Evolving Mobile Agents in Conway's Game of Life

Christian Lentz and Ana Espeleta

Macalester College, Comp 484 Fall 2023

1 Abstract

We use a genetic algorithm with two fitness functions and three tiers of reproduction to evolve agents capable of linear movement in Conway's Game of Life. Throughout, we discuss related work, extensions of this work, and applications of modeling with cellular automaton.

2 Introduction

Since the field of computer science emerged, researchers have been interested in the understanding the limits of computational machines. In particular, much of this effort is invested in understanding the capacity a computer has to think and behave in ways which we typically associate with human intelligence. Some of the earliest inquires into answering these questions include Alan Turing's Imitation Game (Turing, 1950) and John von Neumann's proof of a universal constructor (von Neumann, 1966), which laid the groundwork for the field which we now call Artificial Intelligence. These early investigations into thinking machines not only created the broader field of AI, but also paved the way for newer and more specialized sub-fields of AI, many of which lie at the intersection of other disciplines. Perhaps the most well known of these fields is Artificial Life (ALife).

Alife can be quickly summarized as the use of computational techniques to model biological processes and systems. ALife is particularly useful in that it allows scientists and researchers to build highly abstracted models of real biological systems while still capturing the key behaviors and properties of that system. The result is the ability to understand and uncover new insights into biology which would otherwise be impossible. However, the field has humble beginnings, namely with models of cellular automaton. Undoubtedly, the most well known of these models is the Game of Life, which was invented by mathematician John Conway in 1970.

Conway's game is simple, discrete and deterministic. In fact, it has even been described by Conway himself as a "zero player game". Despite the simplicity and the ease at with which one may implement it, the game has continued to draw attention from the research community for its ability to display emergent behavior. However, in recent years, researchers have pushed even further, showing that the Game of Life is capable of creating highly abstracted models of multi-cellular life. Evolutionary processes, symbiotic relationships (Turney, 2020) and even self-sustaining autopoietic structures (Turney, 2021) can be evolved and modeled in the game.

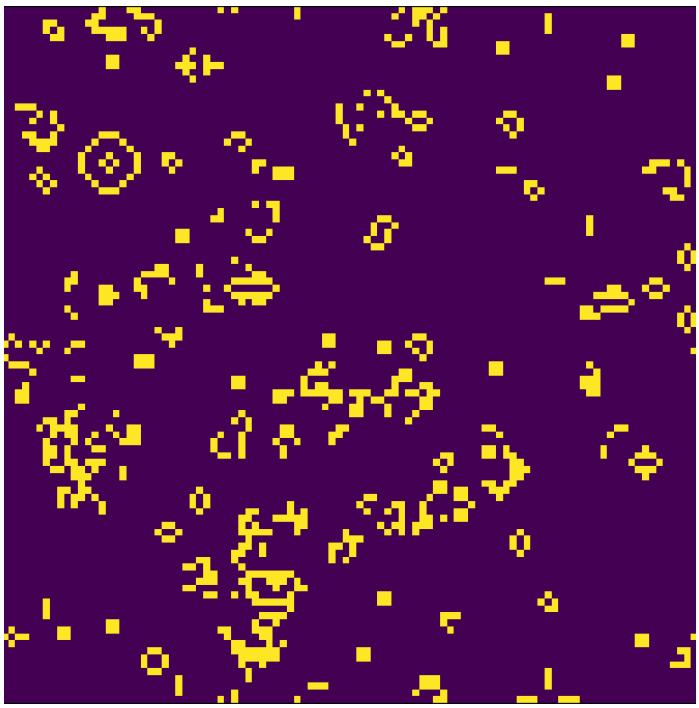

Figure 1: An implementation of Conway's Game of Life.

In this paper, we will build on some of these recent results, especially those of Peter D. Turney, who showed that symbiosis is a key tool in evolving the fitness of a population within the Game of Life (Turney, 2020). While Turney's study judged fitness based on the ability of an agent to grow, we will focus on evolving behaviors similar to those of simple reflex agents. Our approach will be to evolve seeds tilings capable of directed movement using a genetic algorithm and a deterministic implementation of the Game of Life for determining the next generation of agents. Our approach will resemble Turney's *Model-S* (Turney, 2020).

In particular, we are concerned with understanding the limitations of Conway's Game of Life for modeling intelligent agents and the construction of genetic algorithms that allow us to effectively evolve such agents within the game. Through observation of our evolved seed tilings, success will be determined by assessing the agents behavior when compared to that of glider tilings, in addition to assessing the effectiveness of the genetic algorithm at increasing the average fitness of our population. We include an implementation of our model in Python (Lentz and Espeleta, 2023), and a discussion of results and future inquires.

3 Background and Literature

In this chapter, we cover related work and relevant literature. In particular, we will emphasize the rules of Conway's game, local search, genetic algorithms, and Turney's *Model-S*. Although closely related, a full discussion of cellular automaton and artificial life is beyond the scope of this paper.

3.1 Rules for Conway's Game

The earliest cellular automaton models originated in the 1950s, with Stanislaw Ulams lattice network (Ulam, 1962), known as the Ulam-Warburton automaton, and Von Neumann's automaton, which he used to prove the existence of a universal constructor (von Neumann, 1966). Inspired by this work, Conway began experimenting with two-dimensional automaton models in the late 1960s. Like other models before his, Conway sought a set of simple inductive rules which would allow his automaton to produce complex behavior and unbounded growth.

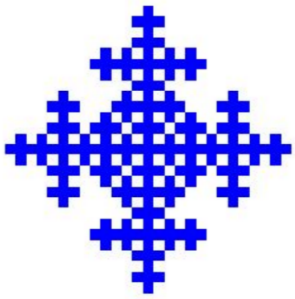

Figure 2: The Ulam-Warburton Cellular Automaton at time step 15 (Warburton, 2019).

