Facilitating Evolution by Natural Selection in Game of Life

Scott Mueller | Yuchen Yao
June 18, 2021

Abstract

This project furthers the research about symbiosis and evolution in Game of Life conducted by Dr. Peter Turney [17, 18, 19]. Turney's impressive work uses a variant of Conway's Game of Life called Immigration Game. In this environment the fitness of organisms can be tested. These organisms are sets of live cells adhering to the simple rules of Game of Life. Each organism starts from a seed configuration of cells. Think of seeds as analogous to genomes. Turney can then determine fitness by placing a pair of seeds in a game to compete against each other.

The hope is that seeds evolve through reproduction and evolution by natural selection, just as in biological organisms, to a point where life as we know it emerges. Turney's simulations demonstrate that evolution can be promoted through symbiosis by joining organisms for mutual benefit. However, it is unknown how far this evolution will go and whether even basic life can emerge. Is evolution unbounded with the reproductive mechanisms and symbiosis that Turney has devised? Far more generations must be computed to determine this. Unfortunately, further generations require too much computational power.

This paper introduces a new platform for deriving life from Game of Life. Extending Turney's work, Derive Life (`https://derive.life`) is built from scratch in JavaScript. This allows researchers, students, and enthusiasts to easily experiment, modify, and extend evolution through cellular automata. Performance is good, with plenty of opportunity for improvement.

Additionally, a new variant on the Immigration Game was created in order to accommodate micro-communities. If unbounded evolution is to be fostered, some mechanism needs to be in place to allow for the most basic communities. After all, biological life evolved through communities.

1 Introduction

Cellular automata (CA) has fascinated some of the greatest minds since, and including, John von Neumann. Von Neumann created the field with his kinematic self-reproducing automaton [10]. His book, posthumously published, *Theory of Self-Reproducing Automata* [11] became a calling card to scientists and mathematicians eager to build artifical life. Or at least to better understand evolution and biological mechanisms.

There were limits on insights gained, especially considering how few people had von Neumann's brain. Part of the issue was that his automatan was a fairly large grid of cells and rules. The virtual organism comprised of approximately 200,000 cells and 29 different colors (states of each cell). The tail, with 150,000 of those cells, contained the instrument to reproduce. In this way, the tail can be considered as encoding the genome to self-replicate. Amazingly, von Neumann designed this without the aid of a computer!

John Conway wanted to simplify CA as much as possible, while still being Turing complete. Conway's Game of Life (GoL) is the result of this effort. Incredibly simple, there are just two rules and two states. A cell is dead or alive. A dead cell becomes alive (is born) only when exactly 3 of its surrounding 8 cells are alive and a live cell dies unless surrounded by exactly 2 or 3 live cells. Astonishingly, because GoL is Turing complete, these rules allow for anything a modern computer can do. Though it can be quite a challenge to actually simulate modern computers.

Countless variations on GoL have been concocted. Most with the purpose of accelerating artificial life. Unfortunately, artificial life remains elusive. Evolution doesn't happen naturally and easily within GoL.

Dr. Peter Turney has been researching ways to facilitate evolution through symbiosis [19]. His work [17, 18] revolves around a GoL variant called Immigration Game, created in 1971 by Don Woods. Whereas GoL is often considered a 0-player game, Immigration game is considered a 2-player game. Genetic operators are introduced to the Immigration Game in order to simulate evolution by natural selection.

The software and analysis presented in this paper extend Turney's work by adapting Immigration Game to be a 4-player game. The result is a platform for experimenting with evolution by natural selection through CA while allowing qualitative analysis of emerging patterns through visualizations of most fit organisms. Performance exceeds popular GoL platforms. Fitness is unbounded in growth as tested so far.

2 Immigration Game

The Immigration Game uses GoL rules for two different alive cell states. Each of the two states can

be thought of as organisms. The organisms are colored to distinguish them. The additions or changes to GoL are as follows:

- Live cells never change colors
- Born cells take on color of the majority of their 3 neighbors
- The grid is toroidal, which means, for example, seeds growing off the left edge appear on the right
- The grid is set at 25×25 cells

The initial configuration of a color or organism is termed *seed*, analogous to a genome.

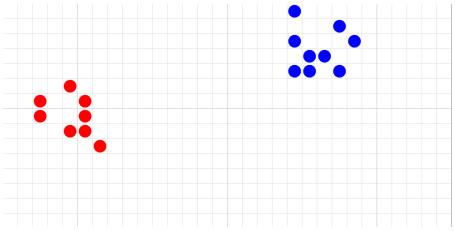

Figure 1: Seeds ready to battle in Immigration Game

A competition in Immigration Game within Golly is pictured in figure 1.

Turney adds several rules to Immigration Game to aid fitness promotion:

- The grid shape is proportional to the sizes of the seeds competing
- The winner is the seed with the biggest *increase* in cells
- No human intervention
- The number of time steps is (grid width + grid height) $\times$ time factor, where time factor is 6 by default

Note that time step here refers to applying the Immigration Game rules to every cell in the grid once. The updated grid is then drawn. Time step is often referred to as a GoL *generation*, but we want to reserve the word generation for when a new set of children are born.

It's important to note that a seed's cells never get taken over in the Immigration Game. The way a seed wins is by replicating more or losing less cells. In a collision between cells of two seeds, the more dominant seed, by way of having more cells around the collision, will more likely be the parent of a new cell. This is because newly born cells take on the majority color of the surrounding 3 live cells.

3 Promoting Fitness

Turney accelerates evolution through natural selection by promoting fitness in the Immigration Game. Fitness scores are evaluated based on the percentage of games won by each seed. More fit seeds have a higher probability of passing their patterns on to new seeds through five different genetic operators:

- Uniform asexual
- Variable asexual
- Sexual
- Fission
- Fusion

These genetic operators and the associated mechanisms will be briefly reviewed below. To provide a good comparison between quad-communities and Turney's use of Immigration Game, all of these mechanisms have been parameterized and included in the new web application introduced here.

The population of seeds is set at 200 seeds. This and many other parameters are configurable by editing Turney's Python source code. However, once the population, or other parameters, is set, it is permanent for the duration of the simulation. A new child seed is born through one of the genetic operators listed above. In order for the population to remain constant, an existing seed needs to be taken out. The least fit seed is then removed.

An initial population of seeds needs to be created. This is simply a 5×5 seed created at random. Each cell has a probability of 0.375 of being alive.

3.1 Uniform Asexual Reproduction

Uniform asexual is the simplest genetic operator. Turney refers to it as layer 1. A child seed is born to 100 potential parent seeds chosen at random. The most fit among those parent seeds is copied. Each cell has a 1% chance of being flipped, dead to alive or alive to dead.

