conway's Game of Life is actually a corollary of the Halting Problem.[3] If one did exist, we can simply choose the final output we desire and run the algorithm against a list of initial conditions which would produce a list of True or False values — a rather straight forward approach to our question.

Since this is not the case, we must use some optimization technique to search for the initial condition in another way. Genetic algorithms lends itself to finding these types of solutions in a rather elegant way and is the reason why I chose to pursue this quest with its assistance.

2.4 Conclusion

Hopefully we understand what the question is, why it is important, how difficult a question it is to answer, and how we will push forward in answering it. The optimization method of choice is a genetic algorithm. In the SUNY Binghamton SSIE 519 course, simple genetic algorithms were discussed with specific features and methodologies. Although the use of genetic algorithms in this project are largely similar to how they were taught in the course, there are important differences that will be discussed in the subsequent chapters. The next chapter will cover the background of the approach used to achieve our goal.

Chapter 3

Approach Background

3.1 Introduction to Optimization

It is beneficial to understand how we know when a problem could be solved using optimization. An optimization problem is one in which the best solution is chosen or found out of many possible solutions. In the context of cellular automata, we can imagine any number of initial conditions all of which give different final states after running the Game of Life rules some arbitrary number of times. In fact, given a grid of size $n \times n$, the number of possible initial conditions is 2^{n^2}. When $n = 2$, for example, the number of initial conditions is 16. However, when $n = 6$, a seemingly innocuous number, the grid actually has $68,719,476,736$ possible initial conditions! There are techniques that I have put in place to manage this combinatorial growth of initial conditions which I go into detail in Chapter 4.

Choosing an initial condition falls in line with what an optimization technique can help us with. Another trivial consideration in optimization is the variable being optimized. The multivariable equation below, for example, has a clear point of optimal value. Although we essentially have an infinite number of possible points to choose from, there are well-developed mathematical techniques for finding the maximal point efficiently.

$$f(x, y) = -(x^2 + y^2)$$

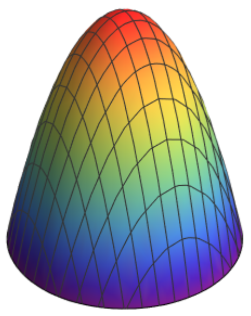


Figure 3.1: Equation: $f(x, y) = -(x^2 + y^2)$ with clear maximum value

In our application, the maximal point is not as clear. What does it mean to have the *best* final state? It is up to the designer to decide. In other words, it is up to me and how I define this. Before we go into what it means for a state to be optimal, let us go into the specific optimization technique we will use for this application, genetic algorithms, and how we will use it to tackle this seemingly unapproachable, *undecidable* problem.

3.2 Evolutionary Programming and Genetic Algorithms

Evolutionary programming is a family of global optimization techniques that are biologically inspired. It works similarly to the natural process in evolutionary biology. Trial and error is a large component along with stochasticity and survival of the fittest. The type of evolutionary computation that we will focus on is genetic algorithms which became popular by the American Professor of Psychology, Electrical Engineering, and Computer Science, John Holland. Genetic algorithms work using natural selection of different possible solutions competing with one another. It incorporates biological concepts such as selection, crossover, and mutation. Candidate solutions can mate with each other and create offspring who can then compete with other solutions in their generation. I will go into each of the main components of a genetic algorithm with respect to the simple genetic algorithm taught in SSIE 519 and discussed in the genetic algorithms textbook by Goldberg[4]. This will also help in establishing a vocabulary for understanding the specific implementation discussed in Chapter 4.

3.3 Components of Genetic Algorithms

3.3.1 Gene, Chromosome, Population

We have mentioned that a genetic algorithm consists of individual solutions that compete with one another so that the optimal solution prevails via artificial evolution. In order to allow these solutions to compete with one another, there needs to be some consistency in how they are represented. The way these solutions are typically represented, if alphanumeric, is with binary numbers. For example, if we wanted to represent the value 34 in binary, we can write it as:

$$100010_b$$

Every 0 or 1 in this list is called the **gene**. The list of these genes is called the **chromosome**. In a genetic algorithm, we want many values (chromosomes) to be tested so there are often many chromosomes that are competing with one another at a time. The group of chromosomes is called the **population**. It is important to note that the chromosomes, initially, are generated at random and have not been pre-designed by the scientist. Below is an example of a population of size 5—there are 5 chromosomes each of which consists of 6 genes.

100010_b	100011_b	101110_b	110001_b	101010_b

These binary numbers correspond to $(34, 35, 46, 49, 42)$ in decimal. Again, these are random numbers that will be used as candidate solutions to our problem. The higher the fitness of these values, the closer they are to answering the question at hand. Let's dive into what the fitness of a chromosome means and how it is useful.

3.3.2 Fitness Function

In order to determine which one of these candidate solutions are more fit than the others, we must use what is called a fitness function. Generally, a fitness function can be difficult to obtain considering it is subjective and left up to the programmer or scientist to decide how fitness is defined in his/her experiment. Let us continue to use our binary numbers for a simple example. Say we have a well-defined function in the form of an equation of x:

$$\beta(x) = -(x - 40)^2 + 40$$

We can use the function itself as our fitness function. Given this fitness function, we can now test our five chromosome values from our population. The plot below shows our candidate solutions on the function β.

The chromosome with the highest fitness is the one for which $\beta(x)$ is maximized out of the five options. We see that 101010_b (42_{10}) is the value that has the highest fitness. All the fitness values are:

$\beta(34) \to 4$	$\beta(35) \to 15$	$\beta(46) \to 4$	$\beta(49) \to -41$	$\beta(42) \to 36$

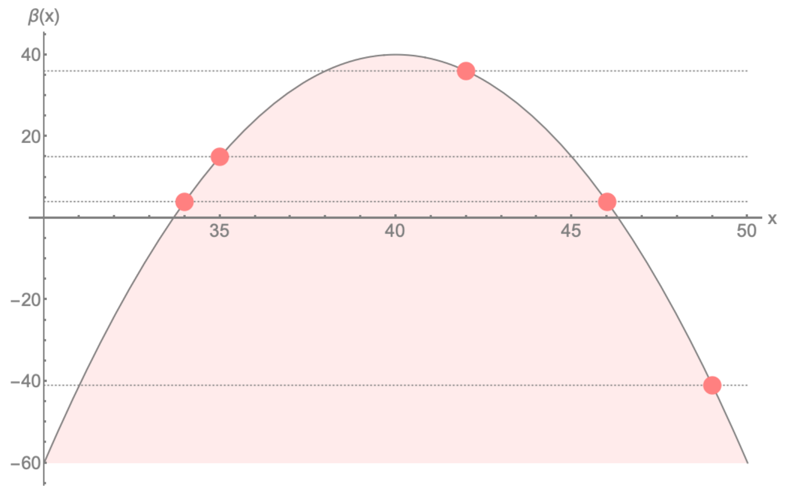


Figure 3.2: Chromosome values plotted on the function $\beta(x)$

3.3.3 Selection

Selection is the process by which chromosomes are chosen to mate with each other and create offspring. In the case of simple genetic algorithms, as defined in GA literature[5], two selected parents are chosen and create exactly two offspring who then replace the very parents that created them. We can say that the two parents *mate* with each other and the resulting two offspring contain *genes* from each of their parents. Another term for mating is *crossover* and is discussed in section 3.3.4. The way this *selection* happens can vary depending on the particular implementation of genetic algorithm being used. Traditionally, the parents with higher fitness are more likely to be chosen and put into the mating pool to create offspring.

To stay consistent with our example above, the two chromosomes (parents) with the highest fitness values are 100011_b (35_{10}) and 101010_b (42_{10}). Therefore, there is a high probability that these two values will be chosen to mate with one another. Let us discuss some specific topics within the topic of selection that will be important to us later on: *elitism*, the *roulette wheel*, and *scaling*.