By 1970, Conway had found such a set of rules. Consider an infinite orthogonal grid of equally-sized square cells. Pick a subset of these cells to be a population of live cells at some initial time t_0. Repeatedly apply the following rules to produce subsequent populations of live cells:

- Any live cell with fewer than two or more than three neighbors at time t_i will be dead at time t_{i+1}.
- Any live cell with exactly two or three neighbors at time t_i will be alive at time t_{i+1}.
- Any dead cell with exactly three neighbors at time t_i will be alive at time t_{i+1}.

The language of a live/dead cell is used interchangeably with on/off. We use the word *tiling* to describe a specific collection of living cells at some fixed time. At an implementation level, we store information about a tiling within the game in a binary matrix where respectively, on and off cells are represented by ones and zeroes.

3.2 Emergent Behavior

In the years since Conway first introduced the Game of Life, much work has been done to investigate and extend the original game in an effort to understand the emergent behavior it is capable of displaying. The earliest such investigations looked at specific patterns of tilings. Some of the most well known of these include *gliders*, which move continuously across the grid unless interacted with, *oscillators*, which loop between a finite number of tilings, and *still lifes*, which remain fixed unless interacted with.

However, patterns may be much more complex than the three basic classes described above. In fact, it has even been shown that the game itself is equivalent to the Universal Turing Machine

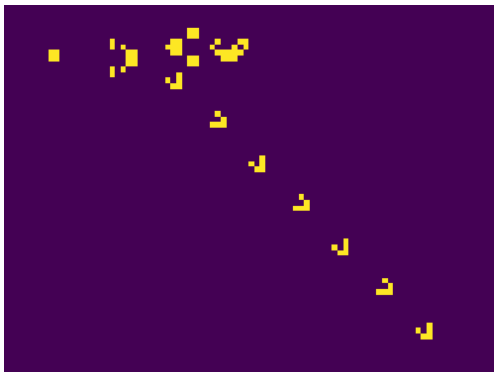

Figure 3: The *Gosper Glider Gun* produces an infinite stream of gliders.

(Berelkamp et al., 1982). Thus, under the assumption of infinite storage and time, there is no theoretical limit to the computational ability of the game. In terms of emergent behavior, this means that the only limitations to what we can model correspond directly to the computational limitations of the machine upon which we model it. However, the caveat is that determining if a given pattern can arise from some initial tiling is undecidable, and has been shown to be equivalent to the halting problem, as noted in Aguilera-Venegas et al., 2019.

3.3 Non-Deterministic Rules

Cellular automaton have been applied in modeling a wide range of problems such as car traffic control (Aguilera-Venegas, Galán-García, and Rodríguez-Cielos, 2014), baggage traffic at an airport (Aguilera-Venegas, Galán-García, Mérida-Casermeiro, and Rodríguez-Cielos, 2014) and the propagation of wildfires (Freire and DaCamara, 2019), among many others. In many of these applied settings, it is often necessary to introduce non-deterministic rules to allow for more realistic scenarios to be modeled and for non-deterministic results to be obtained. Such an extension of Conway's Game of Life, known as the *Probabilistic Cellular Automata Extension of the Game Of Life* (PCAEGOL), was presented in 2019 (Aguilera-Venegas et al., 2019).

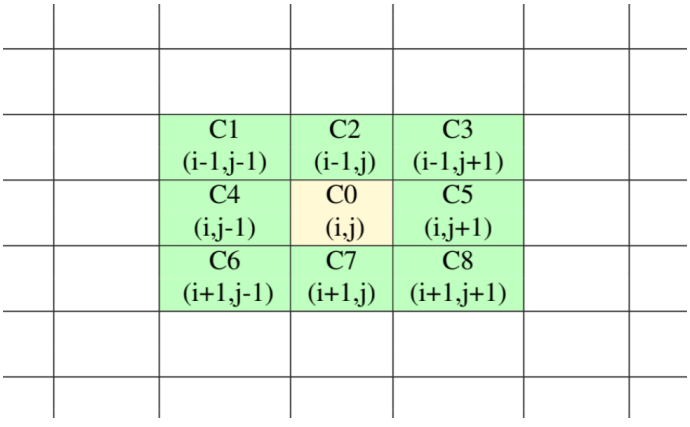

Figure 4: A cell and it's neighbors in the PCAEGOL (Aguilera-Venegas et al., 2019).

The PCAEGOL model includes non-deterministic rules for the transition between successive generations in addition to probabilistic decisions regarding the life and death of cells in the generation which immediately follows the current. Further, the state of a neighboring cell may not be exactly known. The following table describes the scenario.

Let $< pl_i >$ for $i \in \{1 \dots 8\}$ be a vector which describes the probability of C_i being considered alive when it is actually alive. Similarly, define $< pd_i >$ for $i \in \{1 \dots 8\}$ to be a vector which describes the probability of C_i being considered alive when it is actually dead. Trivially, when $pl =< 1, \dots, 1 >$ and $pd =< 0, \dots, 0 >$ we have the classic deterministic version of the Game of Life. By assigning such probabilities to each of the neighboring cells, one can model directed growth within the game, and can describe scenarios where growth or movement in one direction is more probable or favorable than growth or movement in another. In addition, by placing an upper or lower bound on the number of deaths or births that may occur at any given transition between successive generations, one may also model scenarios of high or low density cell growth.

3.4 Genetic Algorithms

Genetic Algorithms (GAs) are programs that model the biological process of evolution. There is great interest in approximating natural processes in order to better understand their complexity and apply novel algorithms to questions outside of the biological realm (Mitchell, 1995).

3.4.1 Overview

John Holland, an American computer scientist and psychologist, is credited with the invention of the simplest version of a GA (Holland, 1992). In the 1960s, his goal was to understand an organism's adaptation as it occurred in natural settings, leading him to seek a way to replicate this phenomena within a computer system. In Holland's GA implementation, he proposes three main components to the algorithm. First, the selection process allows for the transfer of characteristics from one population to a new population by means of "chromosomes", which are represented using a string of bits taking values of either 0 or 1 (representing genes). It involves choosing the subset of individuals in the population that will produce the offspring to form the new population, those who are more fit will be weighted to produce more offspring, while less fit individuals are weighted to produce less or none at all. Once the subset of individuals has been selected, crossover operations are applied to the chromosomes to allow individuals to pair up and rearrange their bits with other individuals in the subset and produce an offspring. Finally, the mutation process makes random bit-level changes to the offspring's chromosome to introduce more diversity into the new population.