This simulates asexual reproduction in nature with mutation accelerated quite a bit. It's possible, by chance, that no cells end up being flipped. In this case, a random cell is flipped.

No seeds change shape in this layer. If uniform asexual reproduction was the only genetic operator, all seeds would remain with a 5×5 shape. This might seem problematic. Are there too few possibilities for an evolved organism with so few cells? Probably, but bear in mind that a 5×5 seed has $2^{5 \times 5} = 33{,}554{,}432$ possible states.

3.2 Variable Asexual Reproduction

Variable asexual reproduction is still an asexual genetic operator. As such, the most fit out of 100 randomly chose parent seeds gets duplicated to form the child seed.

This time, however, the seed can change size. There's a 20% chance the outer row or column is removed and a 20% chance an outer row or column is added. A new row or column is random based on the 0.375 seed density for new random seeds. The remaining 60% chance falls to layer 1.

The hope is that seed size fluctuations allow for more open-ended evolution. Note that an increase in seed size means the grid it competes in must increase in size and the number of time steps must increase. This is labeled layer 2.

3.3 Sexual Reproduction

The third layer is sexual reproduction. As its name implies, sexual reproduction forms new children seeds from two parents. The first parent is chosen just as in layers 1 and 2, most fit out of 100 randomly picked.

The second parent must be similar to the first parent, but not exactly the same. Similarity is defined as the proportion of cells having the same living or dead state. The most fit among 2 randomly chosen parents that have similarity between 80% and 99% to the first parent becomes the second parent.

The question naturally arises, what happens if no other seeds are 80% to 99% similar. This is not unlikely as seeds with different shapes are automatically 0% similar. Then layer 2 takes over for asexual reproduction. This happens in nature as well! When no suitable mates are available, some biological species will reproduce asexually.

Children seeds are produce through crossover, analogous to combining DNA from parents. A random row or column in the new child is selected. One side of that row or column is filled in from the first parent and the other side is filled in from the second parent. The new child is additionally sent back to layer 2 for flipping, shrinking, or growing.

3.4 Symbiosis

The Merriam-Webster definition of symbiosis is, "the living together in more or less intimate association or close union of two dissimilar organisms." Turney accomplishes this by picking the most fit out of 100 random seeds. Most of the time, a 98.5% chance, this seed is sent back to layer 3. Otherwise, the seed is split or it is fused with another seed.

3.4.1 Fission

Fission is one of two symbiotic genetic operators. With a 1% chance, a seed can be split. The sparsest row or column is the splitting point. A random side is then discarded, as long the minimum seed width or height is maintained. If both sides of the split are too small, control gets sent back to layer 3.
The idea with fission is to counterbalance fusion to avoid better fitness scores from random drift.

3.4.2 Fusion

Fusion is the other symbiotic genetic operator. With a 0.5% chance, a seed can be fused together with another seed. They are connected with a column of dead cells in between. Seeds are randomly rotated first for more variation. The partner to the fusion is selected as the best fit among 100 randomly chosen seeds.

Turney offers two options for fusion by editing his Python code. Symbiosis as persistent mutualism is when the fused organism is more fit than either individual organism. If this flag is set, the fitness of the joined seed is tested before adding it to the population. If the joined seed is more fit than either of the original seeds, it enters the population. Otherwise control goes back to layer 3. This was Turney's intention, although his Python code doesn't actually do this. The fused seed is added to the population always, but if it's not fit enough then layer 3 gets *additionally* activated. This potentially erroneous behavior was copied in the new platform presented below to make better comparisons between Turney's approach.

The second fusion flag is to randomly shuffle one of the two component seeds in a fusion. The same shape and density of lives cells are maintained in the shuffle. The idea is that this changes the structure, but not the summary statistics.

4 Quad-Community

In a quad-community, there are four seeds competing. The grid is split into four quadrants where each seed is randomly placed and rotated/flipped. The hope is that evolution will happen more efficiently, for longer, and more familiar patterns of life as we know it will eventually emerge. Seeds have additional options for survival and flourishing. They can now cooperate and collaborate.

The same genetic operators all apply because they are outside of the rules of GoL, Immigration Game, or this new variant. The only additional deviation from Immigration Game, beyond 4 seeds versus 2, is how live cells are born. It is no longer sufficient to simply choose the majority color of the surrounding cells as the color of the newly born cell. This is because there might be three different colors, no majority. In this case, one of the three colors is chosen at random.

As much as deviations from Immigration Game and Turney's genetic operators were avoided, there were still some fundamental differences. The first of which is how fitness scores are calculated.

In Turney's Immigration Game, a seed's fitness score is calculated as the proportion of games won against every other seed in the population. This means that all pairs of seeds must compete. The minimum number of competitions is:

$$\binom{200}{2} = 19{,}900.$$

However, with a quad-community, competitions aren't just one seed against another. A score can be tabulated based on the proportion of games won against all combinations of the three other seeds. This time the minimum number of competitions is:

$$\binom{200}{4} = 64{,}684{,}950.$$

These are just to initialize the scores. Scores need to be recalculated for each child born.

A generation is considered to be when 200 children seeds are born, and therefore 200 seeds are taken out of the population. One generation is then 200 runs of the genetic operators. So 100 generations is 20,000 runs. It's becoming clear just how problematic scoring is in the quad-community with $\binom{200}{4}$ competitions for fitness evaluation.

There were a couple opportunities here for some performance gains by making the process more efficient. If a seed competes in a quad-round with three other seeds, that seed's score should reflect how it did against each of them. For example, if the seed was first place, it should have a first place score against three separate opponents. It would be far less competitions for each seed to play against each other only once per run. One issue with this approach is that even counting the number of necessary competitions is an unsolved problem, let alone enumerating the competitions' participants. This is known as the Social Golfer Problem (SGP) [13]. Interestingly, this is a complex but solved problem for 3-subsets (triples), known as the Steiner Triple System [20]. Triple-communities might be a worthwhile next project or addition to this project's new platform.

The solution created here was in between SGP and full $\binom{200}{4}$ competitions, neither of which were feasible. Each seed competed with the rest of the seeds partitioned into groups of 3. The only overlap was if the last group had fewer than 3 seeds, then it was filled with randomly selected seeds. The total number of competitions ω among population π seeds to initialize fitness scores is:

$$\omega = \pi \cdot \left\lceil \frac{\pi - 1}{3} \right\rceil.$$

Then the average number of competitions per seed is:

$$\overline{\omega}_i = \frac{\pi \cdot \left\lceil \frac{\pi-1}{3} \right\rceil}{\pi - 1}.$$

The total competitions is close to $\frac{2}{3}$ the number of pairwise competitions. The number of competitions per seed is close to $\frac{1}{3}$ the number of pairwise competitions. So there are some efficiencies in this approach.

