Genetic Programming and Evolvable Machines (2019) 20:93–125
https://doi.org/10.1007/s10710-018-9336-1

# Evolving continuous cellular automata for aesthetic objectives

**Jeff Heaton**[1]

Received: 6 January 2018 / Revised: 17 August 2018 / Published online: 27 August 2018
© The Author(s) 2018

**Abstract**
We present *MergeLife*, a genetic algorithm (GA) capable of evolving continuous cellular automata (CA) that generate full color dynamic animations according to aesthetic user specifications. A simple 16-byte update rule is introduced that is evolved through an objective function that requires only initial human aesthetic guidelines. This update rule provides a fixed-length genome that can be successfully optimized by a GA. Also introduced are several novel fitness measures that when given human selected aesthetic guidelines encourage the evolution of complex animations that often include spaceships, oscillators, still life, and other complex emergent behavior. The results of this research are several complex and long running update rules and the objective function parameters that produced them. Several update rules produced from this paper exhibit complex emergent behavior through patterns, such as spaceships, guns, oscillators, and Universal Turing Machines. Because the true animated behavior of these CA cannot be observed from static images, we also present an on-line JavaScript viewer that is capable of animating any *MergeLife* 16-byte update rule.

**Keywords** Cellular automata · Genetic algorithm · Generative art · Multi-objective optimization

## 1 Introduction

Cellular automata (CA) [51] are a class of computer model that are made up of a grid (or other organization) of cells that move between discrete states as dictated by an update rule [58]. The quintessential example of a CA is Conway's Game of Life (GOL) [13]. Though many CA, such as GOL, were introduced as recreational mathematics or generative art [3, 32], there are practical applications [2]. In 1996 a team performed computations with a genetic algorithm (GA) for global coordination [31].

✉ Jeff Heaton
jtheaton@wustl.edu

1 Washington University in St. Louis, St. Louis, MO 63130, USA

In artificial chemistry, scientists have implemented self-replicating cells through a CA [18]. GOL was shown to be Turing complete—capable of computing anything that can be computed with a Turing machine [5]. Likewise, Elementary Cellular Automaton (ECA) rule 110, which was studied extensively by Stephen Wolfram, was proven to be Turing complete [10, 57].

Continuous CA are a generalization of the more common discrete CA where the states are expressed as continuous numbers [23, 57, 58]. GOL has previously been adapted from discrete to continuous. One notable generalization of GOL is *SmoothLife* [37], where GOL is adapted to a continuous domain; however, the resulting animation is still monochrome. *MergeLife* is a continuous CA where each cell is a 24-bit RGB encoded color. These cells are arranged on a 2D rectangular grid of cells where each interior cell has 8 neighbors that influence how that cell is updated. This arrangement of 8 neighbor cells is referred to as a Moore Neighborhood, named in honor of Edward F. Moore, a pioneer of CA theory. The number of alive cells present in a CA neighborhood is referred to as its neighbor count.

Families of update rules are typically expressed in some sort of encoding. For example, an ECA is usually encoded as a decimal number that represents an 8-bit pattern that specifies a new cell value based on the 8 possible combinations of the three cells immediately above the target cell [57]. Wolfram MathWorld lists ECA update rules 30, 54, 60, 62, 90, 94, 102, 110, 122, 126, 150, 158, 182, 188, 190, 222, and 250 as the most frequently studied ECA rules [53]. The fact that many of these rules are no more than one or two binary digits away from each other is indicative of a smooth search space that increases the chances of a GA finding its way to locally optimal solutions.

Other examples of a smooth search space for CA exist. Researchers have generalized GOL to a family of update rules that specify the neighbor counts at which a cell should be born or survive. The update rule for GOL can be written as `B3/S23`, which means a cell is born with a neighbor count of 3, survives with a neighbor count of 2 or 3, and dies in all other cases. *Larger than Life* is a GOL-like family of CA that allow the size and type of neighborhood to be varied [11, 12]. *Larger Than Life* has even been scaled to neighborhood sizes of radius 25 and beyond [49]. For example, the CA *Bugs*, is written in the *Larger than Life* form of `R5,C0,M1, S34..58, B34..45,NM` [54]. This string specifies a neighbor radius (R) of 5, 0 history states (C), the middle cell (M) is included in the neighbor count, the neighbor count for survival (S) is between 34 and 58, the neighbor count for birth (B) is between 7 and 11, the neighborhood is an extended Moore (NM) neighborhood. An extended Moore neighborhood for radius 1 are the 8 cells around the center cell; a radius of 2 include the 24 cells that form two rectangles around the center cell. This encoding and generalization of GOL provides a smooth enough search space to be optimized by a GA. The smooth search space is evident because small changes to many of the components in the Bugs rule produce a CA that appears visually similar to the original Bugs CA.

Literature includes many examples of CA that can produce visually appealing animations as artifacts [11, 15, 37]. The artifact produced by MergeLife is the animated sequence, not a still computer image. Other evaluation methods also place emphasis on the process leading to the final resulting image [8]. The visual appeal of

this artifact must be evaluated over several CA generations to grasp the full intricacy of the animated patterns that these automata can produce. For example, GOL supports spaceships, which are patterns that move across the grid. Guns are even more complex structures that are capable of producing spaceships. Oscillators are patterns that repeat over a certain number of CA generations. None of these pattern types would be evident from a single CA generation. Multiple frames must be analyzed because one of the goals of *MergeLife* is to produce CA that exhibit such patterns. These aesthetic MergeLife moving patterns are not meant to be representative of any natural phenomenon.

Many techniques, such as *Picbreeder* [40] allow users to guide a GA to evolve Compositional Pattern-Producing Networks (CPPNs) [44] that create generative art. Many of these algorithms require the human to serve as the objective function and are sometimes referred to as Human Based GA (HBGA) [24] or Interactive Evolutionary Computation (IEC) [48]. This sort of iteration-level control has yielded greater artistic control for interactive evolution of images and animation [15]. By tweaking the *MergeLife* objective function, the user can define the aesthetic aspects of a new CA. These CA will produce generative art in the form of a sequence of still images that become an animation. We have implemented *MergeLife* in several programming languages, the source code can be downloaded from our GitHub repository.[1]

In this paper, the abbreviation *CA* refers both to singular form *cellular automaton* and plural *automata*. Each application of the update rule in a CA is called a generation. Some literature refers to the *generation* as a *step* or *tick*. GA evolve a population over many generations. Likewise, CA perform an update rule over many generations of cells. Because of this the term *generation* has two meanings in this paper. For clarity, these generations will be referred to as *CA generations* and *GA generations*. This paper contains scalars, vectors, matrices and rank 3 tensors. For clarity, this paper follows naming conventions generally applied to linear algebra. Scalar values are lowercase and italicized ($a$), vectors are lowercase and bold ($\boldsymbol{a}$), matrices are capital and italicized ($A$), and tensors are capital and bold ($\mathbf{A}$).

This paper begins by introducing the MergeLife CA update rule and comparing it to existing CA update rules in literature. The paper continues by explaining how each MergeLife rule is encoded into a hexadecimal string that becomes the genome representation that will be evolved by a GA. Continuing to build the pieces needed for a GA, the paper introduces the MergeLife objective function that is guided by initial human provided aesthetic guidelines, but alleviates the human from manually scoring each rule. With the CA update rule, genome encoding, and objective function in place the paper can now present the design of the GA that will actually evolve the update rules. Next, the results from several runs of the MergeLife GA are presented, along with the rules that were discovered. A discussion of how MergeLife is a generalization of GOL and therefore Turing complete is presented. The paper concludes with a review of previous work, conclusions and next steps.

[1] https://github.com/jeffheaton/mergelife.