Elitism

Incorporating elitism is fairly straight forward. It is when you take chromosomes with the highest values and place them in the next generation, unchanged. This guarantees that the highest fitness in each generation will never decrease—it could very well stay the same for a certain number of generations but will never decrease. If we have a complicated optimization problem and fitness function, it may be helpful to employ elitism to gain some traction on your problem. The number of high-fitness parents that are chosen as *elite* depends on the implementation. It could be a fixed number of parents or a percentage of the top parents.

Roulette Wheel

The roulette wheel works very well like a *real* roulette wheel when thought of as uneven areas on a circle. There is one roulette wheel per generation and, as we already know, contains chromosomes and their respective fitness values. The number of sections on this wheel will be equal to the number of chromosomes in the population. The relative sizes of these sections on the circle will be proportional to the fitness values of each chromosome. The higher the fitness value, the larger the area. More concretely, the percentage of the area taken up by a chromosome is the fitness value of that chromosome divided by the total fitness of all chromosomes in the population. In our example you can imagine a roulette wheel like the one in figure 3.3. The sections with the two largest areas are the two most likely to be chosen for mating (or crossover).

Scaling

Scaling can be an important part of the process as well. To immediately motivate this, let's take our example. One of the fitness values is (-41). Clearly, this cannot represent an area and therefore scaling is needed

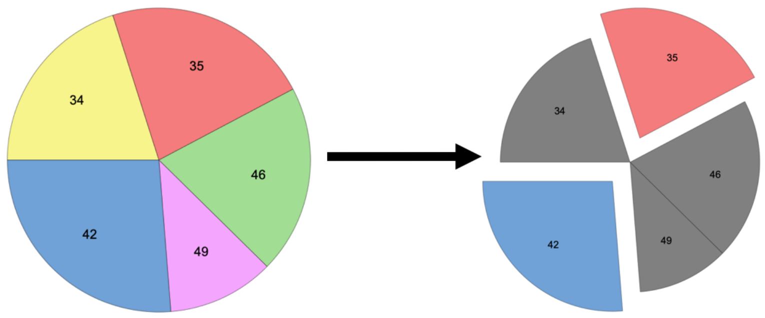


Figure 3.3: Genetic Algorithm Roulette Wheel for Chromosome Selection

to make a useful roulette wheel. Negative values are just one way scaling can help. Typically, scaling is achieved by a linear transformation of fitness values using $f' = af + b$. I will list out three reasons how scaling can be help us not run into issues:

1. Negative numbers like we mentioned above
2. If there are few high fitness scores and most are low fitness scores, the few high fitness chromosomes will take up most of the roulette wheel which could possible make the next generation fitness scores unnaturally uniform. We want to avoid this and scaling can help us in this way.
3. If the best and worst fitness values are near each other, there is little change that the effect of *survival of the fittest* will help us converge to a meaningful answer. Scaling may help us in this way as well.

3.3.4 Crossover

Crossover has been mentioned several times thus far but now we will go into detail as to what it means for two chromosomes to *mate*. There are countless methods on how this is achieved but one common, rather simple method is called *Single Point Crossover*. After we have selected the two chromosomes, we will determine a random location in both chromosomes and exchange genes in such a way that the two resulting offspring contain features from both their parents. This is important to understand so a visual is provided below in figure 3.4. This is the crossover technique that is used in the cellular automata project discussed in chapter 4.

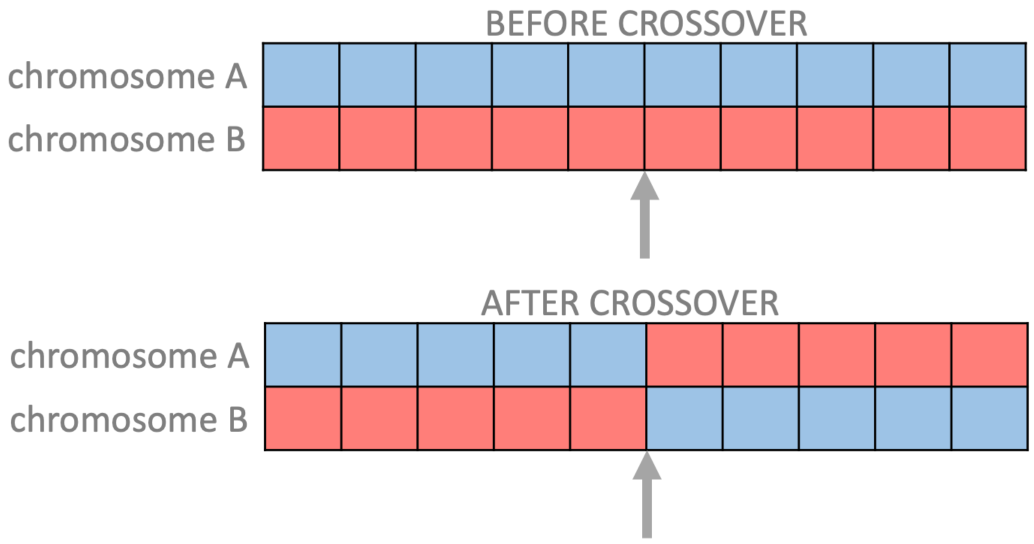


Figure 3.4: Chromosome Crossover

3.3.5 Mutation

Nature could not have evolved if it wasn't for random mutations of genes throughout the population. Mutation works in a very similar way in genetic algorithms. Let there be some percentage that a random gene will change states which occurs after crossover. This is the crux of mutation. This introduction of randomness actually helps the algorithm converge to a solution in a more timely manner than it would have otherwise. A helpful visual has been provided for this method as well in figure 3.5.

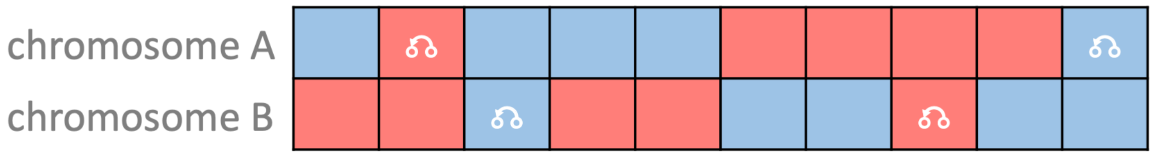


Figure 3.5: Random Chromosome Mutation

3.4 Conclusion

This concludes our discussion of the background of genetic algorithms—the method by which we will explore our cellular automata optimization problem. This chapter provides fundamental knowledge and vocabulary to move forward in understanding how to achieve our objective stated in section 1.2.

Chapter 4

Implementation Details

4.1 Introduction

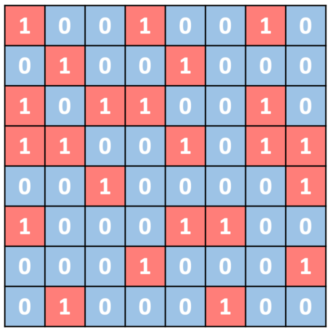
This chapter will provide details of the genetic algorithm used to optimize the cellular automata initial condition. It may be helpful in restating the goal before we move forward into the minutiae of the implementation.

> **Given well-defined cellular automata rules defined by Conway's Game of Life, determine an initial state that produces an optimal final state in terms of growth, diversity, and density after a finite number of iterations.**

To be clear, I will refer to this specific problem as the acronym, **LISO** (Life Initial State Optimization). This chapter will be expound on topics covered in Chapter 3, specifically Section 3.3 where we discuss the components of a genetic algorithm. We will explore, piece by piece, how we achieve our goal and then continue on in Chapter 5 to discuss the final results we obtain.