In order to distribute the seeds among the competitions as evenly as possible, the population is shuffled for each 3-subset partitioning. The modern version of the Fisher–Yates shuffle [23] was used to shuffle in $O(n)$ time and $O(1)$ space.

Quad-communities further differ from Turney's competitions in that they have potentially first, second, third, and fourth places. A seed that achieves first place half the time and second place half the time is a better fit seed than one that gets first place half the time plus one more time and the rest last place. So second and third places have to be accounted for in the fitness scoring. All-or-nothing scoring also doesn't promote collaboration and cooperation. Table 1 provides the scores used in the new platform. The scores in a competition add up to 2, instead of

Place	Score
1^{st}	1
2^{nd}	$\frac{2}{3}$
3^{rd}	$\frac{1}{3}$
4^{th}	0

Table 1: Fitness for 4-way competitions

1. This is good because the average score is still 0.5. Quad-communities can be effectively compared with Turney's Immigration Game.

Finally, ties are accounted for by adding up the associated point amounts and dividing evenly. For example, for a 3-way second place tie, scores are distributed as 1 for first place and $\frac{1}{3}$ for the rest. For a 2-way first place tie, each first place winner gets $\frac{5}{6}$.

4.1 External Fitness

The problem with fitness scores described above is that they are relative to their population. Analyzing relative fitness scores of the top seeds upon each generation will show them sometimes actually decreasing despite true fitness increasing! The reason for this is that the top seeds do get better, but so do the rest of the population. This is especially so because the top seeds are more likely to be replicated (and mutated to some degree). In this way, competitions get more interesting. This is analogous to a sport that is dominated by a particular player. Eventually other players learn, adapt, and improve until the particular player is no longer as dominant.

Then how can we measure how fit a seed is? Turney proposed two external fitness measures that don't suffer from this problem. The first is an *absolute fitness* score. This is simply having each seed compete against itself but with its cells shuffled. This is simple and effective. Turney's results in figure 2 demonstrate that elite (top 50) seed fitness increases, as expected, except for layer 1. This doesn't translate perfectly to the quad-community environment. A seed can compete with three shuffled versions of itself, but that seed may have evolved a mechanism to cooperate with other structured seeds. It will now get a biased fitness score. A seed could be placed with one or two other evolved seeds and a shuffled version of itself. Those one or two other evolved seeds can record their scores as well at the same time, an efficiency gained. However, we then have the same problem as relative fitness scores, they depend on the fitness of the other seeds.

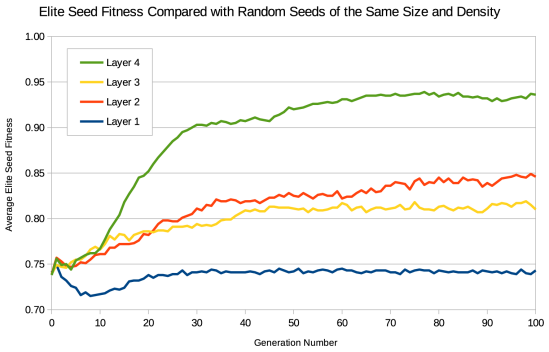

Figure 2: Elite seed fitness vs generation

The second external fitness measure Turney proposed is based on competitions among the seeds at each generation with the highest relative fitness score. Among the advantages, this approach overcomes the limitation of an absolute fitness score being a probability between 0 and 1. The issue is with *slope consistency*. Continuous increases in fitness get closer and closer to 1, where the slope gets closer and closer to 0. Whereas initial increases of the same magnitude have relatively high slopes. This new fitness score, f_n, is unbounded. Its range is $-n$ to n, where n is the generation number:

$$f_n = \sum_{i=0}^{n-1} (2 \cdot p_{\text{in}} - 1)$$

where p_{in} is the proportion of wins the top seed in generation n has against the top seed in generation i. Seeds are randomly placed and rotated/flipped on the grid when competing, so it makes sense to have the same two seeds compete many times. Turney's results in figure 3 demonstrate that unbounded seed fitness has slope consistency. More importantly, his layer 4 appears to keep increasing fitness in perpetuity.

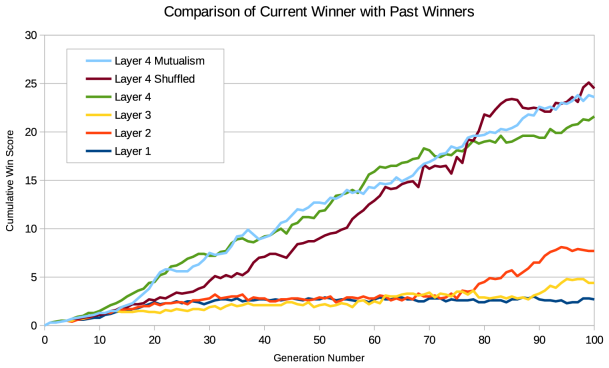

Figure 3: Unbounded seed fitness vs generation

The solution presented here is to apply Turney's unbounded fitness estimator f_n to competitions with the top seed in generation n, the top seed in generation i, and two random seeds with the same shape and live-cell density as the top seed in generation n. This inherits all the advantages of Turney's unbounded fitness score, with only a mild potential bias against seeds that do well with zero or two evolved seeds. This should allow fitness comparisons with Turney's method of pairwise competitions.

5 How to Play

The web application is ready to play. Simply click on `https://derive.life` or type it into your browser. Figure 4 shows a screenshot of the main action area. Four seeds, prepared to fight or collaborate, are seen

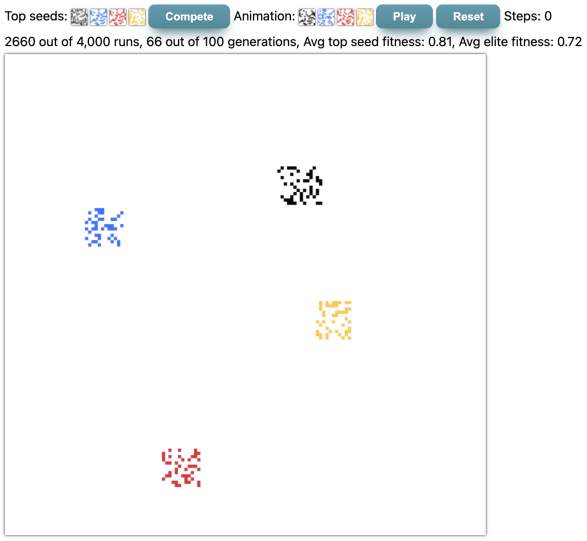

Figure 4: Quad-Community platform screenshot

before the battle begins. They are drawn from the top four most fit seeds of a prior generation. The current generation's top seeds are being computed in background threads. One can imagine, as generations pass, that these seeds grow and start to resemble a 2-dimensional DNA instead of the double helix.