## 2 The MergeLife CA

Like GOL, the *MergeLife* universe is a rectangular grid of cells arranged in a Moore Neighborhood. Unlike GOL, individual cells are not simply on or off. Each cell is a vector containing three integer values between 0 and 255 (inclusive) that represent the intensities of red, green, and blue. A value of 0 indicates that the RGB component is fully off, and a value of 255 means the color component is fully on. For example, white is encoded as [255, 255, 255] and black is encoded as [0, 0, 0]. Similarly, a blue color would be [0, 0, 255]. This is consistent with standard RGB encoding. The universe grid is always initialized to a completely random state, with all RGB components initialized to values uniformly sampled (with replacement) between 0 and 255. This is the state upon which the first CA generation occurs.

For a discrete CA, such as GOL, the neighbor count is simply the number of nearby cells that are in a live state. To perform a continuous neighbor count, *MergeLife* merges (or averages) the red, green, and blue components of each cell together—*MergeLife* is named for this merging characteristic of the algorithm. Though similar, this is not the same as a grayscale operation. Grayscale conversion algorithms usually convert an RGB color to grayscale by taking an unevenly weighted average of the RGB components. The *MergeLife* neighbor count is calculated by taking a regular average of each cell's RGB values. The maximum that any one value in the averaged matrix can hold is 255. This average is an integer value that is obtained by taking the floor of the sum of the three color components divided by 3.

At each CA generation, the matrix mode is taken of the merged grid. The matrix mode is the most commonly occurring RGB averaged value in the grid. Some CA will have large background regions of empty space that are occupied by a common color. The matrix mode of a grid is effectively the RGB average of the background color of the CA for a given CA generation. For some *MergeLife* update rules this background color will be constant and for others it will fluctuate. The current background color averaged value is the contribution to the neighbor count for any cell that does not have 8 neighbors. Cells that are in the interior of the grid will have 8 neighbors. However, cells at the grid edges will have fewer. Corner cells will have only 3 neighbors. All edge cells will be assumed to have 8 neighbors by treating any missing neighbors as having the background color. The neighbor count is calculated by summing the 8 neighbors in the averaged matrix. Because each neighbor can have a maximum value of 255, the maximum neighbor count is 2040 ($8 \times 255$). This is the only concept of background color that MergeLife has. Background colors may appear by virtue of the update rule; however, they are not specifically defined in any way.

### 2.1 Update rule

The *MergeLife* update rule is a generalization of *GOL* to a continuous RGB space. MergeLife and *GOL's* update rules share a similar structure. *GOL* counts

**Table 1** Decoded MergeLife update rule (`cb97-6a74-88c0-28aa-1b6a-834b-4fe8-60ac`)

| High ($\alpha$) | Range | Key-color | Pct. ($\beta$) (%) | Index ($\gamma$) | Octet-1 | Octet-2 |
|---|---|---|---|---|---|---|
| 216 | 0–215 | Blue | 83 | 5 (blue) | `1b (27)` | `6a (106)` |
| 320 | 216–319 | Blue | −67 | 4 (yellow) | `28 (40)` | `aa (-86)` |
| 632 | 320–631 | White | −18 | 7 (cyan) | `4f (79)` | `e8 (-24)` |
| 768 | 632–767 | Black | −65 | 8 (white) | `60 (96)` | `ac (-84)` |
| 848 | 768–847 | Red | 91 | 2 (red) | `6a (106)` | `74 (116)` |
| 1048 | 848–1047 | Purple | 59 | 6 (purple) | `83 (131)` | `4b (75)` |
| 1088 | 1048–1087 | Yellow | −50 | 3 (green) | `88 (136)` | `c0 (-64)` |
| 1624 | 1088–1623 | Red | −82 | 1 (black) | `cb (203)` | `97 (-105)` |

the number of alive neighbors to select between four sub-rules that each specify if each cell should change its state to either alive or dead. Similarly, MergeLife sums the RGB values of the 8 neighbors to select between up to 8 sub-rules that each specify the amount to change the current cell's color to one of 8 key-colors that correspond to each sub-rule. Each MergeLife sub-rule is defined by two values—the neighbor count range that the rule covers and the percent that the current cell should move toward the rule's key-color if the rule applies. At most one MergeLife sub-rule will apply per CA step. It is possible for no sub-rule to apply, in which case the current cell does not change its value.

*MergeLife* update rules are defined by hyphen-delineated hexadecimal strings, such as `cb97-6a74-88c0-28aa-1b6a-834b-4fe8-60ac`. This string is composed of 8 hyphen-delineated octet-pairs that define 8 sub-rules. Table 1 shows these sub-rules decoded from a hexadecimal update rule string. These 8 sub-rules define the state transitions for the *MergeLife* grid and hold all of the information necessary for a *MergeLife* update rule. For a different *MergeLife* rule, the values and ordering will be different; however, the column structure will remain the same. The exact means by which a hexadecimal string is decoded to a sub-rule table is described in the next section. The columns $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\gamma}$ are provided to the *MergeLife* update rule as vectors. The remaining columns are provided for clarity and are not used by the update rule algorithm. All columns are decoded directly from the hexadecimal rule string that is given in the *octet-1* and *octet-2* columns.

Each row is a sub-rule that has a range that specifies what neighbor counts that rule applies to. The sub-rules are sorted by the *high* ($\boldsymbol{\alpha}$) column; however, the original hexadecimal string order is given by the *index* ($\boldsymbol{\gamma}$) column. Each sub-rule row is linked to a key-color, based on this position in the hexadecimal update rule string. For example, the first row in this table comes from index 5 of the hexadecimal string. As a result of this it uses the corresponding index 5 number from the key-color given by Table 2. While the grid allows continuous states for cells, these cells are constantly transitioning towards these 8 key-colors. Despite the fact that the key-colors are eight discrete values, the grid rarely converges to just a few discrete colors at any given CA generation. Rather, many of the *MergeLife*

**Table 2** MergeLife key-colors

| Octet index | Key-color | RGB red | RGB green | RGB blue |
|---|---|---|---|---|
| 1 | Black | 0 | 0 | 0 |
| 2 | Red | 255 | 0 | 0 |
| 3 | Green | 0 | 255 | 0 |
| 4 | Yellow | 255 | 255 | 0 |
| 5 | Blue | 0 | 0 | 255 |
| 6 | Purple | 255 | 0 | 255 |
| 7 | Cyan | 0 | 255 | 255 |
| 8 | White | 255 | 255 | 255 |

update rules given in this paper often have grids that are observed to have many thousands of discrete RGB color values.

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 255 & 0 & 0 \\ 0 & 255 & 0 \\ 255 & 255 & 0 \\ 0 & 0 & 255 \\ 255 & 0 & 255 \\ 0 & 255 & 255 \\ 255 & 255 & 255 \end{bmatrix} \tag{1}$$

The pseudocode presented for *MergeLife* refers to the above equation ($K$) for the RGB values of the key-colors. This matrix contains the same information as what Table 2 conveys. The rows of the matrix are the 8 key-colors and the columns are the RGB components of red, green, and blue for each color. The RGB color model was chosen because it is well known and requires no transformation for the video displays that this algorithm will most likely be viewed upon. It would be possible to encode the grid into other color models, such as hue, saturation, lightness (HSL), hue, saturation, value (HSV), or cyan, yellow, magenta, black (CYMK).