4.2 Gene, Chromosome, Population

From what we understand so far, a chromosome represents one possible solution to our optimization problem and consists of genes that serve as its simplified representation. In the context of LISO, each chromosome is a grid or in other words, an initial condition. Figure 4.1 shows an example of what a LISO chromosome might look like.



1	0	0	1	0	0	1	0
0	1	0	0	1	0	0	0
1	0	1	1	0	0	1	0
1	1	0	0	1	0	1	1
0	0	1	0	0	0	0	1
1	0	0	0	1	1	0	0
0	0	0	1	0	0	0	1
0	1	0	0	0	1	0	0

Figure 4.1: Random LISO Chromosome

We can think of connecting all the rows of this grid to give us a chromosome whose representation is more familiar to us. Figure 4.2 shows what this might look like. Although we will continue to use the representation showed in 4.1 to keep the structure of the grid. We will see why this is helpful when we get to crossover. You can imagine what the LISO population might look like. Figure 4.3 can assists us in this regard.

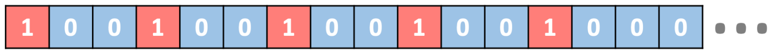


Figure 4.2: LISO Chromosome with Familiar Representation

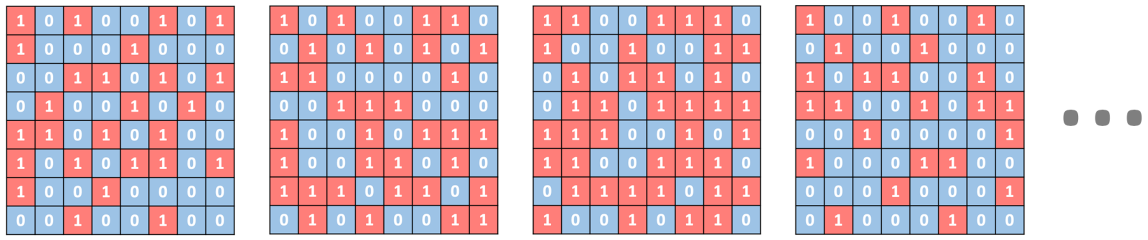


Figure 4.3: Random LISO Population

Zero Border on Initial States

An important addition that I have added to the LISO implementation is the border of zeros on the initial conditions. Since one of the goals in optimizing the initial condition is to maximize growth, we want the initial condition to be as small as possible. This also relates to the desire to limit the number of initial condition configurations which has been motivated in section 3.1. When a grid is instantiated in code, a border size is specified and a border of zeros is added around the grid for the initial states. Figure 4.4 is what this border looks like. This particular initial state is a grid of size 8×8 with border size $\rightarrow 2$.

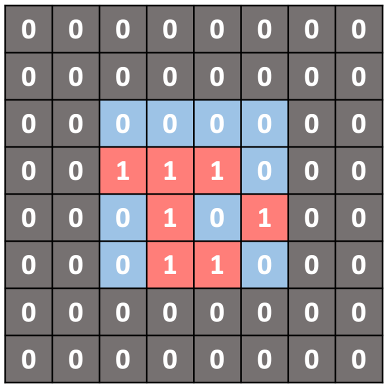


Figure 4.4: Border of Zeros Surrounding Initial Condition Grid

We can say that Figure 4.3 is a population of 8×8 grids with border size $\rightarrow 0$. This is what we should know surrounding the genes, chromosomes, and population of the LISO problem we are exploring.

4.3 Living via Game of Life

Before moving forward to calculate a fitness function we need to generate the *final states* of the initial grids. This is done simply by applying the Game of Life rules discussed in section 2.2. An example of what this might look like starting from the initial state we encountered from the previous section is displayed below. The figure is to be read from *left* to *right* with each grid showing successive steps. The red cells are the *live* cells. The dark gray cells are only to show that there has been an initial zero-border defined of size 2. For all intensive purposes, blue and dark gray can be thought of as the same color since they are both zero. If this pattern continued further, all cells would disappear and the grid would be covered in zeros. If our program specified that the number of iterations should be 6, then the final state certainly has grown with this example. If instead the number of iterations were 10, where all cells would have disappeared, then this initial state will not have survived and therefore not move on to later generations.

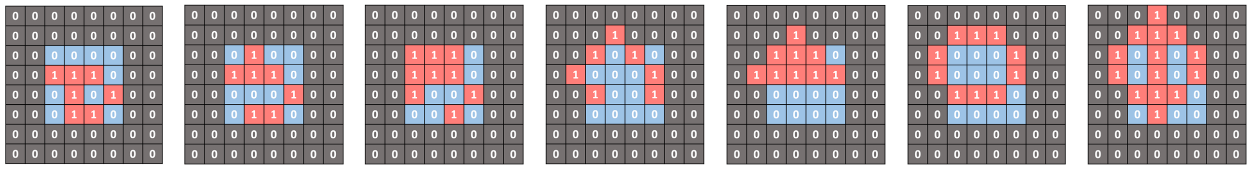


Figure 4.5: Application of Game of Life Rules

4.4 LISO Fitness Function

We have mentioned earlier that the fitness function can sometimes be difficult to define, particularly in problems such as this one. The fitness function in LISO application is defined in terms of the growth, diversity, and density of the grids. In this section, we will explain what each of these mean.

4.4.1 Growth

This simply means that after a certain number of iterations of applying Game of Life rules, we want the initial condition to grow as much as possible. Growth, in this context, means to increase the number of live cells (**1**'s) in the grid. We can claim definitively that if the final state has more number of ones than the initial state, growth has occurred.

4.4.2 Diversity

Maximizing diversity refers to the spread of live cells across the grid. If the spread of live cells is wide and far reaching, the diversity value will increase and therefore increasing the fitness function. This is a rather tricky measure as a standard statistical measure cannot tell the difference between the following two matricies:

$$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

What we will have to do is weight the edges of the grid higher values and get smaller as we tends towards the center as to penalize clustering. The code to do this is fairly straight forward:

```
size = 4
z = np.zeros((size,size), dtype=np.int)
for i in range(size//2):
    for j in range(size//2):
        z[i][j] = size - 1 - i - j
for i in range(size//2):
    for j in range(size//2,size):
        z[i][j] = abs(i - j)
for i in range(size//2,size):
    for j in range(size//2,size):
        z[i][j] = abs(size - 1 - i - j)
for i in range(size//2,size):
    for j in range(size//2):
        z[i][j] = abs(i - j)
```

This code produces the following matrix which we can multiply by the two matricies above and then take the variance. This measure will give us a measure that encourages stretching of live cells across the grid framework.

$$\begin{pmatrix} 3 & 2 & 2 & 3 \\ 2 & 1 & 1 & 2 \\ 2 & 1 & 1 & 2 \\ 3 & 2 & 2 & 3 \end{pmatrix}$$

4.4.3 Density

The density is very similar to the growth except it has no knowledge of the initial state. Density only looks at the final state and asks: how many live cells are there? If there are a high number of live cells then the density is high therefore making the fitness value for this chromosome high as well.

4.4.4 Putting It All Together

In order to come up with an over fitness function, these three values are summed and immediately used in selection.

$$F_i = G_i + V_i + D_i$$

Where F, G, V, and D are the fitness, growth, diversity, and density of the chromosomes respectively.

4.5 Selection

Selection is unique when it comes to LISO as it does not utilize the roulette wheel for choosing parents to mate. Although, elitism play a large role in how this genetic algorithm evolves. In fact, elitism encompasses mostly what selection is in this case. After the fitness has been evaluated, the top 20% of the chromosomes are used to create offspring and *also* move on to the next generation as well. This is an important note about selection in this particular implementation.