As previously mentioned, the application of GAs goes far beyond modeling the natural world. It also serves as an optimization tool for different processes when they require a search through possible solutions. Thus, GAs can be seen as a type of local search algorithm that uses parallelism to simultaneously explore different possibilities in an efficient way (Mitchell, 1995). In the computational realm, computer programs often need to be adaptive, meaning they need to continue to perform well in a changing environment. Other computational problems require innovative computer programs that are able to construct something completely new, such as a novel algorithm for solving a particular computational task. Additionally, GAs are a great "bottom-up" approach for the development of artificial intelligence, as they allow programmers to encode simple rules

that lead to the emergence of more complex behavior.

3.4.2 Components of a Genetic Algorithm

Most algorithms classified as GAs all share a general set of elements: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring. To begin the evolutionary process, one must define a fitness function. In the natural world, fitness is determined by an individual's ability to withstand particular environmental conditions and survive. In an artificial setting, this process can be replicated by defining the set of characteristics that make an individual more fit, which depends on how well it solves the problem at hand. The fitness calculation translates a given bit string into a real number x and then evaluates the function at that value, which gives the fitness score (Mitchell, 1995).

Once the fitness criteria is defined, there are three main operators involved in the algorithm. Selection chooses the chromosomes in the population that will be used for reproduction. With selection, chromosomes are randomly selected from the population with the condition that fitter individuals are weighted so that they have a higher chance of being selected for reproduction (Centre, 2023). The next step is crossover, which mimics the process of recombination between two biological chromosomes and involves swapping parts of each string with the other to generate two completely new individuals. The final step of the algorithm involves the mutation of bits in the new strings. In nature, mutations are crucial for introducing diversity into a population. With a very small probability, mutations may improve an individual's adaptability to a given environment, but most of the time, they do not change an individual's phenotype. In an artificial environment, we can replicate this process by randomly flipping a subset of bits in each string. Mutation generally can occur at every bit position with some minute probability (e.g., 0.001).

3.4.3 A Simple Genetic Algorithm for Bit-Strings

1. Start with a randomly generated population of N L-bit chromosomes, each encoding a candidate solution to a problem.
2. Calculate the fitness $F(x)$ of each chromosome x in the population.
3. Repeat the following steps until N offspring have been created:
    - Select a pair of chromosomes from the current population, with the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.
    - With probability p_c (the crossover probability), cross the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents.
    - Mutate the two offspring at each locus with the probability p_m (the mutation probability), and place the resulting chromosomes in the new population.
4. Replace the current population with the new population
5. Go to step 2

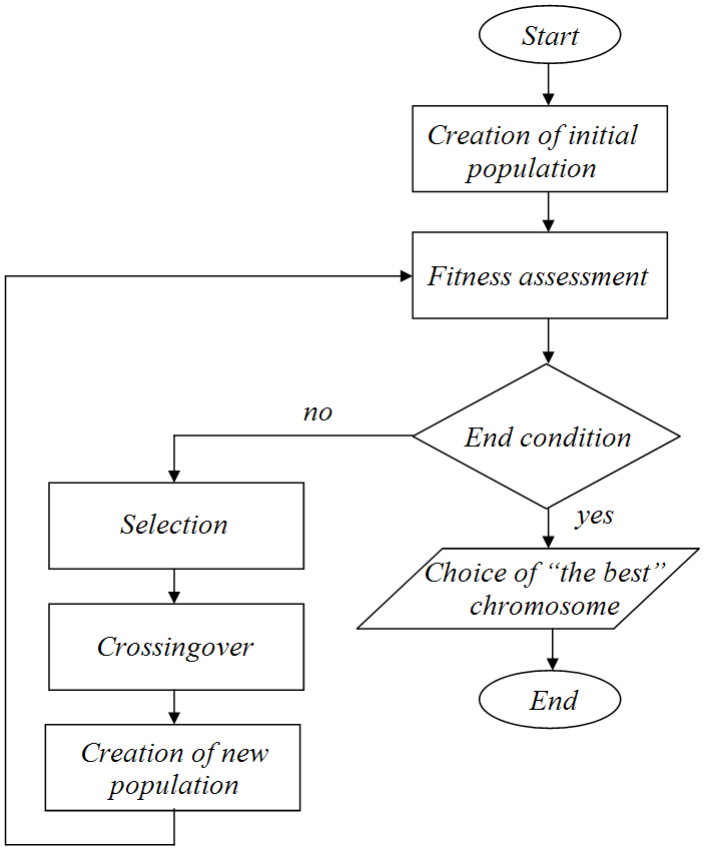

Figure 5: Graphical Representation of the Genetic Algorithm (Selivanov, 2014)

3.5 Turney's *Model-S*: Symbiosis and Growth

Model-S introduces a model of symbiosis, which in an artificial environment is defined as the product of a genetic operator (similar to crossover, selection, or mutation) and is denoted as *genetic fusion* (Turney, 2018). Fusion mimics the process of symbiosis by taking the genomes of two distinct entities that experience selection separately to produce a genome for a merged entity that experiences natural selection as a whole. Although the merging of chromosomes might not always produce better growth fitness at each generation, Turney hypothesized, and later proved in (Turney, 2020), that mutation and selection could adapt them over many generations to work well together.

3.5.1 Layers of Reproduction

Although the main objective of Turney's model is to model the process of symbiosis, *Model-S* is also composed of different layers that are not used for this purpose. Turney created these layers to measure the contribution of different genetic operators to the evolution of a population. The first three layers, which are described as asexual/sexual layers, do not use the fusion operator and instead only vary the mechanisms used for the selection step in order to see their effects on growth. We will primarily focus on the first and third layers, as it provide a good framework for

the kind of behavior we want to model.