Additionally, the pseudocode for the update rule requires two constant vectors to define the Moore neighborhood, which are the 8 cells immediately surrounding the cell to be updated. The Moore neighborhood includes the four cells that are above, below, to the left, and to the right of the current cell. Additionally, a Moore neighborhood also includes the four cells that are the diagonals away from the current cell. The following two vectors ($\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$) contain the needed transitions to visit every Moore neighbor from the current cell.

$$\hat{\boldsymbol{x}} = \begin{bmatrix} 0 & 0 & -1 & 1 & -1 & 1 & 1 & -1 \end{bmatrix} \tag{2}$$

$$\hat{\boldsymbol{y}} = \begin{bmatrix} -1 & 1 & 0 & 0 & -1 & 1 & -1 & 1 \end{bmatrix} \tag{3}$$

The vectors $\hat{x}$ and $\hat{y}$ define the directions that are needed to move from the cell being updated to visit each of the 8 neighbors. The vector $\hat{x}$ specifies column movement and the vector $\hat{y}$ specifies row movement. Taken together, these two 8-value vectors specify to visit the following directions: N, S, W, E, NW, SE, NE, and SW. The key-color matrix ($K$) and neighborhood vectors allow the *MergeLife* update rule defined in Algorithm 1 to lookup the key-color for a particular sub-rule.

**Algorithm 1** MergeLife update rule

**Precondition:** $\mathbf{G}$ is a rank 3 tensor of size $h \times w \times 3$ containing the current CA grid, $\boldsymbol{\alpha}$ is a length-8 vector containing the upper limits of the rule ranges, $\boldsymbol{\beta}$ is a length-8 vector containing the update rule percents as decimal proportions, $\boldsymbol{\gamma}$ is the key color index, based on the position of the sub-rule in the hexadecimal string. $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, & $\boldsymbol{\gamma}$ are all sorted by $\boldsymbol{\alpha}$.

1: **function** UPDATERULE($\mathbf{G}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \boldsymbol{\gamma}$)
2: $M \leftarrow 0_{h \times w}$ ▷ Let $M$ hold the RGB-merged matrix (like $G$, but no RGB depth.
3: $\mathbf{G}' \leftarrow 0_{h \times w \times 3}$ ▷ Let $\mathbf{G}'$ hold the new grid, with RGB depth.
▷ Calculate average RGB for each cell.
4: **for** $y \leftarrow 1$ TO $h$ **do**
5: **for** $x \leftarrow 1$ TO $w$ **do**
6: $M_{y,x} \leftarrow \lfloor \frac{1}{3} \sum_{k=1}^{3} \mathbf{G}_{y,x,k} \rfloor$
7: $b \leftarrow$ `mode`$(M)$ ▷ The background color ($b$) is the most common merged color.
▷ Calculate neighbor counts and determine new grid ($\mathbf{G}'$)
8: **for** $y \leftarrow 1$ TO $h$ **do**
9: **for** $x \leftarrow 1$ TO $w$ **do**
10: $c \leftarrow$ `NeighborCount`$(M, b, x, y, h, w)$
11: **for** $d \leftarrow 1$ TO $8$ **do** ▷ Loop over all 8 rules/key-colors
12: **if** $c < \boldsymbol{\alpha}_d$ **then**
13: $d' \leftarrow \boldsymbol{\gamma}_d$ ▷ Find key color index.
14: **if** $\boldsymbol{\beta}_d < 0$ **then**
15: $d' \leftarrow (d' \mod 8) + 1$ ▷ Add 1 to $d'$, but wrap 9 to 1.
16: $\boldsymbol{\Delta g} \leftarrow \lfloor (K_{d',\ldots} - \mathbf{G}_{y,x,\ldots}) |\boldsymbol{\beta}_{d'}| \rfloor$
17: $\mathbf{G}'_{y,x,\ldots} \leftarrow \mathbf{G}_{y,x,\ldots} + \boldsymbol{\Delta g}$
18: **break**
19: **return** $\mathbf{G}'$

The update rule accepts the current grid ($\mathbf{G}$) and returns an updated grid ($\mathbf{G}'$). These are both rank 3 tensors of the shape $h \times w \times 3$. The depth of 3 accounts for the red, green, and blue components of each cell's RGB color. The sub-rules specified in this table are processed in order, one at a time. Changes to the grid are not visible to the update rule until the processing of the following CA generation. This prevents these changes from affecting the subsequent update rule changes in the same CA generation. This separation is a common requirement for most CA, including GOL. As each cell is processed, the update algorithm must calculate a neighbor count which is defined by Algorithm 2.

**Algorithm 2** Count Mergelife neighbors

**Precondition:** $M$ is the merged matrix ($h \times w$), $b$ is the most frequent value from $M$ (the background color),x & y are the column/row requested, $h$ & $w$ are the height and width of $M$.

```
1: function CountNeighbors(M, b, x, y, h, w)
2:     s ← 0
3:     for i ← 1 TO 9 do
4:         x' ← x + x̂_i                          ▷ Move x' and y' over all neighbors.
5:         y' ← y + ŷ_i
6:         if x' < 1 ∨ y' < 1 ∨ x' > w ∨ y' > h then
7:             s ← s + b        ▷ If off the edge of the grid, use background color (b).
8:         else
9:             s ← s + M_{y',x'}                 ▷ Add the merged/averaged RGB color.
10:    return s
```

The applicable sub-rule is the row in Table 1 whose range includes the neighbor count. There can be no gaps or overlaps between these ranges. Therefore, at most one sub-rule row can apply to a given cell for a given CA generation. Sometimes no sub-rule will apply. If the neighbor count is beyond the range of the highest sub-rule, then no change is made to that cell. For example, Table 1 has a gap beyond 1624. Any cell that has a neighbor count that is greater than or equal to 1624, will not be affected by the update rule. Each sub-rule contains a *target color* and *percent*. These colors are the key-colors defined by the *MergeLife* algorithm. Key-colors are constant and are not defined by the update rule or evolved by the GA. The key-colors are listed in Table 2.

The appropriate key-color index ($d'$) must be determined for the current sub-rule ($d$) with the following equation:

$$d' = \begin{cases} (\boldsymbol{\gamma}_d \mod 8) + 1 & \text{if } \boldsymbol{\beta}_d < 0 \\ \boldsymbol{\gamma}_d & \text{otherwise} \end{cases} \tag{4}$$

Each sub-rule will correspond to a particular key-color ($K$). What key-color a sub-rule corresponds to is dependent on if the $\beta_d$ for that sub-rule is positive or negative. If $\beta_d$ is greater than or equal to zero, the key-color index is $\gamma_d$. If $\beta_d$ is less than zero, the key-color is the next index after $\gamma_d$. If moving to the next index moves beyond the last key-color, then the first key-color is chosen. The color value of each grid cell will move towards the appropriate key-color by the percentage amount each CA generation, according to the following equation:

$$\boldsymbol{\Delta g} = \lfloor (K_{d',\ldots} - \mathbf{G}_{y,x,\ldots}) |\boldsymbol{\beta}_d| \rfloor \tag{5}$$

The value $\mathbf{G}_{y,x}$ contains the current RGB value, which is a 3-value vector at $x$ and $y$. The key-color ($K_{d'}, \ldots$) for the current sub-rule is also a 3-value vector. The sub-rule row's percent ($\boldsymbol{\beta}_d$) is a proportion that governs how much of the distance between the current color and key-color should be covered in each CA generation. Because the percent is a decimal value it is necessary to take the floor of the product to obtain an integer RGB vector.

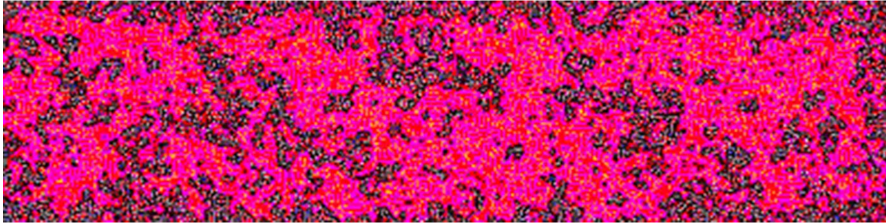

**Fig. 1** CA with a complex substrate (`cb97-6a74-88c0-28aa-1b6a-834b-4fe8-60ac`)

## 2.2 Decoding a MergeLife update rule

*MergeLife* update rules are encoded into hexadecimal strings made up of 8 pairs of octets that occupy a total of 128 bits. These strings serve two purposes. The first is that this notation allows users to easily represent and exchange *MergeLife* update rules. The second is that this hexadecimal string becomes the genome that the GA applies crossover and mutation to.