> **The top 20% of the chromosomes are used to create offspring and *also* move on to the next generation as well.**

The bottom 80% are actually discarded and new chromosomes are created to fill in the rest of the population. These new chromosomes are created by randomly choosing among the elite parents and having them mate with one another. The snippet of code below shows how simple the selection method is:

```python
def selection(agents):
    """
    Select top Agents based on fitness values

    Args:
        agents ([Agent]): List of Agents that represent population
    Returns:
        agents ([Agent]): List of Agents that represent population
    """
    agents = sorted(agents, key=lambda agent: agent.fitness, reverse=True)

    # :: Selects the top 20% of the agents and removes the others
    agents = agents[:int(0.2 * len(agents))]

    return agents
```

4.6 Crossover

We kept the chromosomes represented as two dimensional grids for this particular section, crossover. The way that crossover is implemented in LISO is in the following way:

- A number, η, is randomly chosen between 0 and $row.size() - 1$.
- For each parent, the grid is cut horizontally at η.
- Two offspring are generated using the *Single Point Crossover* method.

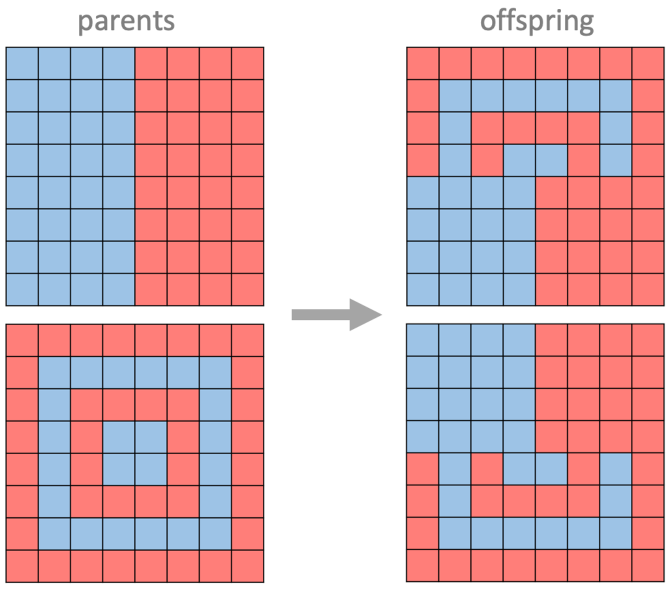


Figure 4.6: LISO Crossover

4.7 Mutation

There are many way that mutation can achieved and in LISO, it is fairly straight forward. The elite chromosomes that have been copied over from the previous generation are not mutated at all. Only the newly created chromosomes are candidates for mutation. The number of chromosomes that are chosen for mutation is governed by a *mutation rate* defined in the beginning of the program. Then, for each chromosome chosen for mutation, a row and column are chosen at random for bit flipping. Since this application is so unpredictable, as explained in section 2.3, mutation will help us converge to an optimal solution more quickly. The code that controls the mutation method is displayed below.

```python
def mutation(agents):
  """
  Randomly change several individual values in each Agent grid

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  family = []

  n = int(0.2 * len(agents))

  for elite in agents[:n]:
    family.append(elite)
  for p in agents[n:]:
    for i in range(int(mutation_rate * grid_size * grid_size)):

      r = random.randint(border, grid_size - border - 1)
      c = random.randint(border, grid_size - border - 1)

      p.state[r][c] = 1 if p.state[r][c] == 0 else 0
      p.final[r][c] = 1 if p.final[r][c] == 0 else 0

    p.fitness = (p.final.var() + p.final.sum())
    p.neighbor_sum()
    family.append(p)

  agents = family
  return agents
```

4.8 Process Flow

Now that we have learned about the components of the genetic algorithm, let's take a look at how these components work together. It is helpful to get a feel for the flow of the program and how it works.

I : Instantiate an array of Game of Life initial states (*create population of chromosomes*)

II : Let each initial state "live" (*apply Game of Life rules n number of iterations*)

III : Calculate fitness values for each chromostome (*using both the initial and final state*)

IV : Sort chromosomes by fitness and choose the top 20%

V : Create offspring using the top 20% of the parents (*randomly choose parents to create many offspring*)

VI : Mutate a selected number of chromosomes based on a global mutation rate

VII : Return to step II and continue for a specified number of generations, g

Global Variables

- **population** — determines how many chromosomes in a population
- **generations** — specifies how many times the flow explained above should occur
- **grid_size** — defines the size of the chromosome grid which will be a (*grid_size* × *grid_size*) matrix
- **border** — specifies the width of zeros to apply to the initial state
- **iterations** — presets the number of times the Game of Life rules should be applied to the initial state to define final state
- **mutation_rate** — tunes the sensitivity of mutation – the higher the rate, the more mutation will occur in offspring

4.9 Conclusion

These are the details of the LISO implementation. Hopefully at this point we have a better idea of what it means to achieve to achieve our overall goal which I have copied below from section 4.1 to refresh our memory. Next we will review some results obtained by the implementation above.

> **Given well-defined cellular automata rules defined by Conway's Game of Life, determine an initial state that produces an optimal final state in terms of growth, diversity, and density after a finite number of iterations.**

Chapter 5

Description of Results

5.1 Introduction

The results from this implementation is rather interesting. Although randomness has been employed to generate initial conditions, there are quite symmetric configurations that are produced. In other cases, there are asymmetric grids that result but still show growth. In order to understand the results given in the remaining of the chapter, we must first specify the global variable values we are using. The definitions of these global variables can be viewed in section 4.8. Furthermore, the way the results are represented is in terms of the initial and final states along with a chart that plots how the fitness grows over the number of generations. The format will follow figure 5.1 and it is understood that the initial state is on the left and the final state is on the right. Other than that, the following shows the exploration of results. I hope you enjoy it!

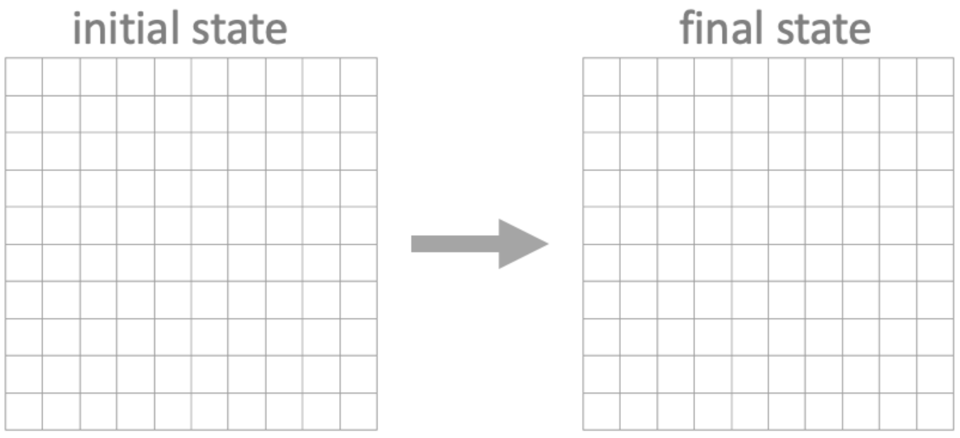


Figure 5.1: Grid Format for Results Interpretation

5.2 Initial Results

Global Variable Values
population — 50
generations — 30
grid_size — 10
border — 3
iterations — 5
mutation_rate — 0.3

When first exploring what this program produces, we start with the values listed above. What I observed from the patterns that emerge are most are symmetric while others are asymmetric. All patterns do show optimization with respect to growth, diversity, and density as described in section 4.4.