At the top of the screenshot is a panel with status information and actions. The top four seeds of the most recently completed generation is displayed. These are not necessarily the four seeds that are being played in the competition below. That animation is playing the seeds listed next to "Animation". When you click "Compete", the top seeds become the competitors in the animation. Click "Play" to watch the seeds in action. "Reset" will position, rotate, and flip seeds randomly before starting over. The number of runs is listed, this is how many child seeds have been created. A generation passes when the the number of new children created equals the population. In this case, since there's a total of 4,000 runs and 100 generations, the population must be 40.

Notice the average top seed fitness. This is the relative fitness discussed in section 4.1 for the top four seeds displayed on the top-left. Similarly the average elite fitness is for the top *elite size* seeds, where *elite size* is a parameter set in the sidebar. Since elite size was set to 10, the average is a little lower than the top four's average. This number will sometimes fall even when seeds are getting more fit, for reasons mentioned in section 4.1. A much better metric for fitness is the unbounded f_n applied to quad-communities, but that currently requires a separate analysis.

Figure 5 is the left sidebar where parameters for the evolution are set. When ready, click "Start" to begin the background threads. In this screenshot, the

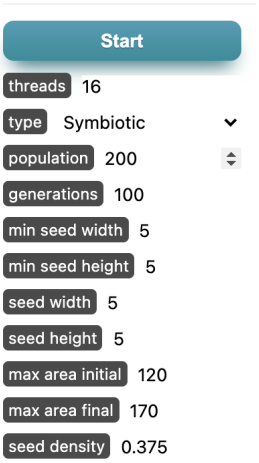

Figure 5: Quad-Community platform sidebar

thread count is set to 16. Unless your CPU is capable of concurrently running a tremendously large number of threads, be prepared for your computer fans to start spinning full blast. The threads parameter can be set directly below the Start button. Note that animating a competition to visualize how top seeds behave consumes very little CPU utilization. These background threads are for computing each generation, they have nothing to do with the animation.

The type parameter allows you to change which layer of Turney's genetic operators will be applied to every run. As discussed in section 3.4, control may be passed down to lower layers if, for example, the fission split produces two seed halves that are too small. This type parameter allows you to set any layer as the default genetic operator for every new child seed.

Below the thread count and genetic operator type are 27 additional parameters listed in table 2. Hovering a mouse cursor over the parameters in the sidebar will pop up a description as well.

This game is really fun to play. Try changing seed widths and heights to see what interesting patterns and behaviors emerge. A fair warning, the computational demands can be high with increasing popula-

Feature	Explanation
population	Constant number of seeds
generations	Population size # seeds born
min seed width	Seeds can't get thinner than this
min seed height	Seeds can't get taller than this
seed width	Seeds start at this width
seed height	Seeds start at this height
max area initial	Seed area, increases each run
max area final	Seed area, increases up to this
seed density	Proportion of live cells
width factor	Multiply max seed width ⇒ grid
height factor	Multiply max seed height ⇒ grid
time factor	× grid width + height = steps
tournament size	# random seeds to compete
mutation rate	Flipping cells in uniform asexual
probability grow	In variable asexual reproduction
probability shrink	In variable asexual reproduction
probability flip	In variable asexual reproduction
elite size	# most fit seeds for avg fitness
min similarity	Min % same cells to mate
max similarity	Max % same cells to mate
random seed	deterministic or -1 for random
trials per run	# of repeat competitions
probability fission	Prob to split seed in symbiosis
probability fusion	Probability to fuse seeds
fusion shuffle	Shuffle cells in fused seed?
persist mutualism	Is fused pair more fit than seeds?
perf stats	Various stats printed in console

Table 2: Quad-Community parameters

tions and seed areas. Of course, the most interesting evolution occurs in large populations.

6 Platform

Golly is a popular platform to run simulations on and supports Python scripting. This is what Turney used for his model. Golly was not sufficient for this project and the desire to rewrite the entire platform in JavaScript came about for the following reasons:

- Existing platform and code was buggy
- There are many parameters to tinker with
- The web allows the platform to run anywhere
- Easier plugins and extensions
- Parallelization is important
- Visualization was very poor

Golly quickly crashed with Turney's Python scripts on ten different computers. This problem was narrowed down to Turney's use of numpy, which appeared incompatible with Golly. Possibly Turney had used specific previous version of Golly, numpy, and Python 2, but couldn't find a working combination. After finally getting it working on a MacBook, confidence in using this platform worsened when much of the Python code had bugs. Considering that the Python code needed to be rewritten anyway, JavaScript became an easy answer.

A usable UI was a priority because experimentation is much more fun and productive when the mechanics are easy to tweak. Additionally there was a potential need to tweak some parameters in real-time. Derive Life provides these features.

The web is a fantastic vehicle for delivering these kinds of applications. Dependencies, such as the problematic Golly and numpy above go away. Future plugins and extensions can be as simple as clicking an upload or edit button.

With the goal of seeing how far evolution by natural selection can go in GoL, parallelization is a core piece of the puzzle. Unfortunately, Golly and Python didn't seem to support multithreading. With powerful modern CPUs able to manage hundreds of threads simultaneously and modern GPUs having the potential to compute millions of numbers simultaneously, this was a big opportunity. Derive Life supports a user-specified number of threads to divide its computations among.

The visualization Golly provided was completely useless. States of games flashed across the screen so fast that it was impossible to understand what was going on. How were users supposed to visualize the evolution of top seeds? Qualitative answers on the behavior and traits of seeds as they evolve is a critical aspect of research in this area. Derive Life was designed to let background threads process evolution by natural selection strategies, while the foreground threads could analyze top seeds by visualizing their actions through GoL animations.

6.1 Optimization

The wonderful aspect of JavaScript is that the architecture allows for many optimization strategies. Unfortunately, most GoL optimal algorithms were written in low-level languages like Rust and C/C++. The reason being that clever tricks not available in a high-level language like JavaScript, such as bit manipulation on CPU/GPU registers, can offer profound speedups. In fact, Rust's official book [16] offers a guide to writing an efficient GoL implementation with WebAssembly.

Low-level languages can still be taken advantage of with Derive Life's architecture. Background threads can easily be offloaded to a cloud server. This cloud server can be very high-powered and use compiled

native code. In this way, dramatic speedups can be provided through client-server communication. Users would never know the difference aside from massively faster generations without taxing their laptop, desktop, tablet, or even mobile phones! This has not been implemented yet in Derive Life, but holds fascinating options for future research on the platform.

The bulk of performance gains come from advanced GoL algorithms, hardware support, potential machine learning (ML) and stochastic approaches, and parallelization. First, let's review some basic algorithmic techniques employed.