Model-S uses a GENITOR-style algorithm (Whitley, 1988) with a one-at-a-time reproduction, a constant population size, and rank-based tournament selection. In Turney's model, fitness is evaluated through a set of one-on-one competitions, where each individual faces off with other individuals a number of times and the winning individual is the one with the largest growth. The amount of wins an individual accrues over the total number of games they play becomes their fitness. An individual in the population is represented as an object containing a binary matrix that specifies a seed pattern, and an array of real values stores a history of the results of its competitions with all other individuals in the population. The population is stored as an array of individuals of the specified size.

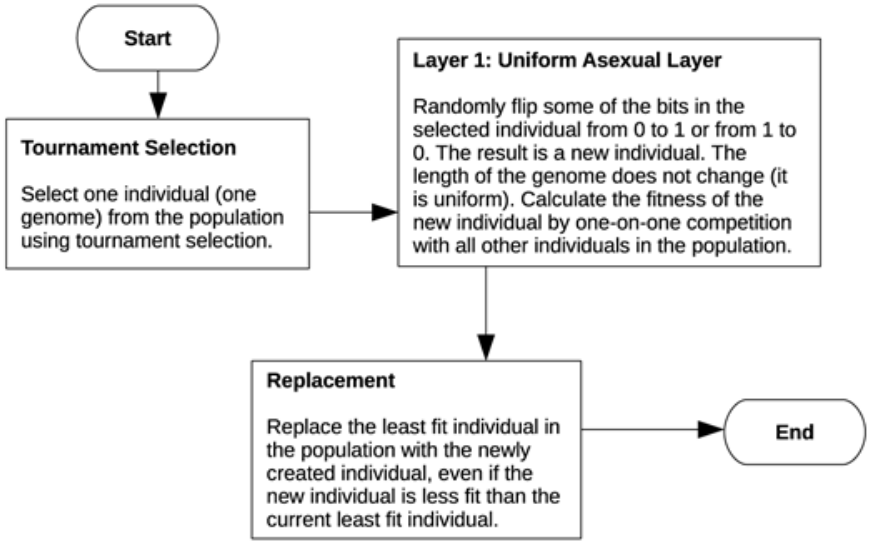

Figure 6: Graphical Representation Model-S first layer (Turney, 2020)

In the first generation, *Model-S* starts with a population in which the binary matrices are randomly initialized. Each individual pattern is a seed, where the probability of there being ones in the matrix is determined by a seed density variable. In the first layer, a constant size of random individuals is selected at each generation and the fittest individual in that set is chosen as a parent. The parent is copied to make a child, which is then mutated by randomly flipping bits in the matrix and used to replace the least fit member of the population. The child's fitness is determined by its performance against all other individuals in a new series of Immigration Games.

Layer 3 adds sexual reproduction to the model, meaning that each child is created from two parents, instead of just one. The first parent is chosen in the same way as in layer 1, but the second parent is chosen by looking for all individuals in the population with a certain degree of similarity to parent 1. The similarity between two individuals is measured by the fraction of corresponding matrix cells that have the same binary values. Two matrices that have different numbers of rows and columns have a similarity of zero. If there are no suitable maters in the required degree of similarity, Layer 3 passes the first parent to Layer 2, for asexual reproduction.

4 Methodology

In this chapter, we describe our work in detail. Following this chapter, we will discuss results, improvements and extensions.

4.1 Rules and Environment

Similar to Conway's game, our simulation will take place in a two-dimensional, orthogonal grid comprised of equally-sized square tiles. Given the finite storage capacity of computers, it is impossible to simulate the infinite grid in which Conway's game takes place. It is common in related work to use toroidial boundary conditions to mitigate storage limitations. However, in our simulation, we will not use any boundary conditions. This is due to the fact that we require a relatively small number of time steps and no interaction between seeds in order to score fitness.

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

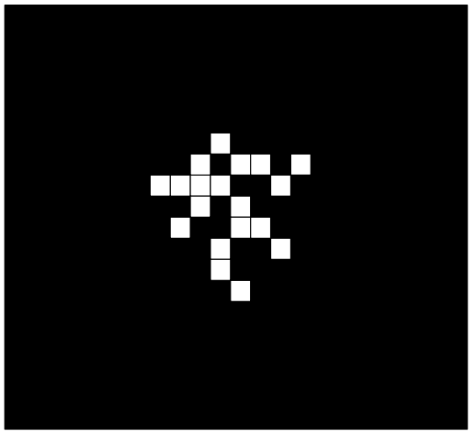

Figure 7: An 8×8 seed with its embedding in a 20×20 environment.

The environment is stored as an $n \times n$ binary matrix, E, originally initialized with all cells off, or all zero entries. While most genetic algorithms encode candidate solutions as some type of string or bit string, our genetic algorithm will encode candidate seed tilings as a randomly generated $m_1 \times m_2$ binary matrix, where $m_1, m_2 \leq n$. Each randomly generated seed has a randomly generated *width*, *height* and *density*. Additionally, we refer to the matrix of a given candidate seed as S_i, where $i \in \{1, \ldots, N\}$ corresponds to a specific seed in a population of N seeds. The *center* of seed i is computed by taking the average row and column indices of living cells, or nonzero entries in S_i, which we use to determine its location within the environment, as in *figure 7* above. The *area* of seed i is the total number of living cells contained in S_i.

There are two benefits to storing the seed and environment separately. First, this allows us to apply Conway's rules and update the matrix E independent of S_i. Thus, we can evaluate fitness of a given seed while still maintaining a copy of the seed at the initial time step t_0 in the matrix S_i. Second, as noted previously, each matrix S_i is at most as large as E. Often, it is the case that E is much larger than S_i. Since we must store N seed matrices, this allows us to save on storage.

Each seed is added to the environment by locating living cells in S_i and updating corresponding

cells in E. Once a seed has been added to the environment, the simulation proceeds according to the rules of Conway's game. This continues for a fixed number of time steps, T. Simultaneously, we record data about the seed, and use this to score fitness.

4.2 Scoring Fitness

As previously mentioned, we wish to evolve seed tilings similar to that of gliders. Thus, there are two general characteristics that our seed tilings must have, including consistent linear movement across time steps and consistent area across time steps. Therefore, we define fitness functions to asses both movement and growth of our evolved tilings using area and location data collected throughout the execution of our genetic algorithm. The overall fitness of any given seed is just the average of these two measures.