5.2.1 Asymmetric Optimal Patterns

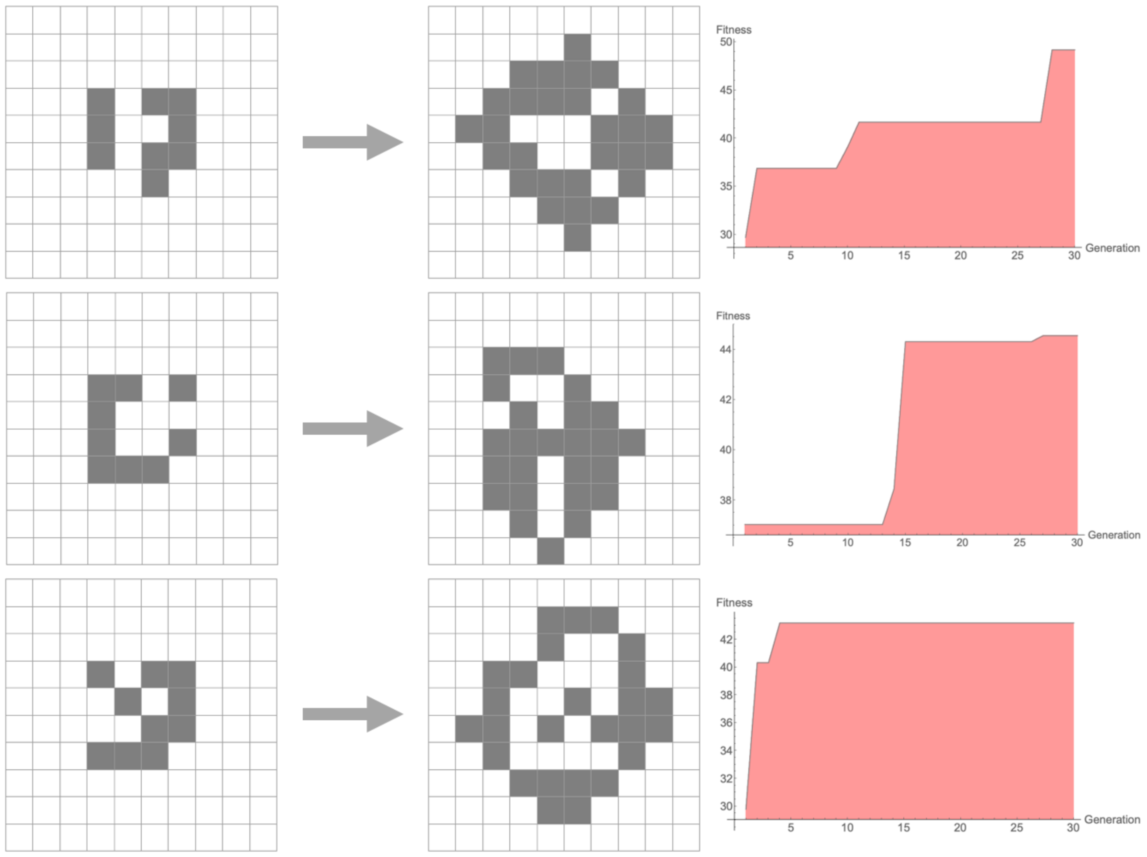


Figure 5.2: Asymmetric Patterns

5.2.2 Symmetric Optimal Patterns

These show three symmetric patterns.

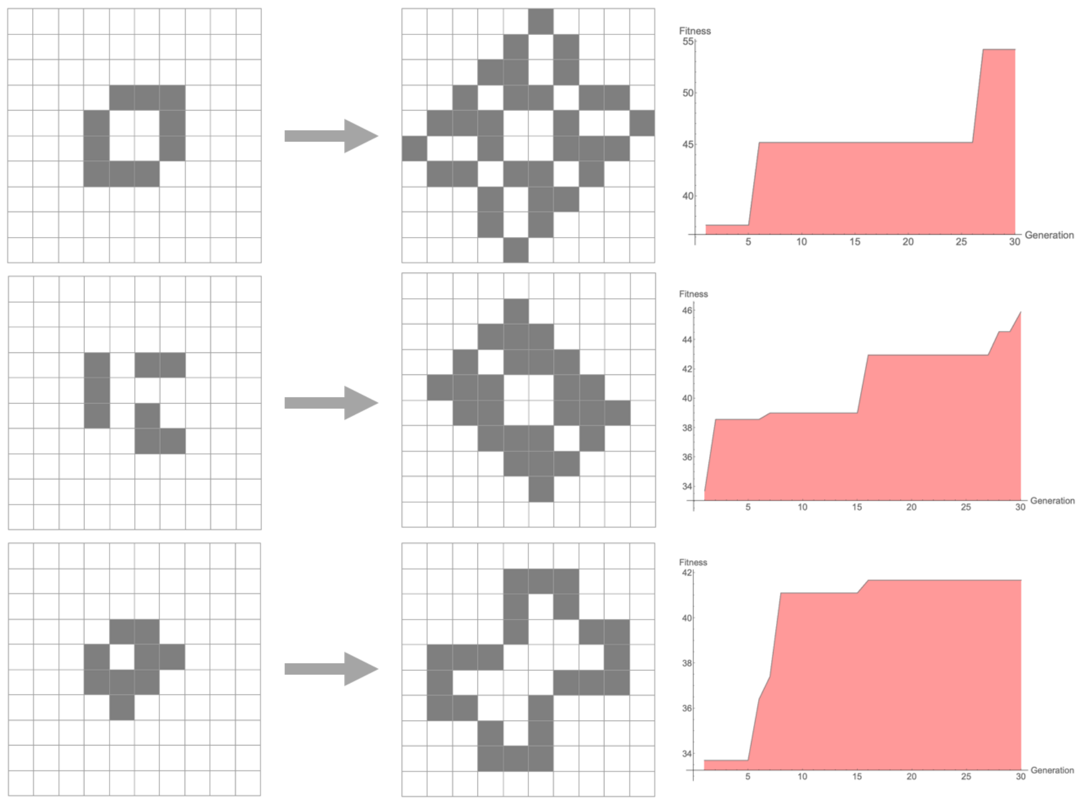


Figure 5.3: Symmetric Optimal Patterns

5.3 Increasing Population and Generation

Global Variable Values
population — 100
generations — 100
grid_size — 10
border — 3
iterations — 5
mutation_rate — 0.3

Now we increase the *population* and *generation* variables to 100. I would like to make three points about the results that occur when increasing these values.

1. The number of optimal solutions that are produced actually decrease when the population and generation variables are increased.
2. The patterns that do emerge are almost always symmetric
3. The patterns are all similar to each other (usually rotations of each other)

This is a rather interesting results. As we can see, the solutions are nearly identical and the only difference in the direction they are pointing. There is also a strong diagonal symmetry in these solutions. From the plots on the right, these solutions converge quickly.

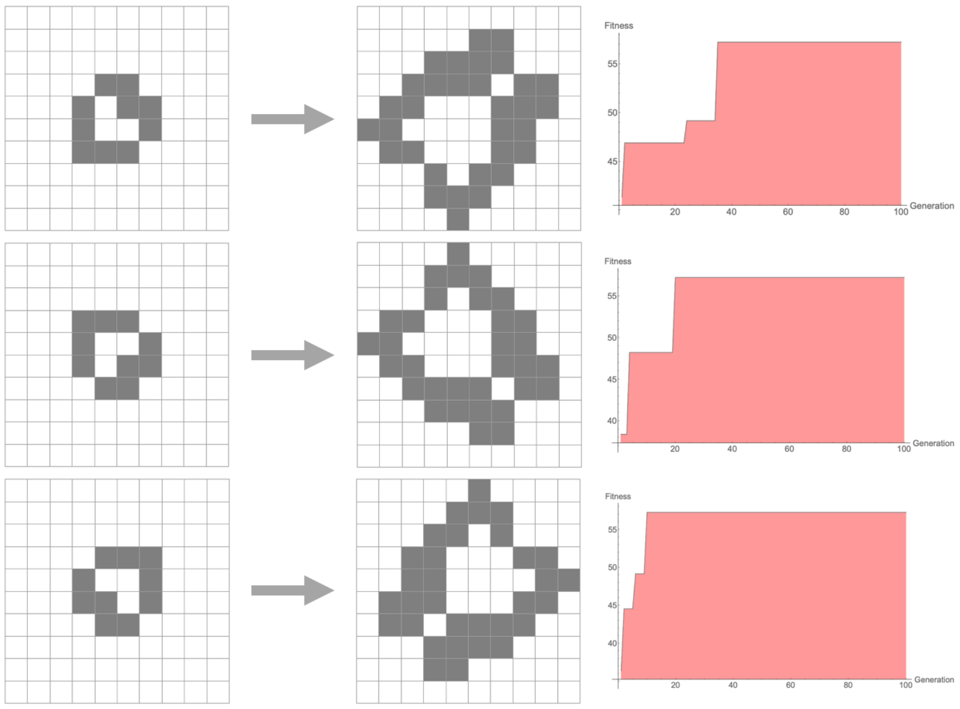


Figure 5.4: Resulting Patterns From Increasing Population and Generation

5.4 Exploring Larger Grid Sizes

Generations: 50

Global Variable Values
population — 100
generations — 50
grid_size — 20
border — 5
iterations — 7
mutation_rate — 0.3