This paper represents all *MergeLife* rules in this notation. The following merge life update rule hexadecimal string illustrates this format and will serve as a decoding example for this section:

```
cb97-6a74-88c0-28aa-1b6a-834b-4fe8-60ac
```

We created a JavaScript web application that allows the animated viewing of any *MergeLife* rule.[2] The JavaScript application allows the user to observe the universe grid over the passage of CA generations. The rule provided earlier in this section will produce a pattern that resembles Fig. 1, if viewed in our JavaScript application. The JavaScript application also produces a decoded table, similar to Table 1, for any *MergeLife* rule. Because the figures in this paper cannot properly show the animated transformations between CA generations, the reader is encouraged to view the *MergeLife* rules provided with this paper with the JavaScript viewer.

*MergeLife* decodes the hexadecimal string given earlier in this section into a form similar to Table 1. This provides Algorithm 1 with the vector-parameters of $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\gamma}$ that are needed for execution. These parameters correspond directly with the columns of the same name in Table 1. The remaining columns are provided to illustrate the decoding process of the update rule.

Each sub-rule row in the table is represented by one of the 4-digit hexadecimal numbers in the string. The first octet pair in the string is `cb97`, which contains the octets `cb` (unsigned decimal 203) and `97` (two's complement decimal $-105$). The first octet is an unsigned number and the second octet is a signed two's complement number. The *high* ($\alpha$) is $203 \times 8 = 1624$ and the merge *percent* ($\beta$) is $105 \div 128 \approx 0.82$.

[2] http://www.heatonresearch.com/mergelife/.

Despite the fact that `cb97` is the first number in the hyphenated hexadecimal string, it is sorted to the final position of the decoded update rule. This position is because the rows are ordered by the $\boldsymbol{\alpha}$ column, which specifies the high end of the neighbor count for each sub-rule. The fact that this row occurred first in the original hexadecimal string is represented by the 1 in the *index* ($\boldsymbol{\gamma}$) column.

For convenience, Table 1 shows the neighbor count range covered by each sub-rule, which is the previous row's $\alpha$ value up to, but not including, the current row's $\alpha$ value. The sub-rule with the lowest $\alpha$ starts its range with zero. If a sub-rule's range is a single number (e.g. 0–0), then that sub-rule row is inactive.

Decoding *octet-2* ($o_2$) is more complex than *octet-1* ($o_1$), because *octet-2* is a two's complement signed number. This is evident from the fact that *octet-1* in Table 1 contains only positive decimal values, while *octet-2* contains a mix of positive and negative decimal values. The *percent* can be decoded from *octet-2* by the following equation as a decimal proportion:

$$p = \begin{cases} o_2/127 & \text{if } o_2 > 0 \\ |o_2|/128 & \text{otherwise} \end{cases} \tag{6}$$

The sign of *octet-2* is simply a flag that allows two sub-rules to be mapped to the same key-color. A positive percent sub-rule is mapped to the key-color that corresponds with the sub-rule's *index* ($\boldsymbol{\gamma}$) within the hexadecimal string. A negative *percent* sub-rule is mapped to the next index's key-color. A zero *percent* sub-rule will not affect the current cell. The next key-color for the final row of white (index 8) will be wrapped around to black (index 1). This equation obtains the *percent* ($p$) by dividing the value of *octet-2* by the magnitude of either the maximum value of a signed or unsigned byte. The final *percent* is always a positive number. The ability of a negative *percent* to change the key-color is a needed complexity to allow multiple ranges to apply to a single key-color. This is needed to emulate GOL and will be discussed in further detail later in this paper. The column named *Key-Color* specifies the key-color that each sub-rule is ultimately mapped to after the sign of the original percent is handled.

## 3 Objective function design

Creating an objective function that evaluates the aesthetic ranking of MergeLife rules presented two difficult problems. The first problem is that the objective function must execute the MergeLife update rules over a potentially large number of update steps to see what patterns these rules produce. This requires considerable computational expense during the running of the MergeLife GA.

The second problem is that the evaluation of aesthetic qualities by a computer requires a level of creativity that is difficult for a machine to reproduce. To provide a consistent vocabulary for this Computational Creativity Theory (CCT) frameworks such as FACE, IDEA, and SPECS have been proposed [9, 22]. These frameworks provide a model to describe the creative component of a computer program. These models also examine what it means to be creative [21] and explore the spectrum from minimal

human guidance to situations where the human essentially becomes the objective function. MergeLife is only an extension of the human's own creativity. The objective function statistics covered in the next section simply automate and alleviate the human user from needing to manually score a large number of genomes produced by crossover and mutation.

Literature contains a number of different aesthetic measures for the evaluation of computer generated art. Heijer and Eiben [17, 19] propose four aesthetic measures that can be applied to static images: Machado and Cardoso (image complexity), Ross and Ralph (bell curve), fractal dimension, and combined weighted sum. MergeLife evaluates some fitness through static image analysis. However, the bulk of MergeLife's fitness assessment occurs through the analysis of the image transition between CA steps.

Wolfram, in *A New Kind of Science*, and several papers dating from the mid-1980s [28, 55, 56], defined four classes into which CA can be divided according to their behavior [57]. The Wolfram class number increases along with the complexity of the patterns produced by an update rule. Wolfram's rules are defined as follows [57]:

- Class 1: Nearly all initial patterns evolve quickly into a stable, homogeneous state. Any randomness in the initial pattern disappears.
- Class 2: Nearly all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may filter out, but some remains. Local changes to the initial pattern tend to remain local.
- Class 3: Nearly all initial patterns evolve in a pseudo-random or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread indefinitely.
- Class 4: Nearly all initial patterns evolve into structures that interact in complex and interesting ways, with the formation of local structures that are able to survive for long periods of time. Class 2 type stable or oscillating structures may be the eventual outcome, but the number of CA generations required to reach this state may be large, even when the initial pattern is relatively simple. Local changes to the initial pattern may spread indefinitely.

Wolfram conjected that many, if not all, class 4 CA are capable of universal computation. This has been proven for GOL, which is a class 4 CA [10]. These classes are subjective, making it difficult to determine an update rule's Wolfram class programmatically. The objective function chosen for *MergeLife* attempts to give higher class update rule's similarly high scores. However, these scores must be continuous, rather than a discrete number of classes. As is common with evolutionary algorithms, the higher scored update rules will have a greater chance of contributing components of themselves to future CA generations of update rules.

### 3.1 Configurable multi-objective function

MergeLife does not attempt to be creative in and of itself. Rather, a configurable objective function gives the user the ability to define objectives that might result in an aesthetic CA. For the purposes of this paper, we considered an aesthetic CA

**Table 3** MergeLife objective function for this paper

| Stat | Min | Max | Weight | Weight min | Weight max |
|---|---|---|---|---|---|
| Generations | 300 | 1000 | 1 | − 1 | 1 |
| Foreground | 0.001 | 0.1 | 1 | − 0.1 | − 1 |
| Active | 0.001 | 0.1 | 1 | − 1 | − 1 |
| Rect | 0.02 | 0.25 | 2 | − 2 | 2 |
| Mage | 5 | 10 | 0 | − 5 | 0 |

to be one that exhibits complex emergent behavior through patterns, such as spaceships, guns, and oscillators. A different human user would likely have different criteria. However, spaceships, guns, and oscillators are some of the patterns that have spawned the greatest interest in GOL. The process of creating such an objective function involved much trial and error as we developed new statistical measures and introduced them as multiple objectives. By evaluating the CA that achieved high scores and also visually displayed complex emergent behavior, we were able to tune the objective function to produce visually pleasing CA's with the desired behavior. The fact that the higher scored MergeLife rules display this sort of behavior speaks to the ability of these objective statistics to score the aesthetic appeal of MergeLife rules.