4.2.1 Movement Fitness

Suppose a seed tiling S within an $n \times n$ environment, E. S is placed within the environment at time step t_0 and proceeds according to Conway's rules for fixed number of discrete time steps T. At each time step, we record the center location of S within E. Thus, we record a finite sequence of points p_t such that each $p_i \in p_t$ satisfies $p_i \in \{(x, y) \in \mathbb{R}^2 : 0 \leq x, y \leq n\}$.

For any three consecutive points p_{i-1}, p_i, p_{i+1} of the form (x, y) in p_t, define the vectors

$$v_i = \begin{pmatrix} x_i - x_{i-1} \\ y_i - y_{i-1} \end{pmatrix} \qquad\qquad v_{i+1} = \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{pmatrix}$$

which by construction must share a common endpoint. Thus v_i and v_{i+1} form an angle. The movement score at time step i is judged by evaluating the angle formed between v_i and v_{i+1}:

$$\phi(v_i, v_{i+1}) = 1 - \frac{arccos(\frac{v_i \cdot v_{i+1}}{|v_i||v_{i+1}|})}{180}$$

We normalize each score by dividing by 180, since the angle between two vectors cannot exceed 180 degrees. Further, we subtract this score from 1, as angles closer to zero are closer to straight line movement. Further, by normalizing the score relative to a 180 degree angle, we can ensure that a seed will move straight in only one single direction. Using ϕ_i, we can judge the movement fitness of an agent over all time steps. Given the finite sequence p_t, define the function

$$M(p_t) = \frac{k \sum_{i=2}^{T-1} \phi(v_i, v_{i+1})}{(T-2)(T-1)},$$

where $k \geq 1$, is the total number of "good" movement scores throughout the sequence p_t. We consider movement step score to be good if $\phi > 0.95$. The term k allows this fitness function

to assign higher scores to seeds which consistently make good movements. The division by $(T-2)(T-1)$ normalizes the score such that $M(p_t) \in (0,1)$.

The following pseudocode describes this algorithm more explicitly:

```
Algorithm scoreMoveFitness(p_t, steps)
// p_t is a finite sequence, an array, of points in ℝ^2
// steps is the max number of data points that can be scored.
1.   let moveFit = 0
2.   let numPoints = length of p_t
3.   let maxAngles = steps − 2
4.   let k = 1
5.   for i ∈ {1, ..., numPoints − 1}
6.       let prevPoint = P_t[i − 1]
7.       let currPoint = P_t[i]
8.       let nextPoint = P_t[i + 1]
9.       if no pair of the above points is equal
10.          let v_1 = vector from prevPoint to currPoint
11.          let v_2 = vector from currPoint to nextPoint
12.          let stepScore = φ(v_1, v_2)
13.          if stepScore > 0.95
14.              k += 1
15.          moveFit += stepScore
16.  return k * moveFit/(maxAngles)(maxAngles + 1)
```

4.2.2 Growth Fitness

Once again consider the seed S in an environment E. However now suppose that at each discrete time step from t_0 to T we also record the area of the seed. Thus, we have a finite sequence of positive integers which describe how a seed may grow or shrink throughout the simulation. Call this sequence of areas a_t. Given such a sequence, we may perform the following algorithm in order to score the growth fitness, $G(a_t)$:

```
Algorithm scoreGrowFitness(a_t, steps)
// a_t is a finite sequence, an array of positive integers
// steps is the max number of data points that can be scored.
1.    let growFit = 0
2.    let initialArea = a_0
3.    let numAreas = length of a_t
4.    let k = 1
5.    for i ∈ {1, ..., numAreas − 1}
6.        let stepScore = 0
7.        let areaProportion = a_i/initialArea
8.        if areaProportion ≥ 1
9.            if areaProportion ≥ 1.05 and areaProportion ≤ 1.2
10.               k += 1
11.           stepScore = 1/areaProportion
12.       if areaProportion < 1
13.           if areaProportion ≥ 0.85 and areaProportion ≤ 0.95
14.               k += 1
15.           stepScore = areaProportion
16.       growFit += stepScore
17.   return k * growFit/(steps)(steps + 1)
```

For each point in the sequence of areas, we determine how much that area differs from the initial area of the seed. If that area is sufficiently similar to the original, we record a "good" step using the accumulator variable k. Specifically, lines 9 and 13 assess how good the step score is. The conditions provided in these if statements specify that the area must be sufficiently close to the original, but note close enough such that the fitness measure favors still-lifes. Finally, we increment and normalize the growth fitness of the seed, $growFit$, in a similar manner to $moveFit$.

4.2.3 Favoring Persistent Behavior

A final note about how we score fitness regards how we favor "good" behavior which persists. In each iteration of our fitness algorithms, we increment our fitness following the same general pattern. Let $F(a_n)$ be a score of fitness for a seed given a finite sequence of data a_n. Further, let $f(a_i) \in [0, 1)$ be a fitness step score corresponding to data a_i and N be the length of the sequence a_n. In our use cases, N is the number of angles or the number of areas recorded. With these definitions, the previous algorithms compute fitness using the following pattern:

$$F(a_n) = \frac{k \sum_{i=1}^{N} f(a_i)}{N(N+1)} = \frac{\sum_{i=1}^{N} f(a_i)}{N} \times \frac{k}{N+1}$$

where $k \geq 1$ is the number of good step scores along the sequence a_n. The right hand side of this equation is rather useful in understanding how we are able to favor persistent behavior. The first term is a normalized sum which gives an average step score for the seed. The second term describes a score of how the behavior of the seed persisted. Note that the second term is

normalized with a division by $N+1$, as k can be no more than $N+1$. Thus, an optimal score can only be achieved by seeds which repeatedly perform well at each step.