6.1.1 Algorithmic

Software algorithms were adapted to quad-communities to optimize performance. Although none of the following algorithms individually cleared a performance bottleneck, together they did improve efficiency noticeably.

A JavaScript library was needed to handle basic linear algebra matrix operations as well as random number generation (RNG). The reason to use a library versus vanilla JavaScript array indexing and `Math.random()` is that these operations are performed many times per seed per competition per run. Given that there are many seeds, competitions, and runs, peak performance on these operations becomes important. Additionally and surprisingly, JavaScript doesn't have random seeding functionality. Though there is a stage 1 proposal in the works [1]. The lack of seeding limits our ability to have repeatable and deterministic evolution.

Unfortunately, popular linear algebra JavaScript libraries, such as math.js, were all orders of magnitude slower than simple solutions implemented in vanilla JavaScript. It was far more efficient to create single-dimensional row-major order arrays representing 2-dimensional seeds, grids, and other objects. Table 3 compares some operations with math.js.

Function	Random seeds	Time
`math.subset()`	10,000,000	364.873s
* `Array.slice()`	10,000,000	0.898s
`math.pickRandom()`	10,000,000	24.404s
* `Math.random()`	10,000,000	2.555s

Table 3: Vanilla JavaScript (starred) vs math.js

6.1.2 Hashlife

Hashlife is the most famous GoL optimization due to its clever architecture and wild performance gains. Figure 6 from [9] is the 6,366,548,773,467,669,985,195,496,000th step of Paul Rendell's Turing machine in Game of Life. Shockingly, this was calculated in the most popular GoL platform, Golly, using Hashlife in just seconds on a now ancient Intel Core DUO (circa 2006). Furthermore, this algorithm was created by Bill Gosper [5] back in 1984 and nothing has come close to its performance since. As a side note, Gosper was also the discoverer of glider guns and buffer, key to creating a Turing machine in GoL. Hashlife is the natural choice to overcome current research's performance limitations.

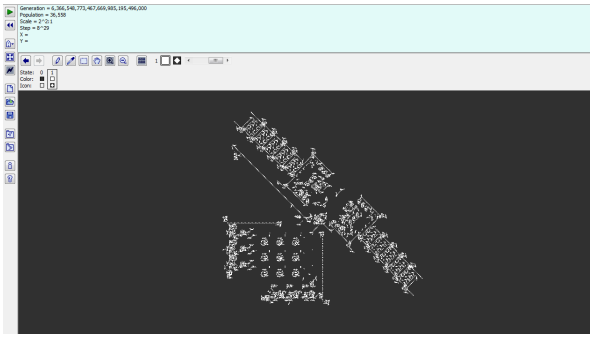

Figure 6: 6,366,548,773,467,669,985,195,496,000th step of Turing machine in Game of Life

Hashlife works by memoizing (caching results) of a group of cells, known as a macrocell. The idea is that a $2^n \times 2^n$ macrocell will predict the $2^{n-1} \times 2^{n-1}$ inner macrocell hovering at the center. Hashlife hashes the outer macrocell and can immediately predict the inner macrocell. Note that the inner cell is 2^{n-2} cells away from any edge. Since GoL cells only change based on neighbors one cell away, the inner cell is predicted 2^{n-2} time steps into the future!

Unfortunately, the standard implementation of Halflife doesn't work so well in the quad-community variant of GoL. The complete algorithm is fairly complex and makes clever use of otherwise ballooning memory requirements. Yet, large amounts of memory can still be necessary. Our environment, with five different states for each cell (dead or one of the four seeds), significantly increases hashing and memoization memory requirements in two ways.

The first concern is that five states present problems for both the storage of the predicted macrocells and the number of hashes that need to be maintained.

As an example, a 3×3 GoL grid comprises of $2^{3\times3} = 512$ states. The base 2 is because there are only 2 states for each cell. With 5 states, this small 3×3 GoL grid comprises of $5^{3\times3} = 1{,}953{,}125$ states. One can imagine the explosion of states in any reasonably sized macrocell.

Additionally, care must also be taken with Hashlife and all hashing algorithms as applied to the quad-community environment. Quad competitors necessitate some stochasticity in new cells being born. In particular, when a dead cell has 3 live neighbors, all of different colors.

Hashlife is great for predicting GoL states very far out into the future, but the overhead necessary makes it non-optimal for the 540 to $2,000$ time steps a typical competition takes. Nevertheless, Hashlife holds great promise for being able to see evolution dramatically beyond the number of generations evaluated so far. It should be able to be adapted for relatively short numbers of time steps in a quad-community environment through clever changes to the algorithm.

6.1.3 Lookup Table

Lookup tables [15] are essentially giant arrays indexed by a number that represents an $m \times n$ subgrid in GoL. The values of the array correspond to the $m-2 \times n-2$ subgrid produced in the next time step. This works very well, as long as you are clever about bit manipulation to create array indices from a grid. Table 4 shows the array size for various subgrids. Standard GoL only has 2 states per cell: alive or dead. This can be represented as just 1 bit. Immigration Game has 3 states per cell: blue, red, or dead. This requires 2 bits. Quad-communities have 5 states per cell. Unfortunately, 5 just misses the cutoff for representing states in 2 bits, so 3 bits are necessary. This means 8 states per cell would still use the same memory. There are two components to be aware of in an array data structure for a lookup table, the number of elements in an array and the size of each element. Table 4 represents these in the Indices and Values columns, respectively. The total size of the lookup table comes from the number of elements multiplied by the size of each element. Keep in mind that modern computers typically only have memory sizes of 8, 16, 32, and 64 bits. So, for example, even though a 4×4 grid of 5 states only has 2^{12} values for each element, that must be stored in a 16-bit (2-byte) integer. That's why the size is $2^{48} \cdot 2 = 2^{49}$. However, this size basic lookup table isn't actually feasible as 2^{49} bytes $\approx$ 563 terabytes of RAM! A 3×4 grid is a stretch at 2^{36} bytes $\approx$ 69 gigabytes of RAM, but a powerful server can handle that. Note that the 5×5 grid for Immigration Game (3 states per cell) uses 3-byte values. Computers generally don't store 24-bit numbers, but this can be split into 2 separate 8-bit and 16-bit numbers.