The provided objective function for *MergeLife* can be adjusted to fit each user's own aesthetic tastes. This is done by adjusting the importance weighting of a number of statistics from the grid as the updates are evaluated. These statistics form a multi-objective function that provides the score that allows the *MergeLife* GA to find the most desirable update rules for crossover and mutation. These multiple objectives are created by a user and then stored in a table such as Table 3 that is executed by Algorithm 3. The values contained in this table are the parameters that guided the evolution of all *MergeLife* update rules discovered for this paper. These values can be tweaked to individual preferences.

The *MergeLife* objective function begins by initializing the CA grid to random values and running the update rule until the grid pattern stabilizes. Once this grid has stabilized the statistics needed for the multiple objectives can be collected. The number of update steps that occurred prior to stabilization is one of the objective statistics. The means by which stabilization is defined is covered in the next section where the *update steps* statistic, as well as each of the other objective statistics are defined. It is not necessary to use all of the provided statistics—the update rule used in this paper did not use every statistic. The objective function given by Table 3 did not make use of either the *colors* or *background* statistics. Though unused in this paper, both of these statistics can be used for other aesthetic guidelines.

Algorithm 3 loops over each statistic in Table 3. These statistics are individually defined in the next section. If a given statistic is below the *Min* column value the *Weight Min* value will be added to the score. Similarly, if a given statistic is above the *Max* column value the *Weight Max* value will be added to the score. If a given statistic is within the range of *Min* and *Max* then a value based on *Weight* will be added that is based on the following equations:

$$y = 0.5(x_{max} - x_{min}) \tag{7}$$

$$\Delta s = t_{weight}\frac{0.5y - |\hat{y} - y|}{0.5y} \tag{8}$$

The value $y$ is the ideal value for the statistic, which is the midpoint of the range of acceptable values. The value $\hat{y}$ is the actual value of the given statistic. The value $t_{weight}$ specifies the weight of this rule from the table. The resulting $\Delta s$ is added to the overall score ($s$).

**Algorithm 3** MergeLife Objective Function

**Precondition:** The vector $\boldsymbol{o}$ contains the object function specification (such as Table 3) and $\mathbf{G}$ holds stats on the objective statistics from the grid.

```
1: function ObjectiveFunction(o, G)
2:     s ← 0
3:     for t IN o do                         ▷ Loop over each of the objectives and sum into s.
4:         x ← MLStat(t, G)
5:         y ← 0.5(x_max − x_min)
6:         if x < t_min then                                                    ▷ Too small.
7:             Δs ← t_weightMin
8:         else if x > t_max then                                               ▷ Too large.
9:             Δs ← t_weightMax
10:        else                                                             ▷ Within range.
11:            Δs ← t_weight (y−|x−y|)/y
12:        s ← s + Δs
13:    return s
```

The objective function is stochastic because the initial state of the grid is random. Because of this randomness there will be some variance in the score obtained over multiple calls to the objective function. To mitigate the randomness of these scores the objective function is executed over 5 cycles. The max score from these 5 runs becomes the score for the current update rule being evaluated. We tried a number of different cycle counts and ultimately decided on 5. With this number of cycles the standard deviation between 10 runs of the objective function score of the same update rule was typically less than 0.5. This provided consistent enough scores for rules to evolve that sufficiently met the criteria specified in Table 3. While additional cycles provide more consistent results, this causes a linear increase in the computation time requirements for each objective function evaluation.

### 3.2 Choosing objective parameters

The *MergeLife* implementations provided with this paper encode objective rule parameters as JavaScript Object Notation (JSON). Table 3 could be represented as the following JSON:

```
{
    "config": {
        ...
    },
    "objective": [
        { "stat": "steps", "min": 300, "max": 1000,
          "weight": 1, "min_weight": −1, "max_weight": 1
        },
        { "stat": "foreground", "min": 0.001,
          "max": 0.1, "weight": 1, "min_weight": −0.1,
          "max_weight": −1
        },
        { "stat": "active", "min": 0.001, "max": 0.1,
          "weight": 1, "min_weight": −1,
           "max_weight": −1
        },
        {
            "stat": "rect", "min": 0.02, "max": 0.25,
             "weight": 2, "min_weight": −2,
             "max_weight": 2
        },
        {
            "stat": "mage", "min": 5, "max": 10,
            "weight": 0, "min_weight": −5,
            "max_weight": 0
        }
    ]
}
```

The overall goal of the parameters that we chose was to create long running CA that had objects moving over distinct background regions. Additionally, these CA should run for upwards of 1000 CA generations before stabilizing. The chosen objective function parameters encourage the generation of update rules that will remain active for at least 300–1000 CA generations. The amount of stable foreground structure should be small, between 0.1 and 10%. Similarly, the amount of active cells should be small, between 0.1 and 10%. There should be enough space that a rectangle of the background color should be able to take up at least 25% of the grid. Additionally, the background color (mode of the averaged matrix) should not have changed for 5 CA generations. The exact definition of these measures is given in the next section. The format of the *config* section will be defined later in this paper.

To arrive at these settings was an objective iterative process. Initially we generated many hundreds of random update rules and quickly examined the single frames at 300 CA generations. Most converged to a single background color or chaotic behavior. We selected favorable patterns and began to divide the update

rules into five categories (from most favorable to least, bolded text indicates the category name):

– **Good** pattern with potential activity for spaceships and other patterns (similar to Wolfram Class 4).
– Simple **sustained** pattern (similar to Wolfram Class 3).
– **Expansion** to s single pattern (similar to Wolfram Class 3).
– **Shrink**ing to single color (similar to Wolfram Class 1 or 2).
– Quick convergence to a **dead** grid with no patterns/single background color (Similar to Wolfram Class 1).
– Any patterns quickly descend into **chaos** (similar to Wolfram Class 3).

From an initial random sample of 100 rules 55% were chaos, 37% were dead, 6% were shrinking, and 2% were expansion. None of the 100 random rules were good or sustained. An optimization algorithm beyond random sampling is needed to obtain better CA rules. Obviously, sufficiently large random samples would produce occasional good rules. These classes were used for observation only and are too discrete to be effectively used for an objective function [30].

The objective function parameters were tuned so that the patterns that we visually determined to be more desirable had higher scores—less desirable patterns would receive lower scores. Once the objective parameters were tuned for the random patterns, we began using a GA (described later in this paper) to evolve more visually appealing patterns. We kept a set of patterns to constantly tune the objective parameters against. Table 4 is the set of MergeLife rules that we discovered in our research, along with their scores. Though it is not perfect, the scores for the more desirable patterns (such as spaceships and oscillators) are considerably higher than those of the less desirable patterns (such as quick convergence to a blank grid).

The hexadecimal notation used to encode MergeLife rules into genomes provides a smooth enough search space that the GA is able to maximize the objective function and produce CA patterns that often include spaceships, oscillators, still life, and other complex emergent behavior. These patterns can be observed through the visual inspection of the CA produced by MergeLife.

Further analysis of these rules demonstrate two additional characteristics of the MergeLife search space and objective function. The first characteristic is the amount of noise in the MergeLife objective function. MergeLife scores are the maximum of 5 evaluation cycles run from randomly initialized grids. While MergeLife itself is deterministic, these randomly initialized grids are not. Rerunning the rules in Table 4 produced scores with a standard deviation of 0.5 over a sample of 10 calculations of the score using 5 cycles. More cycles will produce more consistent results for a given MergeLife rule at the cost of objective function runtime. While higher cycle counts lowered the standard deviations they linearly increased the computation time. Therefore, we found that the 5 cycles provided an acceptably consistent balance between consistent scoring and computation time.