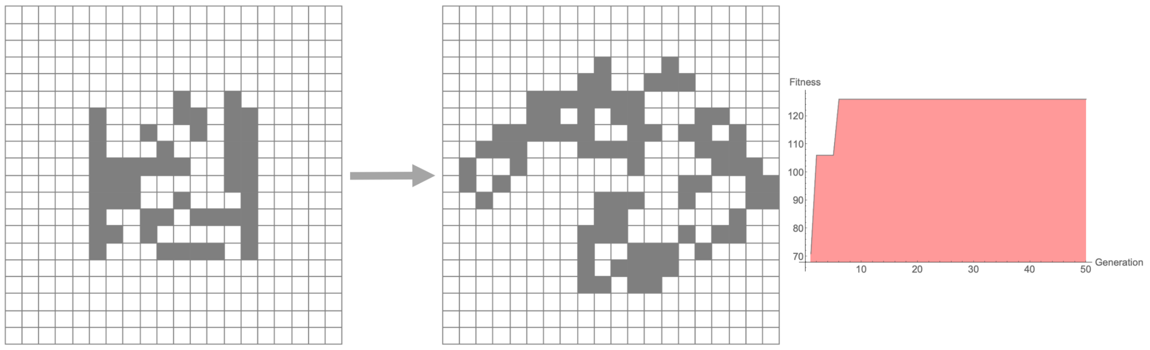


Figure 5.5: Large Grid Optimization

It is only natural we push our program to include larger grid sizes. Since the grids are larger, we can adjust other parameters in order to help us find optimal solutions. The main difference in this case is that the grid size has doubled to 20. In order to make the computation timely, I have decreased the number of generations to 50. The border size has also increased to 5.

Generations: 100

Global Variable Values
population — 100
generations — 100
grid_size — 20
border — 5
iterations — 7
mutation_rate — 0.3

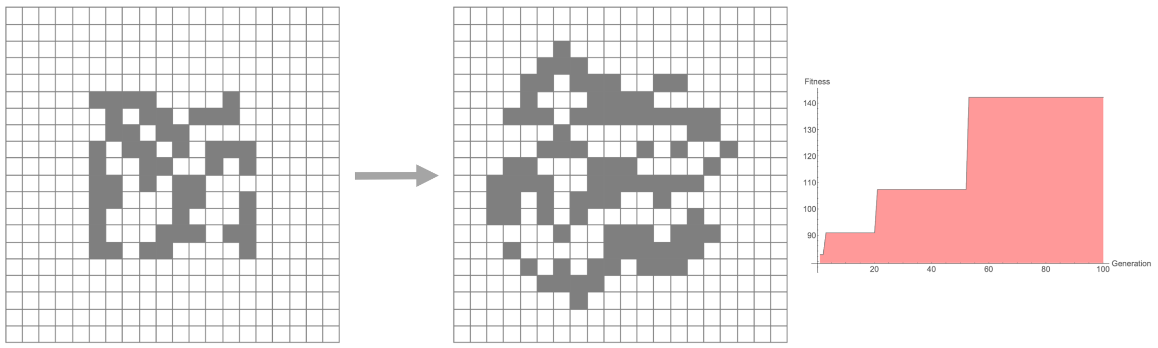


Figure 5.6: Large Grid Optimization for More Generations

5.5 LISO Evolution

In an effort to understand how reliable this evolution is, I have use the parameters listed and ran the program several times while recording the results. Figure 5.7 shows how this evolution tends toward the similar optimal fitness values for different runs of the program.

Global Variable Values
population — 50
generations — 100
grid_size — 15
border — 0
iterations — 5
mutation_rate — 0.3

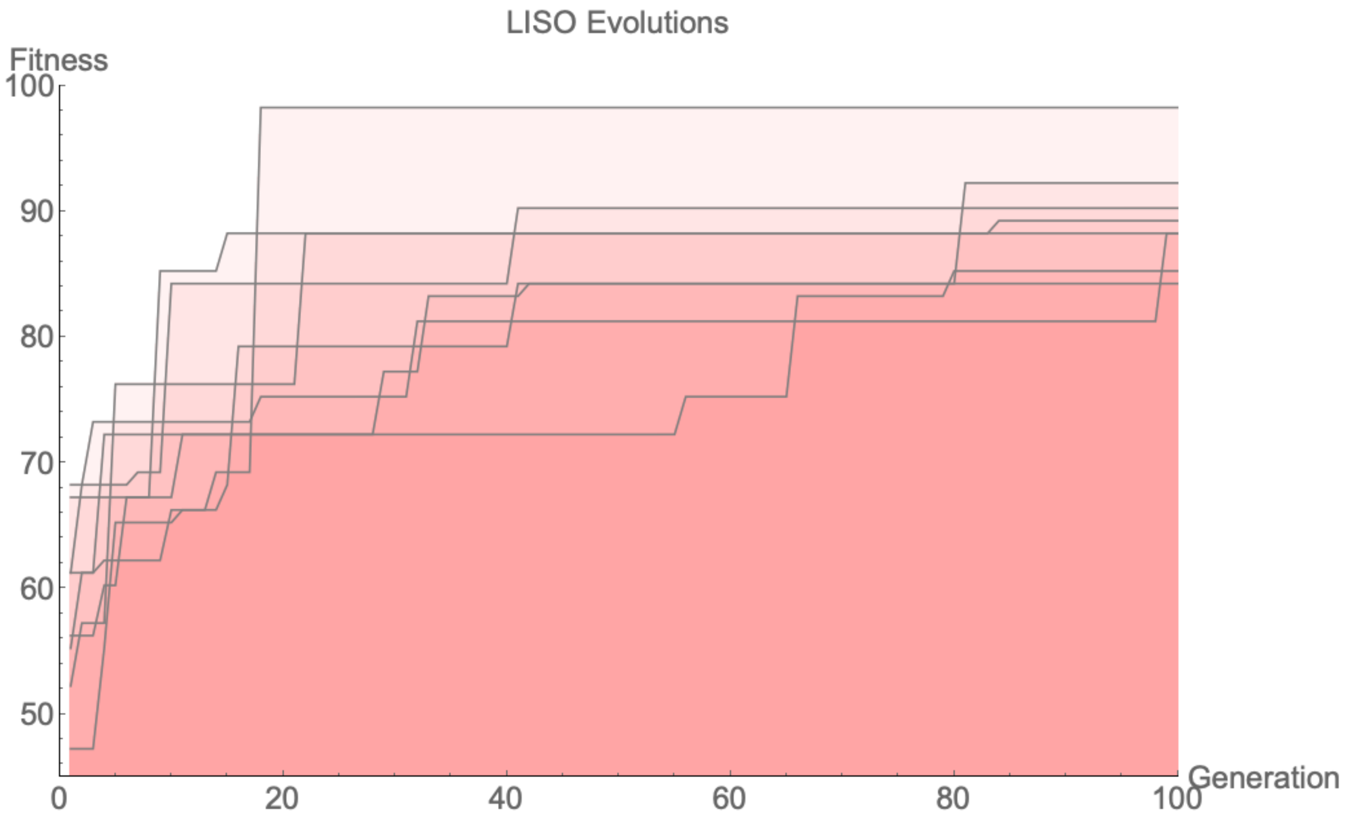


Figure 5.7: LISO Evolution Trends Over Several Simulations

5.6 Conclusion

These are interesting results but it is clear that the search space for optimal solution grows combinatorially and it is difficult to tell if larger grid sizes give successful results. We could, in theory, increase the number of generations and increase the population size to be sure we have suitable results.