4.3 Our Genetic Algorithm

The GA we developed closely resembles Turney's GENITOR-style algorithm and encompasses three different reproduction layers (Asexual, Sexual, and Sexual with Similarity). As opposed to Turney's Model-S, our approach does not use tournament selection to determine the relative fitness of individuals in the population. Instead, we use the fitness evaluations defined in the previous section to score each individual's fitness and perform a particular selection operation, Roulette Select. The specific components of each genetic operation will be further outlined in this section.

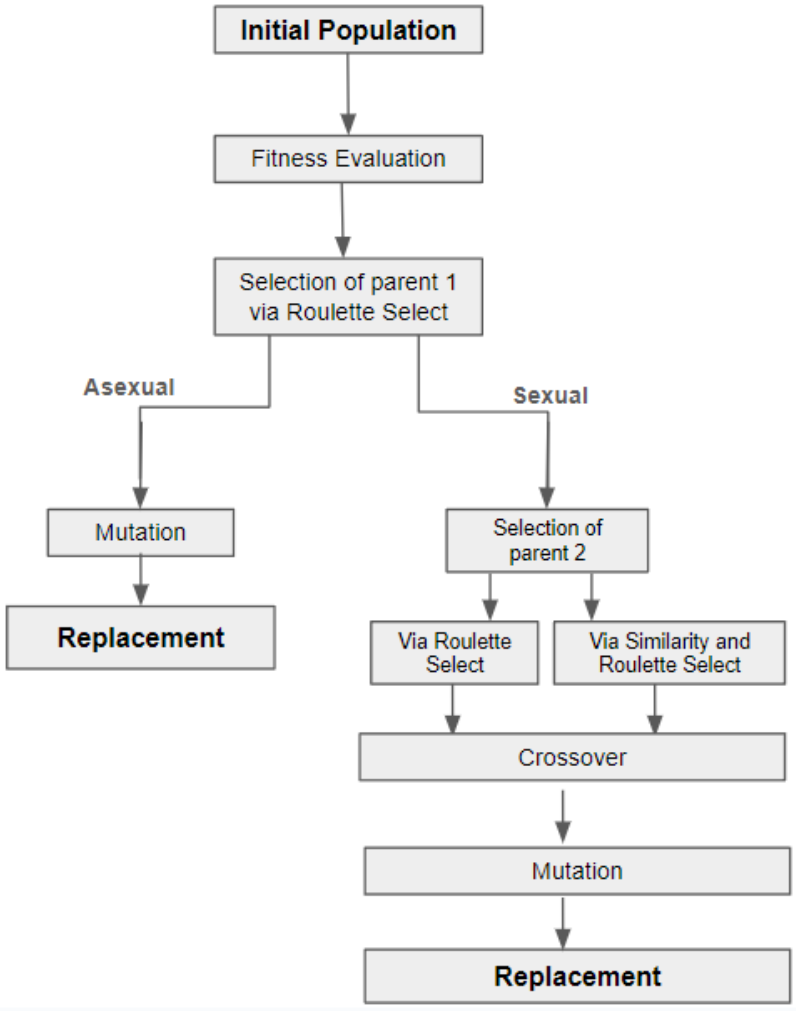

Figure 8: Graphical Representation of our Genetic Algorithm

4.4 Selection

We were particularly interested in modifying one component of Model-S to introduce more diversity into the population. Specifically, we focused on the parent selection process. In Model-S, the parents for each generation are defined to be the most/two most fit individuals in the population. In our model, we use the Roulette Select algorithm, which randomly selects individuals with a probability proportional their fitness. The code for the Roulette Select Algorithm was adapted from an assignment in our Introduction to Artificial Intelligence course, taught by Prof. Susan Fox (Macalester College).

4.5 Asexual Layer: Mutation only

For each layer in our model, we perform the same mutation process. After the corresponding selection or crossover operations, we randomly flip the child seed's bits starting at a rate of 0.01. Additionally, to further preserve diversity in our population, we include a step to alter the mutation rate. To do this, we begin by calculating how much the population's average fitness has changed in the last five steps, and then generate a mutation rate proportional to that change, allowing for a greater probability of mutation when the population fitness has changed very little.

The following pseudocode outlines the mutation algorithm in greater detail:

Algorithm AlterMutationRate($avgFit$)
// $avgFit$ is the current fitness of the population
1. **let** $prevAvgFit =$ average fitness of the population in last five steps
2. **let** $maxMutRate = 0.07$
3. **if** $prevAvgFit$ **not equal to** $None$
4. **let** $percentChange =$ |1 - (avgFitness/prevAvgFit)|
5. **let** $alpha = 1 - percentChange$
6. **let** $mutationRate = alpha * maxMutationRate$
7. $prevAvgFit = avgFit$

4.6 Sexual Layer: Adding a crossover operation

For the crossover (mating) operation, we closely followed Turney's approach of splitting and swapping one subsection of parent 1's matrix for the corresponding subsection in parent 2's matrix, and vice versa. In Model-S, crossover occurs between two seeds, where there is a 50 percent chance that the crossover point will split the matrix along the x axis or split it along the y axis. Because Turney's model assumes both parent seeds to be the same size, and this is not the case for the parents in our model, we defined the child seed's x and y spans according to its parents' minimum spans. This is only one approach for handling size differences, but it could be worth exploring other options for the future improvement of our model.

The crossover operation is outlined with the following pseudocode:

```
Algorithm SexualLayer(parent1, parent2)
// parents 1 and 2 are the seeds selected for reproduction
1.    let xSpan = min(parent1.xSpan, parent2.xSpan)
2.    let ySpan = min(parent1.ySpan, parent2.ySpan)
3.    let probSwap = a random probability
4.    if probSwap < 0.5
5.        let seed1 = parent1
6.        let seed2 = parent2
7.    else
8.        let seed1 = parent2
9.        let seed2 = parent1
10.   let childSeed =  Seed (xSpan, ySpan)
11.   let probSwap = a random probability
12.   let probSwitch = a random probability
13.   if probSwitch < 0.5 and ySpan > 1
14.       let ySplitPoint = random integer  ∈ {0, ..., ySpan − 1}
15.       for  ∈ {0, ..., xSpan − 1}
16.           for y ∈ {0, ..., ySpan − 1}
17.               if (y ≤ ySplitPoint)
18.                   let childSeed.cells[x][y] = seed1.cells[x][y]
19.               else
20.                   let childSeed.cells[x][y] = seed2.cells[x][y]
21.   else if probSwitch ≥ 0.5 and xSpan > 1
22.       let xSplitPoint = random integer  ∈ {0, ..., xSpan − 1}
23.       for x ∈ {0, ..., xSpan − 1}
24.           for y ∈ {0, ..., ySpan − 1}
25.               if (x ≤ xSplitPoint)
26.                   let childSeed.cells[x][y] = seed1.cells[x][y]
27.               else
28.                   let childSeed.cells[x][y] = seed2.cells[x][y]
```