The second characteristic is the smoothness of the MergeLife search space for the provided objective function. This smoothness is evident from the fact that small

**Table 4** Evaluated MergeLife update rules

| Hex rule | Category | Score |
|---|---|---|
| a07f-c000-0000-0000-0000-0000-ff80-807f | Good | 2.6979885814668423 |
| 7b58-f7b4-c5b4-fd87-22fa-eb10-6de8-107c | Good | 1.8599999999999999 |
| d44f-9ba5-53e7-7373-fb07-9984-b444-727b | Good | 3.0164163372859023 |
| eaa3-fb3c-4d19-e732-7376-9601-764b-e810 | Good | 3.0774703557312257 |
| E542-5F79-9341-F31E-6C6B-7F08-8773-7068 | Good | 3.792929292929293 |
| 0de6-3496-8507-7cc7-34c6-d5a9-bcfd-2355 | Good | 3.9313131313131313 |
| 8503-5eb6-084c-04df-7657-a5b3-6044-3524 | Good | 2.3944664031620553 |
| 658e-bef0-6ada-c5b7-6ad2-2984-383d-2f0e | Good | 3.6494949494949496 |
| 6769-5dd6-7d03-564e-a5ec-cae2-54c4-810c | Good | 3.625841018884497 |
| df1d-bba1-8e06-aa66-48ff-7414-6a2f-6237 | Good | 4.545226174791392 |
| d8a7-8915-629b-2248-6055-fc87-ef92-15f5 | Good | 3.9434343434343435 |
| E542-5Fc0-2041-F31E-6C6B-7F08-8773-7068 | Good | 3.4676767676767675 |
| b774-93f4-ac95-f5b3-86f0-dc6e-e397-37de | Good | 3.7339833113746157 |
| 309d-92a3-e31b-3b63-5e24-c9d2-2dcc-8168 | Good | 4.224242424242425 |
| 8b05-3be0-e9aa-996a-ee61-2529-646b-2fff | Sustained | 3.7298902064119455 |
| 6007-7d42-05e5-1b9b-2899-e043-1cd4-2f7b | Sustained | 3.9954325867369347 |
| fd3d-8ef7-d4a4-3a7a-70ab-e7cb-03ba-5760 | Sustained | 2.3237944664031622 |
| 4ec6-2584-fc1f-0456-ceee-152c-3992-6fee | Expansion | 2.7715590689503733 |
| 1e6b-9afa-16f2-23bc-3396-9fb3-46f4-cb4f | Expansion | – 3.9757575757575756 |
| 7c6a-1fa3-74aa-6d87-eb8c-d12b-a4ef-dda5 | Expansion | – 2.889566472175168 |
| f7f7-bda5-e41c-f965-7a72-4aa1-bafd-f4c9 | Expansion | 1.377830478700044 |
| 6d1d-e41a-841e-e983-b6ea-53f5-9eff-7c1a | Shrink | 2.1669565217391304 |
| 19c9-7565-68f0-82c0-abab-474a-2a7d-eab1 | Shrink | 1.7330158730158731 |
| 7223-85f8-3c9a-7058-6b7a-92ee-3efc-d1f7 | Shrink | 1.5742857142857143 |
| 3907-5c95-b561-44e9-b4b0-4fc3-adf4-8895 | Shrink | 1.9 |
| de0c-1960-1030-ab0a-8105-38d6-1c6b-f85f | Shrink | – 3.9676767676767675 |
| d028-66d8-5563-5958-5824-35ba-f604-0d2d | Shrink | 1.76 |
| 6d1d-e41a-841e-e983-b6ea-53f5-9eff-7c1a | Shrink | 2.0990250329380764 |
| 7107-794d-41f7-8ef8-0948-f95f-a8be-00f9 | Shrink | 1.800210803689065 |
| 03c3-62b4-458a-0d6e-6d2d-1d1d-7afc-ec9b | Dead | 1.7685714285714287 |
| 3e69-82bb-ee4a-f54e-841e-acb7-6664-f541 | Dead | 1.949459815546772 |
| 4af3-9216-c799-4501-4e58-50a6-ddca-ae26 | Dead | – 0.5436024844720497 |
| 6b46-687f-4885-4988-0a56-efc7-c494-9bfb | Dead | 1.0514285714285714 |
| 6bfb-5015-58ab-8161-8d70-c743-a8ba-34d3 | Dead | – 0.5126745718050066 |
| 7cd2-a302-364a-47ad-fe92-1eab-0d8e-d47b | Dead | 1.917986071899115 |
| 9cd5-588b-9532-4af0-7816-af1f-23a1-6683 | Dead | 1.9545893719806764 |
| 31d0-16fd-626d-68ed-9dfe-bea9-1997-7d51 | Dead | 2.099159294811469 |
| 745b-d779-6d22-a650-32d7-c272-9cab-6e66 | Dead | 0.7418633540372671 |
| 31d0-16fd-626d-68ed-9dfe-bea9-1997-7d51 | Dead | 2.389821193299454 |
| 79e4-d466-6d3f-1cb6-83ff-df8c-c73c-4154 | Dead | 0.043478260869565 19 |
| 3392-112d-6f45-73bd-075b-5223-25a6-e28b | Dead | 0.7969609134826525 |
| 620c-0efc-3d88-7cf2-14d8-d960-58ab-9b78 | Dead | – 0.10000000000000009 |

**Table 4** (continued)

| Hex rule | Category | Score |
|---|---|---|
| 56ac-691f-d2ba-46cd-80db-379b-a7aa-0c17 | Dead | 0.022222222222222143 |
| 4c26-7f82-a6bb-d3fa-a932-d39b-a5b1-0e8a | Chaos | −4.1 |
| 9b55-7855-ba02-cccc-1c9e-2007-08bb-389c | Chaos | 0.786824769433465 |
| 17b5-8b30-5e0d-ac4c-ee86-c95a-6fd6-af19 | Chaos | −3.2 |
| 46e5-9fa2-1718-bd61-1ccc-d7b9-9d34-3c9e | Chaos | −3.9474747474747476 |

changes to each of the 16 hexadecimal digits usually produce correspondingly small changes in the score for that rule. To test this we reran all 47 rules in Table 4 and increased/decreased each digit. Because there are 32 digits each rule has a maximum possibility of 64 movements if the digit is allowed to both increase and decrease by one. Because digits 0 and F cannot move down or up respectively some of the rules had less than 64 perturbations. The change in score was greater than 1.0 in only 31% of these 64 possible changes over all 47 rules. This shows that the score is relatively smooth over small movement within the search space.

The search space is also composed of many local minima. This is not surprising, given that there are many different MergeLife CA that can exhibit spaceships, oscillators, still life, and other complex emergent behavior. We considered it desirable to find many interesting MergeLife update rules rather than attempting to find one global maximum for the objective function. The results of this research are over 1100 MergeLife rules that are all local minima.

## 4 Objective function statistics

The *MergeLife* objective function has multiple objectives. Recent research of multiple objectives for CA suggests techniques for both static analysis of individual CA generations as well as changes to the grid over a series of CA generations [33, 35]. These techniques provided some direction for the statistics collected for the *MergeLife* objective function. This section describes the means by which each of these objectives is calculated. Many of these calculations are based on the average (or merged) value of the RGB colors. The averaged matrix, which was calculated by the update rule, is a part of many of the objective function statistics.

The mode of the average matrix is the most common number in that matrix, and is effectively the background color of the grid. Statistics related to the background color of the grid can be important indicators for the objective function. A matrix is kept that counts how many CA generations individual cells have been the background color. A cell is considered a stable background cell if it has been the background color for longer than 100 CA generations.