Chapter 6

Conclusion, Improvements, & Future Work

6.1 Final Thoughts

It is satisfying to learn that such a powerful method such as genetic algorithms can assist us in tackling this seemingly intractable problem. There are many examples in nature by where patterns emerge via individual component interaction. It can difficult to understand how these patterns emerge given the unpredictability of that is inherent in emergent phenomena. One glimmer of hope is that even a small number of components with a low level of interaction can still lead to interesting emergent patterns. This seems to be a good place to start when understanding how systems evolve in time, particular complex systems.

6.2 Improvements

Of course, there are certainly always improvements we can make when working on a project and this project is no different. I have included a list of potential improvements below. I also welcome any ideas on how to improve this project further.

— Incorporating the Roulette Wheel methodology into the algorithm. The decision to not include it was by design but it may be the case that this improved the results of the study

— Allowing more computational energy be put toward increasing the size of the population and running for more generations in order to explore a larger search space of interesting behavior

— Include more criteria into the LISO fitness definition in order to constrain the problem further — this may be beneficial for larger grid sizes

— Include a class dedicated to producting cellular automata animations and graphics that allows the investigator to more readily interact with the program

— Organize code structure according to Object Oriented Programming paradigms and standards

6.3 Future Work

Throughout the course of working on this project, ideas have to come to mind regarding the potential avenues of exploration for this type of work. Below are a few ideas that I have captured while working.

— Using a different *neighborhood* in the cellular automata grid — the Von Neumann Neighborhood instead of the traditional Moore Neighborhood which is often typical for Game of LIfe implementations

— Using stochastic cellular automata in the *Live* portion of the program — nature tends to operate probabilistically so it may be the case that incorporating this can help product interesting behavior

— Using new rules could be interesting as well — we were using Conway's Game of Life rules but we could have used any rule we wanted

— Consider new ways of *crossover* — in this implementation we used *Single Point Crossover* but perhaps exploring other possibilities can be helpful — especially considering the genes are two dimensional grids instead of 1-D arrays

— Allow program to find interesting behavior on its own — we could perhaps use another genetic algorithm that is used to understand what the term *interesting* or *emergent* means and observe what it produces

— Allowing fitness values to determine how many offspring are produced — allow fitter parents to produce more offspring than the parents with lower fitness

6.4 Contact Information

Feel free to contact me for suggesting improvements or sharing ideas regarding the work presented in this report.

`email: eric.pena@binghamton.edu`

Bibliography

[1] Stephen Wolfram. *A new kind of science*. Vol. 5. Wolfram media Champaign, IL, 2002.

[2] Melanie Mitchell. *Complexity: A guided tour*. Oxford University Press, 2009.

[3] Elwyn R Berlekamp, John H Conway, and Richard K Guy. *Winning Ways for Your Mathematical Plays, Volume 4*. AK Peters/CRC Press, 2004.

[4] E Goldberg David et al. "Genetic algorithms in search". In: *Optimization and Machine Learning, Reading, Massachusetts* (1989).

[5] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[6] Melanie Mitchell, James P Crutchfield, Rajarshi Das, et al. "Evolving cellular automata with genetic algorithms: A review of recent work". In: *Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*. Vol. 8. Moscow. 1996.

[7] Clinton Sheppard. *Genetic Algorithms with Python*. Clinton Sheppard, 2018.

[8] J.E. Smith A.E. Eiben. *Introduction to Evolutionary Computing*. Springer Science Business Media, 2007.

Appendix A

First Appendix — Python Code

A.1 Section 1 — Main Program

```
"""
# :: Author: Eric Pena
# :: December 2019
# :: File: LISO.py
# :: Note: Optimizing Game of Life Program Using Genetic Algorithms
"""

import matplotlib.pyplot as plt
from agent import Agent
import numpy as np
import random

# Global Parameteres
#---------------------------------------------
population = 100
generations = 100
grid_size = 20
border = 5
iterations = 7
mutation_rate = 0.3

filename_fitness = 'test_values.txt'
file_fitness = open(filename_fitness, 'w')

filename_ic = 'optimized_ic.txt'
file_ic = open(filename_ic, 'w')

# :: Create weight matrix for diversity measure in fitness
z = np.zeros((grid_size,grid_size), dtype=np.int)
for i in range(grid_size//2):
    for j in range(grid_size//2):
        z[i][j] = grid_size - 1 - i - j
for i in range(grid_size//2):
    for j in range(grid_size//2,grid_size):
        z[i][j] = abs(i - j)
for i in range(grid_size//2,grid_size):
    for j in range(grid_size//2,grid_size):
        z[i][j] = abs(grid_size - 1 - i - j)
for i in range(grid_size//2,grid_size):
    for j in range(grid_size//2):
        z[i][j] = abs(i - j)
#---------------------------------------------

def init_agents(population, length):
  """
  Initialize a population of Agents

  Args:
    population (int): Number of agents in the population/generation
    length (int): Size of an individual agent
  Returns:
    (list): List of Agents that represent the population
  """
  return [Agent(length, border) for _ in range(population)]

def live(agents):
  """
  Evolve an individual Agent a certain number of iterations

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  for agent in agents:
    for _ in range(iterations):

      agent.update()

  return agents

def fitness(agents):
  """
  Calculate the fitness in terms of growth, diversity, and density

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  for agent in agents:

    growth = agent.final.sum() - agent.state.sum()

    d = z * agent.final
    diversity = d.var()
    density = agent.final.sum()
    agent.fitness = (diversity + density + growth)
  return agents

def selection(agents):
  """
  Select top Agents based on fitness values

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  agents = sorted(agents, key=lambda agent: agent.fitness, reverse=True)

  # :: Selects the top 20% of the agents and removes the others
  agents = agents[:int(0.2 * len(agents))]

  return agents

def crossover(agents):
  """
  Crossbreed top agents to create offspring for the next generation
  This is using single-value crossover

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  offspring = []
  for j in range(int((population - len(agents)) / 2)):

    parent1 = random.choice(agents)
    parent1.print_state()
    parent2 = random.choice(agents)
    parent2.print_state()
    print(f"-----------{j}")
    child1 = Agent(grid_size, border)
    child2 = Agent(grid_size, border)

    row_split = random.randint(border, grid_size - border - 1)

    child1.state[border:row_split] = np.copy(parent2.state[border:row_split])
    child1.state[row_split:grid_size - border] = np.copy(parent1.state[row_split:grid_size -
     border])
    child2.state[border:row_split] = np.copy(parent1.state[border:row_split])
    child2.state[row_split:grid_size - border] = np.copy(parent2.state[row_split:grid_size -
     border])

    offspring.append(child1)
    offspring.append(child2)

  agents.extend(offspring)

  for agent in agents:
    agent.reset()

  return agents

def mutation(agents):
  """
  Randomly change several individual values in each Agent grid

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  family = []

  n = int(0.2 * len(agents))

  for elite in agents[:n]:
    family.append(elite)

  for p in agents[n:]:
    for i in range(int(mutation_rate * grid_size * grid_size)):

      r = random.randint(border, grid_size - border - 1)
      c = random.randint(border, grid_size - border - 1)

      p.state[r][c] = 1 if p.state[r][c] == 0 else 0
      p.final[r][c] = 1 if p.final[r][c] == 0 else 0

    p.fitness = (p.final.var() + p.final.sum())
    p.neighbor_sum()
    family.append(p)

  agents = family
  return agents

def record(agents):
  """
  Record data to filename and print to terminal and keep track of most
  optimized initial configuration

  Args:
    agents ([Agent]): List of Agents that represent population
  Returns:
    agents ([Agent]): List of Agents that represent population
  """
  print('MAX:\t' + str(max(agents, key=lambda agent: agent.fitness).fitness))
  file_fitness.write(str(max(agents, key=lambda agent: agent.fitness).fitness) + ', ')

  return agents

def main():
  """
  Main program utilizes all other methods to optimize IC using GA

  Args:
    None
  Returns:
    None
  """
  agents = init_agents(population, grid_size)
  opt_state = np.zeros((grid_size, grid_size))
  opt_final = np.zeros((grid_size, grid_size))

  for generation in range(generations):

    print('Generation: ' + str(generation))

    agents = live(agents)
    agents = fitness(agents)
    agents = selection(agents)
    agents = record(agents)

    opt_state = max(agents, key=lambda agent: agent.fitness).state
    opt_final = max(agents, key=lambda agent: agent.fitness).final

    agents = crossover(agents)
    agents = mutation(agents)
  print('OPTIMIZED STATE')
  print(str(opt_state).replace('0', ' ').replace('1','+'))
  print('OPTIMIZED FINAL')
  print(str(opt_final).replace('0', ' ').replace('1','+'))

  print('OPTIMIZED STATE')
  print(str(opt_state).replace('0', '0,').replace('1','1,'))
  print('OPTIMIZED FINAL')
  print(str(opt_final).replace('0', '0,').replace('1','1,'))

  # :: Save the optimized initial configuration
  file_ic.write(str(opt_state.tolist()))

  file_fitness.close()
  file_ic.close()

if __name__ == '__main__':
  main()
#----------------------------------END----------------------------------
```