States	Bits	Grid	Indices	Values	Bytes
2	1	3×3	2^9	2	2^9
2	1	4×4	2^{16}	2^4	2^{16}
2	1	8×8	2^{64}	2^{36}	$2^{64} \cdot 5$
3	2	3×3	2^{18}	2^2	2^{18}
3	2	4×4	2^{32}	2^8	2^{32}
3	2	5×5	2^{50}	2^{18}	$2^{50} \cdot 3$
5	3	3×3	2^{27}	2^3	2^{27}
5	3	3×4	2^{36}	2^6	2^{36}
5	3	4×4	2^{48}	2^{12}	2^{49}
5	3	5×5	2^{192}	2^{27}	2^{194}

Table 4: Lookup table array sizes

Lookup tables did in fact enable speeding up generation processing. However, JavaScript prevents threads from sharing data in order to avoid race conditions. So lookup tables became problematic. This should be a solvable issue, but there are better speedups to be had through other approaches.

6.1.4 Future Algorithms

Hashlife's concept of hashing subgrids can possibly help if those subgrids are kept small. The 563 terabytes of RAM requirement aren't necessary for 4×4 subgrids anymore with hash tables. The effectiveness of this approach depends on seeing repeated patterns. Otherwise the overhead of hashing could easily make this approach slower. There is little doubt that significant gains can be had in quad-community evolution with hashing by pre-hashing the most common patterns seen. This could be interesting for future research.

Priority queues can maintain top n lists, such as elite seeds for fitness evaluation, top 4 seeds for animating, and top 1 seeds in each generation for the external fitness metric described in section 4.1.

Finally, many games enter static states or very short loops well before the time limit is up. These can be stopped early for some multiple of speedup.

6.1.5 GPU

GPU support in software and hardware is particularly alluring as CUDA cores exceed 10,000 and memory is up to 24 GBytes on modern GPUs [21]. Unfortunately, WebGPU, an API allowing JavaScript to

compute with GPUs, is still in draft stages and only supported on beta versions of browsers behind a feature flag [22].

WebGL is typically used for 3d graphics in browsers. This is a stable API and a way to access the power of a GPU indirectly for our purposes. Unfortunately, there were issues around the necessity to use the WebGL API within a Web Worker (threads in JavaScript) [2].

6.1.6 Machine Learning and Stochasticity

Today's ML is essentially making predictions on patterns observed in data. Dramatic speedups can be had if we applied ML to the patterns of sections of the grid. This comes at the risk of making mistakes. Are mistakes so bad? This is just another form of mutation. In fact, this variation might allow us to simplify some of the genetic operators as mutation now comes from the rules of life. Although attempts at this have been somewhat unsuccessful [3, 6, 14], recent work [4] has shown great promise.

Along the same lines as ML, a stochastic process in GoL rules could have a big impact on performance, eliminating some of the need for mutations elsewhere. For example, some cells can be skipped for processing with some probability if previous calculations satisfied some heuristic.

6.2 Parallelization

The biggest performance boost attained, and what ultimately allowed us to best Golly in evolution through Turney's genetic operators, was through parallelization. Greater gains yet can be achieved through parallelization with programming languages like Rust and C/C++.

JavaScript uses Web Workers to run concurrent threads. These Web Workers can't share data. This makes parallel programming easier at the expense of significant communication between threads. Portions of the seed population are constantly being transferred and, unfortunately, this is slow. Greater care could potentially speed this up, but ultimately this points to the inevitable need to use a low-level programming language if we want to extend evolution exponentially beyond 100 generations.

A back-end server that receives initial parameters and seed populations from JavaScript front-end clients can be quite capable. Instead of having 16 threads from a powerful 8-core desktop (2 threads per core), Amazon has an x1.32xlarge EC2 instance with 128 fast vCPUs (and 2 terabytes of RAM for those gigantic lookup tables)! This is available immediately to rent for $4/hour (as a spot instance). Amazon's p4d.24xlarge EC2 instance has 8 NVIDIA A100 GPUs and 96 vCPUs. Each A100 GPU contains 6,912 CUDA cores.

A low-level programming language optimizes parallelization, often produces native code that is significantly faster than JavaScript, allows the clever bit manipulation and register tricks necessary for many fast algorithms, and has direct access to GPUs.

7 Results

There were several goals in this project:

- Implement platform for evaluating evolution by natural selection with quad-communities
- Improve performance to go well beyond 100 generations
- Attain better fitness as evaluated by an adaptation of Turney's unbounded external fitness measure f_n
- Create UI to visualize competitions among top seeds as they evolve in the background

The platform has been successfully implemented. Countless hours have been spent playing and experimenting with it. Performance has certainly been improved on a quad-core MacBook (8 threads). However, it still took 103 hours to complete 100 generations of 200 seeds. This is considerably better than Turney's model in Golly, which took just over a week to complete.
This is a great result in light of the fact that quad-communities have, on average, twice the height of an Immigration Game competition. Recall that Turney's rule for calculating the number of time steps in a competition before declaring a winner is (grid width+ grid height) $\times$ time factor. This means the number of time steps increases by a factor of $\frac{4}{3}$ with quad-communities. Figure 7 shows how each model progressed. It makes sense why both models slow down over time at pretty regular rates. As seeds grow through mutation and fusion, grids grow and time steps increase.

Unfortunately, there wasn't enough time to restart and go beyond 100 generations. Nevertheless, it's clear this is more possible now, especially when considering the optimizations of sections 6.1 and 6.2 along with bigger hardware.

Another kind of performance, unbounded external fitness as measured by f_n, was also good. Quad-

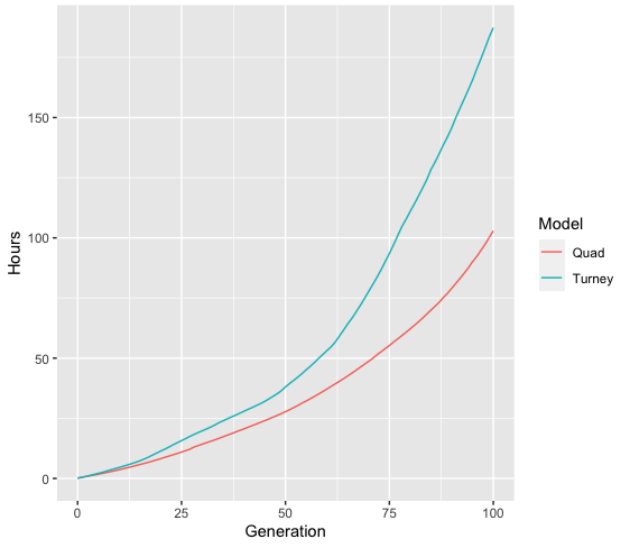

Figure 7: Cumulative hours to complete generations

community seeds performed close to Turney's layer 4 seeds. Figure 8 is Turney's chart with his unbounded external fitness score at each generation for his four layers. The quad-community model's performance is superimposed in purple. Performance of the quad-

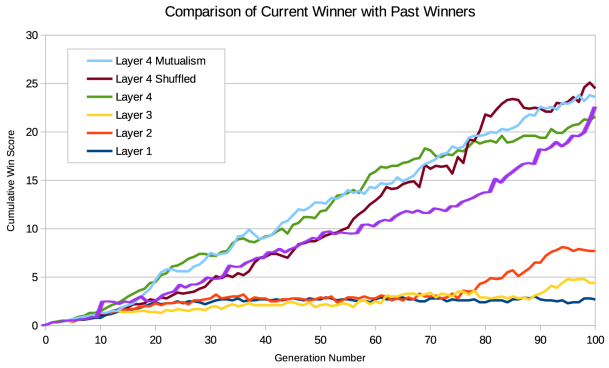

Figure 8: Turney's unbounded fitness scores with the Quad-Community model overlaid in purple

community approach suffered a little in the middle generations. However, it caught up in the end. Layer 4 was used for the quad-community generations.