### 4.1 Generation count

The CA generation count (*generations* from Table 3) is the number of CA generations that have occurred since the CA started from a random initial grid. Each CA generation is one application of the update rule. CA generations will continue until the grid converges, which is defined as:

- Less than 1% of the merged grid cells have had different values for 100 CA generations.
- The count of stable background cells has not changed for 100 CA generations.
- More than 1000 total CA generations.

If any of the above three conditions are met, the CA will stop and its objective function will return a score. The CA generation count is the number of CA generations that occurred prior to these conditions being met. The generation count statistic is useful when you would like to ensure that high scoring genomes run for an acceptably long number of generations.

### 4.2 Background ratio

The background ratio (named *background*, but not used in Table 3) is the ratio between the number of background cells and the total number of cells in the grid. A background cell is defined to be any cell that has a current merged color value that is equal to the matrix mode of the merged color matrix. Some CA, such as GOL, have relatively large background areas. This statistic allows CA with large background areas to be rewarded or penalized, as desired.

### 4.3 Foreground ratio

The foreground ratio (*foreground* from Table 3) is the ratio between the number of foreground cells and the total number of cells in the grid. A foreground cell is defined to be a cell that has had the same non-background merged color for more than 5 CA generations. A feature of many CA are cell groups that are stable until they are disturbed by other nearby patterns. These stable static patterns are usually called "still life" [2]. A related pattern is an oscillator where a rectangle of cells will repeat a pattern within a certain period. Unless an infinite sized grid is provided, GOL will almost always converge to a grid containing only "still life" and oscillators. Figure 2 shows GOL after enough CA generations to evolve it to only oscillators and "still life". This statistic allows the user to define the amount of still life that is acceptable in a MergeLife CA. Depending on user preference, still life can be either desirable or undesirable.

**Fig. 2** GOL with only still life (`a07f-c000-0000-0000-0000-0000-ff80-807f`)

### 4.4 Active cell ratio

The active cell ratio (*active* from Table 3) is a ratio between the number of active cells and the total number of cells in the grid. An active cell is defined as a cell that has not been a background cell for the last 5 CA generations, but was a background cell in the last 25 CA generations. Active cells are those that are often background, but will typically have other patterns, such as spaceships, that travel through them.

### 4.5 Largest rectangle ratio

The largest background rectangle ratio (*rect* from Table 3) is the ratio of the largest rectangle of background cells and the total number of cells in the grid. This statistic indicates that not only is there a large amount of background cells, but there are also large unobstructed rectangles of background cells. The *rect* statistic is similar to the *background* statistic, except that *rect* specifies the largest contiguous area of background cells and *background* tracks how much of the grid is made up of background cells. If spaceships are desired, the *rect* and *active* statistics are the most important two statistics. The *active* statistic detects the spaceships and *rect* detects that they have space to fly through. If spaceships are present in a CA with a low *rect* they will rapidly crash into something and often cease to exist, similar to what Wolfram describes for Class 3 CA.

### 4.6 Mode age

The current mode age (*mage* from Table 3) is the number of CA generations that have occurred since the matrix mode has changed. Usually it is desired for the merged color matrix mode to remain constant. Because the mode becomes the neighbor value for edge cells, changing the mode can have a large scale effect around the outer perimeter of the grid. Additionally, some *MergeLife* update rules will produce patterns where a large amount of cells rapidly switch between two different colors. This can cause a potentially undesirable flashing effect, such as update rule `a7f6-1d56-9574-9645-c318-5c36-feca-fa0e`.

### 4.7 Color count

The color count (named *color*, but not used in Table 3) is the number of distinct RGB colors currently present on the grid. This allows the colorfulness of an update rule to be scored.

## 5 GA design

The MergeLife GA described in this paper is based on crossover and mutation that follow the algorithms described by Mitchell [30]. Genomes are defined to be *MergeLife* update rules. These update rules are stored as arrays of 16-byte numbers that come directly from the hexadecimal *MergeLife* rule format described earlier in this paper. These genomes are each assigned a score by the objective function described earlier in this paper. Superior and inferior genomes are chosen by tournament selection after each GA generation. Superior genomes are chosen to produce offspring through crossover and mutation. Inferior genomes are chosen to be killed, making room for the offspring produced by the superior genomes in the next GA generation.

### 5.1 Objective function

The objective function is somewhat stochastic, as the phenotype is the ultimate converged grid that was produced by *MergeLife* update rule genome on a random starting grid. Though multiple random starting grids can produce a variety of different converged patterns, most patterns produced from a common *MergeLife* update rule will appear closely related. The objective function goes through several evaluation cycles, with different starting grids, and takes the maximum score. A cycle count of five provides relatively stable scores.

### 5.2 Crossover of two MergeLife update rules

The crossover technique chosen for this paper takes two parents and produces two offspring genomes that are based entirely on the parent genomes. This is done by selecting two cut points that divide each of the parents into three splices, giving six splices total. The cut points are the same for each of the two parents. A first child is created that contains the outer two splices from the first parent and the middle splice from the second parent. Similarly, a second child is created that contains the outer two splices from the second parent and the inner splice from the first parent. This is illustrated by the following listing:

```
Parent 1: 0de6-3496-8507-7cc7-34c6-d5a9-bcfd-2355
```

```
Parent 2: 8503-5eb6-084c-04df-7657-a5b3-6044-3524


Off Spring 1: 0de6-3496-8507-7cc7-34c6-d5a9-6044-2355
Off Spring 2: 8503-5eb6-084c-04df-7657-a5b3-bcfd-3524
```

For these two update rules, the two parents produce two offspring that are somewhat similar to the original parents. The fact that this small mutation produces a very similar CA to the parent speaks to the smoothness of the search space provided by the MergeLife rule encoding.

### 5.3 Mutation of a MergeLife update rule

The mutation technique chosen for this paper takes a single parent and produces a new child that contains a random shuffle of the single parent. To perform this shuffle, two random half-bytes (4-bits) are chosen from the *MergeLife* update rule. This corresponds to two digits in the *MergeLife* hexadecimal rule (dashes are not considered). This is illustrated by the following listing:

```
Parent 1: d8ab-8915-6297-2248-6055-fc87-ef92-15f5
Off Spring 1: d8a7-8915-629b-2248-6055-fc87-ef92-15f5
```

The original parent is a chaotic CA. However, by flipping the fourth digit with the twelvth we now have a similar, but more organized CA. The new CA has a much larger background region and spaceships can be seen as it runs.

### 5.4 GA configuration

The configuration settings for the MergeLife CA are set using the JavaScript Object Notation (JSON) file.

```
{
    "config": {
        "rows": 100,
        "cols": 100,
        "populationSize": 100,
        "crossover": 0.75,
        "tournamentCycles": 5,
        "zoom": 5,
        "renderSteps": 250,
        "evalCycles": 5,
        "patience": 1000,
        "scoreThreshold": 3.5,
        "maxRuns": 1000000
    },
    "objective": [
        ...
    ]
}
```


The above settings were used for all experiments performed in this paper. The CA grid was $100 \times 100$. The GA had a population size of 100. Crossover produced 75% of new genomes and mutation the remaining 25%. Tournament selection with 5 rounds selected the parents from the population. For any renders produced, the pixels will be zoomed by 5 and 250 CA steps will occur for the render. The objective function will use 5 cycles to mitigate the effects of a random starting grid. When the GA is ran, 1000 steps can go by without improvement before the CA is assumed to have converged. The score threshold of 3.5 means to keep any CA that recieves and objective function score higher than 3.5. The final section defines the objective function. The format of the *objective* section was defined earlier in this paper.