4.7 Similarity Layer: Sexual Layer with Similarity Selection

Sexual similarity adds a form of restricted mating to the sexual layer to further optimize the mating process. It consists of altering the selection method for parent 2, so that it is a more suitable mate for parent 1. We define a degree of similarity (between a minimum similarity of 0.6 and maximum similarity of 0.9), measured by the fraction of corresponding matrix cells that have the same binary values. selection is performed on the similarity subset to select a final seed for parent 2. If no suitable mates are found, the algorithm passes parent 1 onto the asexual layer. This process resembles nature in that many organisms are able to reproduce asexually or sexually depending on their ability to find suitable mates. The proceeding genetic operations remain identical to the asexual and sexual layers.

The sexual similarity pseudocode is outlined below:

Algorithm SexualSimilarity($seed0, population$)
// $seed0$ is a parent seed selected for reproduction
// $population$ is an array containing the current generation of seed objects
1. **let** $minSimilarity = 0.6$
2. **let** $maxSimilarity = 0.9$
3. **let** $similarSeeds =$ seeds most similar to seed0
4. **let** $numSimilarSeeds = len(similarSeeds)$
5. **if** $numSimilarSeeds == 0$
6. **asexual(seed0)**
7. **let** $seed1 =$ **rouletteSelect** $(similarSeeds)$
8. **let** $seed2 =$ **crossover** $(seed0, seed1)$
9. **let** $seed3 =$ **mutate** $(seed2)$
10. **let** $seed4 =$ **findWorstSeed** $(population)$
11. **let** $pos =$ worst seed position in population array
12. add $seed3$ to $population[pos]$
13. **for** $i \in \{1, \ldots, length(population)\}$
14. **updateSimilarities** $(population, pos, i)$
15. **return** $population$

4.8 Experiments: GA Effectiveness

Our initial experimentation was concerned with understanding the effectiveness of the underlying GA, rather than with the resulting evolved seed tilings. Largely, we were concerned with measuring the contribution of each layer of reproduction in terms of increasing the average fitness of the population. We used a population size of 20 in a simulation consisting of anywhere from 1,000 to 60,000 GA steps, where each step results in the birth of a new child.

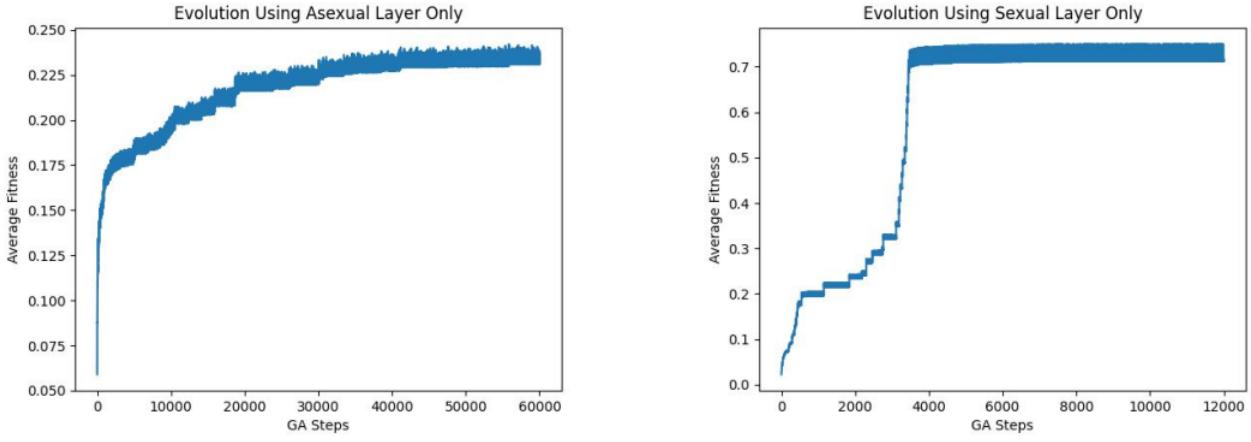

Figure 9: Average fitness for asexual and sexual layers.

As expected, it was clear that the sexual layer was considerably more efficient and successful at increasing the average fitness of the population when compared to the asexual layer. Further, evolving seeds with the target behavior quickly became an intractable problem when using only the asexual layer. Following our initial intuition, the sexual similarity layer had no added advantage when compared to the sexual layer without similarity, which we conjecture is due to the already existing similarity within our randomly generated population.

4.9 Experiments: Evolving Target Behavior

The second step of our experimentation was concerned with the behavior of evolved seed tilings. In particular, we were interested in understanding if such a result was possible, and if so, how consistently the GA could produce this result. Once again, we used a population size of 20 in a simulation consisting of anywhere from 1,000 to 60,000 GA steps.