Notice the slope is very high indicating the potential for quad-communities to far surpass Turney's Immigration Game approach. This may be analogous to humans taking significantly longer than incomparably less intelligent and less evolutionarily fit animals to do basic tasks, such as walking or eating. Seeds may need extra time to evolve to learn how to cooperate or how to deal with three-on-one instead of Immigration Game's one-on-one competitions. Once they do learn, these seeds become much more capable.

As a note, Turney has a bug in his code that may be limiting fitness slightly. It was noticed in log files that top seeds of later generations tended to be thin rectangles, with shapes such as 16×5. This was not seen in quad-community evolution. Occasionally a top seed will have double size in one dimension due to fusion, as in figure 9. However, triple and greater

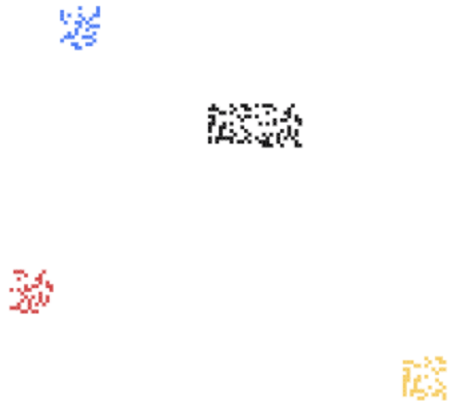

Figure 9: The black seed has gone through fusion

sizes in one dimension was never seen. The culprit appears to be in the following code:

```
if ((choice == 0) and
    (self.xspan > mparam.min_s_xspan)):
  # delete first row
```

This says the first row is to be deleted only if the seed's width is greater than the minimum width. This should've been that the first *column* is to be deleted. The same mistake was made for the other three `choice`s in his code. This issue allows for skinny seeds. It's an interesting question whether this bug aids or limits fitness. It would seem to be a problem for fitness, however, why did so many skinny seeds have the highest relative fitness scores?

There are two additional sources of variation for the quad-community unbounded fitness scores. First, with the limited time, 100 generations was only run once. Reruns for the first generations already show that there is considerable variation on early runs.

Second, there was stagnation in fitness in generations 10 to 20 and 64 to 74. There were also leaps in fitness in generations 10, 82, 90, and 98 to 100. When these occur seem fairly random. This seems to be a natural evolutionary process, leaps and stagnation. Subsequent runs could show quite a different picture

due to this. There are tactics to correct for this variation, but should it be corrected?

The UI is complete, notwithstanding a graphic designer's help. It is now easy to see how top seeds behave. Interesting structures have been observed:

- Nomadic clusters trudge through the grid
- Seeds can reduce quite a bit in number before exploding up, presumably to avoid predators
- Factories form to shoot gliders

Cooperation, collaboration, partnerships, and alliances haven't been observed. Though it's hard to know exactly what to look for. What are the basic building blocks of society at this level? Figure 10 showcases stable shapes observed in the wild that have mixed seeds.

(a) Split loaf (b) Plus oscillator (c) Eraser

(d) Gold ring (e) Half circle

Figure 10: Cooperating seeds forming stable unions

Finally, it's interesting to note that top seeds of small populations start to look similar as winning patterns emerge. Even with mechanisms in place to limit exact copies, top seeds stop changing much after a while. Winners remain dominant. This suggests local minima are easy to get trapped into with small populations. This has not been seen with population sizes of 200 and above.

8 Future

An obvious next step for future work is a simple rewrite of code in Rust to take advantage of massive Amazon servers at $4/hour. Parallelization at that level should be able to blast through generations.

Hashlife was discussed in section 6.1.2 as a possible ticket to unbounded evolution. Considerable work needs to go into adapting the algorithm or adapting Turney's genetic operators and competition mechanisms. This work may be well worth the effort. One approach to making Hashlife work in this context is to parallelize the algorithm. The gauntlet was thrown down by Tomas Rokicki when he wrote, "So far, hashlife has been resistant to parallelization. A big challenge for the community is to write an effective, efficient, parallel hashlife" [15].

Adapting the competition structure, genetic operators, and even relative fitness scores might have other evolutionary benefits as well. It was decided to stay as close to Turney's mechanisms as possible in order for comparisons to make sense. Future research, of course, can drop this requirement.

As an example of a subtle change, Turney's symbiosis as persistent mutualism checks if a fused seed is more fit than its parts. If it is more fit, then the fused seed enters the population and the least fit seed gets killed. However, Turney *first* removes the least fit seed, *then* evaluates fitness scores with the fused seed versus the rest of the population. The issue is that the least fit seed might no longer be least fit with the introduction of the fused seed. This is like tech geeks. In high school, they may have been the least fit. In a different population, like Silicon Valley, they could be most fit.

Ashes are what's left over after a GoL game has run its course. They are stable shapes, shapes that don't move or oscillate between different states. Ashes have been studied for decades [7, 12] and continue to be studied today [18]. Ashes are colorful in quad-communities and tell a more interesting story. An example is shown in figure 11.