A.2 Section 2 — Gene Class

```
"""
# :: Author: Eric Pena
# :: December 2019
# :: File: gene.py
# :: Note: Defines Agent (gene) class to be used else where
"""

import numpy as np

class Agent:

  def __init__(self, size, border):
    """
    Initializes an Agent object

    Args:
      self (Agent): Single chorosome that represents an IC
      size (int): Size of the grid that represent the CA world
      border (int): Size of the border of zeros surrounding IC

    Returns:
      None
    """
    # : Determine the size of the inner agent
    inner_size = 0
    for i in range(border):
      inner_size = inner_size + size - (2 * (i + 1) - 1)
    inner_size = int(np.sqrt(size**2 - 4*inner_size))

    # : Create randomized inner agent
    inner_agent = np.random.choice(2, inner_size*inner_size, p=[0.5, .5]).reshape(inner_size
    , inner_size)

    # : Place inner agent into larger field of zeros
    s = np.zeros((size, size), dtype=np.int)
    s[border:border + inner_agent.shape[0], border:border + inner_agent.shape[1]] =
    inner_agent

    # : Initialize other object states
    self.state = np.copy(s)
    self.final = np.copy(self.state)
    self.border = border
    self.size = size
    self.locale = self.neighbor_sum()
    self.fitness = -1

  def neighbor_sum(self):
    """
    This will return an array whose elements represent the
    sum of alive neighbors each position has

    Args:
      self (Agent): Single chorosome that represents an IC

    Returns:
      np.array: Array of values that represent sums of neighbors
    """
    neighbors = np.zeros((self.size, self.size))
    for i in range(self.size):
      for j in range(self.size):

        # TOP ROW
        if i == 0:
          if j == 0:
            # Top-Left Corner
            neighbors[i][j] =   self.final[i + 1][0] + \
                      self.final[i][j + 1] + \
                      self.final[i + 1][j + 1]
          elif j == (self.size - 1):
            # Top-Right Corner
            neighbors[i][j] =   self.final[i][j - 1] + \
                      self.final[i + 1][j] + \
                      self.final[i + 1][j - 1]
          else:
            # Top-Center
            neighbors[i][j] =   self.final[i][j - 1] + \
                      self.final[i + 1][j - 1] + \
                      self.final[i + 1][j] + \
                      self.final[i + 1][j + 1] + \
                      self.final[i][j + 1]
        # BOTTOME ROW
        elif i == (self.size - 1):
          if j == 0:
            # Bottom-Left Corner
            neighbors[i][j] =   self.final[i - 1][j] + \
                      self.final[i][j + 1] + \
                      self.final[i - 1][j + 1]
          elif j == (self.size - 1):
            # Bottom-Right Corner
            neighbors[i][j] =   self.final[i][j - 1] + \
                      self.final[i - 1][j] + \
                      self.final[i - 1][j - 1]
          else:
            # Bottom Center
            neighbors[i][j] =   self.final[i][j - 1] + \
                      self.final[i - 1][j - 1] + \
                      self.final[i - 1][j] + \
                      self.final[i - 1][j + 1] + \
                      self.final[i][j + 1]
        # MIDDLE ROW
        else:
          if j == 0:
            # Left Side
            neighbors[i][j] =   self.final[i - 1][j] + \
                      self.final[i - 1][j + 1] + \
                      self.final[i][j + 1] + \
                      self.final[i + 1][j + 1] + \
                      self.final[i + 1][j]
          elif j == (self.size - 1):
            # Right Side
            neighbors[i][j] =   self.final[i - 1][j] + \
                      self.final[i - 1][j - 1] + \
                      self.final[i][j - 1] + \
                      self.final[i + 1][j - 1] + \
                      self.final[i + 1][j]
          else:
            # Middle cell with eight neighbors
            neighbors[i][j] =   self.final[i - 1][j - 1] + \
                      self.final[i - 1][j] + \
                      self.final[i - 1][j + 1] + \
                      self.final[i][j - 1] + \
                      self.final[i][j + 1] + \
                      self.final[i + 1][j - 1] + \
                      self.final[i + 1][j] + \
                      self.final[i + 1][j + 1]
    self.locale = neighbors
    return self.locale

  def update(self):
    """
    This will update the final configuration of the cellular automata
    one iteration given the rules of Game of Life

    Args:
      self (Agent): A single chorosome that represents an IC
      Returns:
        None
      """
      for i in range(self.size):
        for j in range(self.size):

          if self.final[i][j] == 1:
            if self.locale[i][j] < 2 or self.locale[i][j] > 3:
              self.final[i][j] = 0
          else:
            if self.locale[i][j] == 3:
              self.final[i][j] = 1
      self.locale = self.neighbor_sum()

    def update_follower(self):
      """
      This will update the state of the cellular automata one iteration
      given the "follower" rule --> do what your neighbors are doing

      Args:
        self (Agent): A single chorosome that represents an IC

      Returns:
        None
      """
      for i in range(self.size):
        for j in range(self.size):

          if self.final[i][j] == 1:
            if self.locale[i][j] < 4:
              self.final[i][j] = 0
          else:
            if self.locale[i][j] > 4:
              self.final[i][j] = 1
      self.locale = self.neighbor_sum()

    def reset(self):
      self.final = np.copy(self.state)
      self.neighbor_sum()
      self.fitness = (self.final.var() + self.final.sum())

    def print_state(self):
      print(str(self.state).replace('0', ' ').replace('1','+'))

    def print_final(self):
      print(str(self.final).replace('0', ' ').replace('1','+'))

    def movie(self):
      return 0

  #----------------------------------END----------------------------------
```