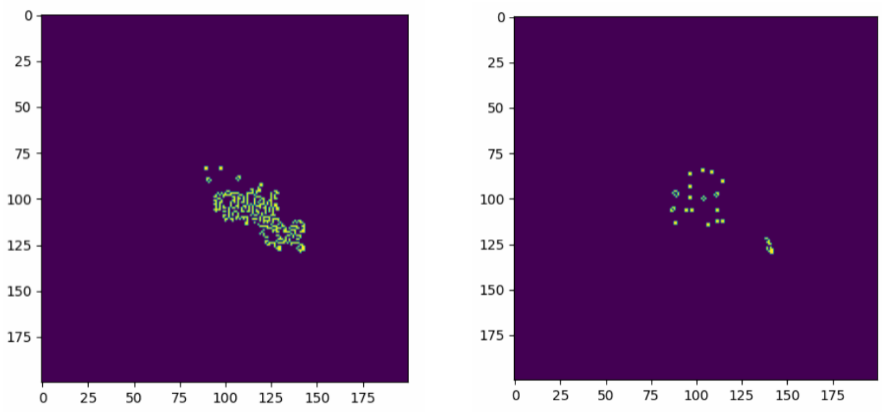

Figure 10: Agent Showing Directed Growth (left) and Agent Showing Glider Behavior (right)

There are a number of results worth noting. First, although we were successful at evolving the target behavior, a minimum of 30,000 GA steps was needed to do so. Secondly, the similarity in seeds which are or may contain gliders with seeds which are still-lifes results in the GA commonly looking for still-lifes and scoring them well, which is a major limitation to this work. Finally, when compared to directed movement, it was rather simple to evolve directed growth. The movement fitness measure worked well on its own. When combined with the growth measure, challenges were more common.

5 Limitations and Future Work

As previously noted, the underlying similarity in still life and glider tilings was a rather evident limitation to this work. We conjecture that further experimentation and modification of the fitness functions is needed to fully resolve this issue. Another existing issue within this work that we wish to resolve is the similarity which exists in the randomly generated initial populations, which we believe impacted the effectiveness of the sexual-similarity layer of the GA. The authors believe there are two solutions to this issue, including changing how seeds are generated, or implementing a sexual layer with a dissimilarity measure.

There are two immediate extensions of this work. The first would be to implement Turney's symbiotic layer to determine if symbiosis promotes fitness in a more general setting, rather than for growth of seed tilings within the Game of Life. Second, we wish to implement a probabilistic implementation of the Game of Life, as is discussed in section 3.3, as this would allow for more expeditious evolution, in addition to allowing for more complex behaviors of agents, including that of reflex agents such as the Braitenberg bots.

6 Conclusions

We have discussed and implemented a GA using two fitness measures and three levels of reproduction. The GA is a GENITOR-style algorithm which includes asexual, sexual, and similarity layers of reproduction, along with roulette selection and variable mutation rate. We have shown that, although limitations exist, it is possible to evolve directed growth and movement within the Game of Life. Throughout, we have related to previous and similar work, and have outlined next steps, extensions and improvements of this work. We hope this work serves to inspire other inquires into both the strengths and limitations that artificial life and cellular automaton models posses in regard to modeling and evolving emergent behavior.

References

Aguilera-Venegas, G., Galán-García, J., Egea-Guerrero, R., Galán-García, M., Rodríguez-Cielos, P., Padilla-Domínguez, Y., & Galán-Luque, M. (2019). A probabalistic extension to conway's game of life. *Advances in Computational Mathematics*, *45*, 2111–2121.

Aguilera-Venegas, G., Galán-García, J., Mérida-Casermeiro, E., & Rodríguez-Cielos, P. (2014). An accelerated-time simulation of baggage traffic in an airport terminal. *Mathematics and Computers in Simulation*, *104*, 58–66.

Aguilera-Venegas, G., Galán-García, J., & Rodríguez-Cielos, P. (2014). An accelerated-time model for traffic flow in a smart city. *Journal of Computational and Applied Mathematics*, *270*, 557–563.

Berelkamp, E. R., Conway, J. H., & Guy, R. K. (1982). *Winning ways (for your mathematical plays)*.

Centre, N. E. D. (2023). Roulette wheel selection. http://www.edc.ncl.ac.uk/highlight/rhjanuary2007g02.php/

Freire, J. G., & DaCamara, C. C. (2019). Using cellular automata to simulate wildfire propogation and to assist in fire management. *Natural Hazards and Earth Systems Sciences*, *19(1)*, 169–179.

Holland, J. (1992). Genetic algorithms. *Scientific American 267(1)*, 66–73. https://doi.org/http://www.jstor.org/stable/24939139

Lentz, C., & Espeleta, A. (2023). Mobile agents ga source code. https://github.com/comp484-IntroToAI/project-ana-christian.git

Mitchell, M. (1995). Genetic algorithms: An overview. *Complexity, 1*, 31–39. https://doi.org/https://doi.org/10.1002/cplx.6130010108

Selivanov, S. e. a. (2014). The use of artificial intelligence methods of technological preparation of engine-building production. *American Journal of Industrial Engineering*, *2(1)*, 10–14.

Turing, A. M. (1950). Computing machinery and intelligence. *In Epstein, R., Roberts, G., Beber, G. (eds) Parsing the Turing Test*, 23–65. https://doi.org/https://doi.org/10.1007/978-1-4020-6710-5_3

Turney, P. D. (2018). Conditions for major transitions in biological and cultural evolution. in proceedings of the third workshop on open-ended evolution (oee3) at the 2018 conference on artificial life. https://doi.org/https://doi.org/10.48550/arXiv.1806.07941

Turney, P. D. (2020). Symbiosis promotes fitness improvements in the game of life. *Artificial Life*, *26(3)*, 338–365. https://doi.org/https://doi.org/10.1162/artl_a_00326

Turney, P. D. (2021). Evolution of autopoiesis and multicellularity in the game of life. *Artificial Life*, *26(1)*, 26–43. https://doi.org/26âft43.https://doi.org/10.1162/artl_a_00334

Ulam, S. (1962). On some mathematical problems connected with patterns of growth of figures. *Proceedings of symposia in applied mathematics*, *14*, 215–224.

von Neumann, J. (1966). *Theory of self-reproducing automata* (A. W. Burks, Ed.).

Warburton, M. (2019). Ulam-warburton automaton-counting cells with quadratics. https://doi.org/https://arxiv.org/pdf/1901.10565.pdf

Whitley, D. (1988). Genitor : A different genetic algorithm. *Proceedings of the 4th Rocky Mountain Conference on Artificial Intelligence, June 1988*. https://cir.nii.ac.jp/crid/1571980075539717120