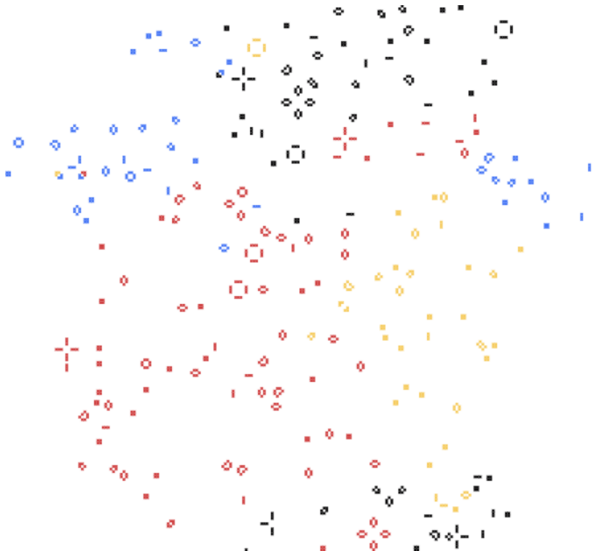

Figure 11: Ashes of a competition long forgotten

How far can a 2-dimensional GoL-like environment evolve seeds? Is open-ended evolution possible? 2D isn't life as we know it. 3-dimensional GoL has been researched and holds interesting possibilities for future research in the context of this paper. Figure 12 shows a beautiful example of a 3d earth-like CA world using rules similar to GoL [8]. The benefits to

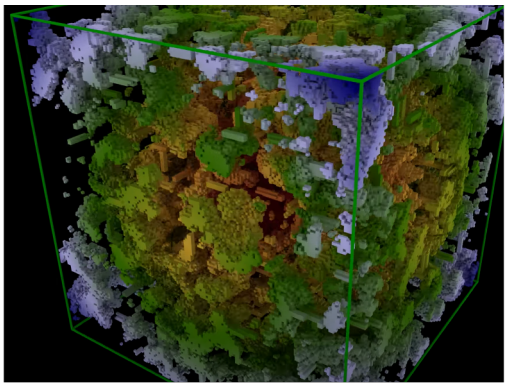

Figure 12: A finite 3d CA by Softology

going 3d are appealing beyond the beautiful visuals. Instead of 2×2 competitions in a quad-community, there would be $2 \times 2 \times 2$ oct-communities with more opportunity to develop real collaboration. This might seem computationally expensive. For example, 5×5 seeds have 2^{25} variations, whereas $5 \times 5 \times 5$ seeds have 2^{125} variations. Lookup tables are certainly out of the question. However, with three dimensions, maybe seeds and grids can be very small. A $3 \times 3 \times 3$ seed has 2^{27} variations, this is comparable to the familiar 5×5 seed.

9 Conclusion

The big question is how far evolution can go. The global minimum may not exist and life as we know it could emerge.

Faster algorithms and computational resources are necessary to see this through. This project has shown that there is potential. Both unbounded evolution and systems to advance far into the future of these virtual worlds are possible. The groundwork has been laid with a quad-community platform. This platform can accelerate performance immediately through many strategies discussed in this paper. Visualizations now allow researchers and enthusiasts to qualitatively analyze evolution by natural selection.

Variation in fitness scores may be a problem that needs correcting, as mentioned in section 7. Or maybe jumps in fitness are necessary. Earth had the Cambrian explosion, a big leap in evolution. Way later, some extremely small mutations, compared to the mutations in Turney's genetic operators, caused humans to make the extraordinary transition from apelike ancestors.

References

[1] Tab Atkins-Bittner. *Seeded Pseudo-Random Numbers*. URL: https://tc39.es/proposal-seeded-random/.

[2] Nick Desaulniers. *WebGL Off the Main Thread*. Jan. 2016. URL: https://hacks.mozilla.org/2016/01/webgl-off-the-main-thread/.

[3] Ben Dickson. *Why neural networks struggle with the Game of Life*. Sept. 2020. URL: https://bdtechtalks.com/2020/09/16/deep-learning-game-of-life/.

[4] William Gilpin. "Cellular automata as convolutional neural networks". In: *Physical Review E* 100 (Sept. 2019). DOI: 10.1103/PhysRevE.100.032402.

[5] Bill Gosper. "Exploiting Regularities in Large Cellular Spaces". In: *Physica D* 10.1-2 (1984), pp. 75–80. DOI: doi:10.1016/0167-2789(84)90251-3.

[6] Tom Grek. *Evolving Game of Life: Neural Networks, Chaos, and Complexity*. Jan. 2020. URL: https://medium.com/@tomgrek/evolving-game-of-life-neural-networks-chaos-and-complexity-94b509bc7aa8.

[7] Margaret Hill, Susan Stepney, and Francis Wan. "Penrose Life: Ash and Oscillators". In: *Advances in Artificial Life* 3630 (2005), pp. 471–480. DOI: https://doi.org/10.1007/11553090_48.

[8] Jason. *3D Cellular Automata*. Dec. 2019. URL: https://softologyblog.wordpress.com/2019/12/28/3d-cellular-automata-3/.

[9] Nathaniel Johnston. *HashLife*. https://conwaylife.com/wiki/HashLife. 2021.

[10] S. Levy. *Artificial Life: The Quest for a New Creation*. Pantheon Books, 1992. ISBN: 9780679407744. URL: https://books.google.com/books?id=IKEZAQAAIAAJ.

[11] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[12] Nick D. L. Owens and S. Stepney. "Investigations of Game of Life Cellular Automata Rules on Penrose Tilings: Lifetime, Ash, and Oscillator Statistics". In: *J. Cell. Autom.* 5 (2010), pp. 207–225.

[13] Ed Jr. Pegg. *Social Golfer Problem*. https://mathworld.wolfram.com/SocialGolferProblem.html.

[14] Daniel Rapp. *Learning Game of Life with a Convolutional Neural Network*. Sept. 2015. URL: https://danielrapp.github.io/cnn-gol/.

[15] Tomas Rokicki. *Life Algorithms*. https://www.gathering4gardner.org/g4g13gift/math/RokickiTomas-GiftExchange-LifeAlgorithms-G4G13.pdf. 2018.

[16] *Rust and WebAssembly*. https://rustwasm.github.io/docs/book/. 2021.

[17] Peter D. Turney. *Conditions for Open-Ended Evolution in Immigration Games*. 2020. arXiv: 2004.02720 [cs.NE].

[18] Peter D. Turney. "Evolution of Autopoiesis and Multicellularity in the Game of Life". In: *Artificial Life* 27.1 (2021), pp. 26–43. DOI: https://doi.org/10.1162/artl_a_00334.

[19] Peter D. Turney. "Symbiosis Promotes Fitness Improvements in the Game of Life". In: *Artificial Life* 26.3 (2020), pp. 338–365. DOI: https://doi.org/10.1162/artl_a_00326.

[20] Eric W. Weisstein. *Steiner Triple System*. https://mathworld.wolfram.com/SteinerTripleSystem.html.

[21] Wikipedia contributors. *List of Nvidia graphics processing units*. https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=1028219150. [Online; accessed 17-June-2021]. 2021.

[22] Wikipedia contributors. *WebGPU*. https://en.wikipedia.org/w/index.php?title=WebGPU&oldid=967380943. [Online; accessed 17-June-2021]. 2020.

[23] Manish Yadav et al. "Modern Fisher–Yates Shuffling Based Random Interleaver Design for SCFDMA-IDMA Systems". In: *Wireless Personal Communications* 97 (2017), pp. 63–73. DOI: https://doi.org/10.1007/s11277-017-4492-9.