## 6 Results and MergeLife rules discovered

We discovered over 1100 different MergeLife rules while working on this paper. Some of these rules are provided in this paper, the remainder are provided at the author's GitHub repository. These rules were discovered mainly from the Java implementation of MergeLife. Currently, the Java implementation has better performance than the Python and JavaScript versions. The Java version makes use of multithreading and scales well to multicore systems. To produce the results given in this paper we ran the MergeLife GA on an Amazon Web Services (AWS) Elastic Cloud 2 (EC2) instance of type `c5.18xlarge`. This instance type is compute optimized and contains 72 cores and 144 GB of RAM. During the run, all 72 cores were utilized at nearly 99%; however, the memory was only utilized at 11%. This instance type was chosen for its large number of CPU cores. Memory is much less important. All of the higher CPU count AWS instances have more than enough RAM for MergeLife. This instance type was able to evaluate around 3000 genomes a minute.

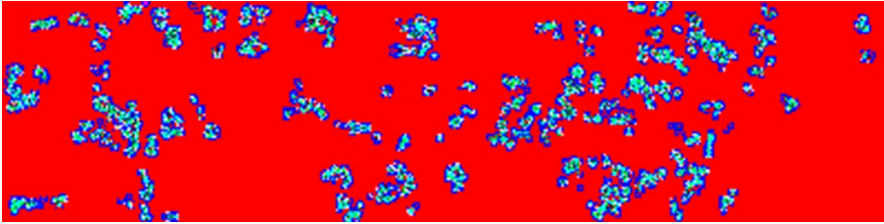

**Fig. 3** Red world update rule (`e542-5f79-9341-f31e-6c6b-7f08-8773-7068`) (Color figure online)

We performed two runs on this type of EC2 instance. The first run was for 24 h. A population was allowed to continue until no new top genome had been produced for 1000 evaluations. At this point the population is considered to have converged. After convergence the top genome was recorded if its score was above a configurable threshold of 3.5. During these 24 h a total of 2702 convergences occurred. From these a total of 146 rules successfully exceeded the threshold of 3.5. Therefore the success rate is around 5.4%. Further optimization of the GA hyperparameters might increase this percentage. For this run the hyperparameters were as follows: 100 population size, crossover percent of 75%, tournament cycles of 5, and the objective function score was the max of 5 evaluation cycles. These hyperparameters were chosen though by informal experimentation and could certainly be tuned further. The majority of the time spent in preparation of this paper was the development of the objective function and the creation of a smooth encoding for the MergeLife CA.

The second AWS run was much longer. We used 5 concurrent EC2 instances of type `c5.18xlarge` for 33 h with the goal of finding 1000 rules that exceeded the objective score threshold of 3.5. This resulted in a total of 18,796 convergences that had occurred before 1000 rules were discovered. The results of the first run were representative in that the much larger second run achieved a similar 5.3% success rate. While conducting both of these two runs we encountered a variety of interesting *MergeLife* update rules that are stored at GitHub.[3]

In the following sections, we will highlight five of these rules. The selection of rules to add here is largely subjective. We chose these update rules for their creation of spaceships, interesting still life patterns, and ability to span a large number of CA generations before converging. We assigned a name to each of these update rules, based on their overall appearance. The images of each of these update rules were produced from a random grid and applying the update rule for 1000 CA generations.

### 6.1 Red world

*Red World* is our favorite update rule discovered by this algorithm so far. While running this update rule on random initial grids we have observed: spaceships,

[3] https://github.com/jeffheaton/mergelife-experiments.

**Fig. 4** Yellow world update rule (`7b58-f7b4-c5b4-fd87-22fa-eb10-6de8-107c`) (Color figure online)

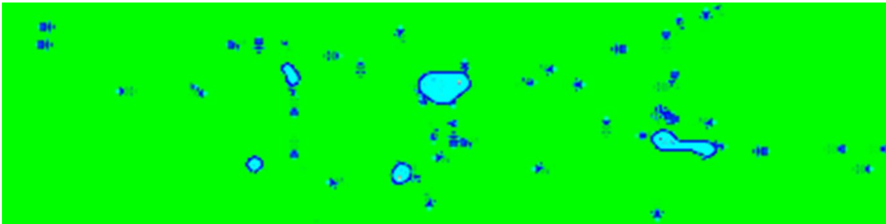

**Fig. 5** Cells with spaceships update rule (`8503-5eb6-084c-04df-7657-a5b3-6044-3524`)

oscillators, and still life. Spaceships are frequently spontaneously produced by this update rule. Additionally, even on the finite grids, this update rule can take considerable time to converge. Though this update rule can consistently and spontaneously produce spaceships, we have not yet observed a distinct isolated gun, such as the *Gosper Gun* [2] in GOL. A grid produced by the *Red World* update rule is shown in Fig. 3 and is represented by the following *MergeLife* update rule:

```
e542-5f79-9341-f31e-6c6b-7f08-8773-7068
```

Some of the *Red World* spaceships move across the screen in their initial shape with only small internal change and leave nothing behind. Other spaceships are somewhat gun-like in that they leave behind exhaust trails that spawn additional patterns without destroying their source spaceship.

### 6.2 Yellow world

*Yellow World* is similar to *Red World* in that it can spontaneously produce spaceships, still life, and oscillator patterns. However, there are considerably fewer background patterns than *Red World*. This was accomplished by evolving *Red World* and placing emphasis on larger background rectangles (the *rect* statistic). This produced an update rule that is similar to *Red World*, except that there is considerably less background noise. This causes the spaceships to be more apparent, yet causes this update rule to generally run for considerably fewer CA generations

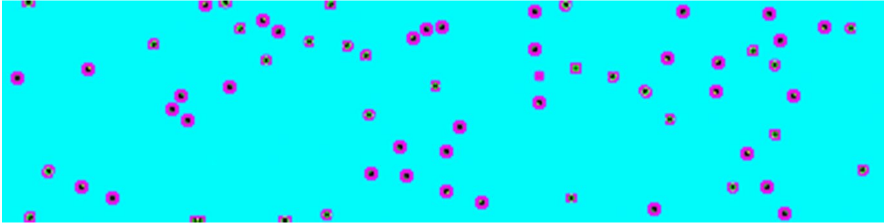

**Fig. 6** Still life and oscillators update rule (`6769-5dd6-7d03-564e-a5ec-cae2-54c4-810c`)

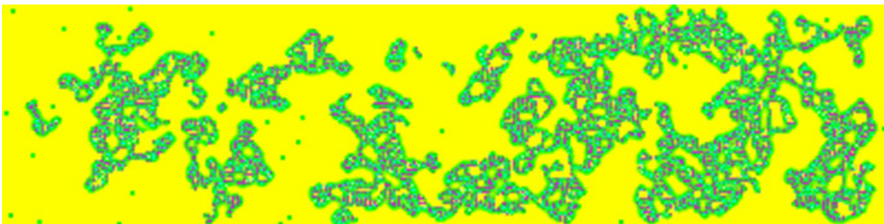

**Fig. 7** Sustained cellular update rule (`df1d-bba1-8e06-aa66-48ff-7414-6a2f-6237`)

before converging. A grid produced by the *Yellow World* update rule is shown in Fig. 4 and is represented by the following *MergeLife* update rule:

```
7b58-f7b4-c5b4-fd87-22fa-eb10-6de8-107c
```

The *Yellow World* spaceships act similarly to their *Red World* counterparts in that some leave exhaust trails and others do not. Like *Red World*, the *Yellow World* exhaust trails are gun-like in that they will often produce additional spaceships.

### 6.3 Shrinking cells with spaceships

The *Shrinking Cells with Spaceships* update rule produces cellular structures that emit several different types of small spaceship. These cellular guns do not move, but rather slowly contract and eventually disappear. Several types of spaceship are produced by the cells in this update rule. The spaceships produced by gun-cells will impact other cells, giving the illusion that the cells are shooting at each other. The spaceships simply dissipate once they impact the cell they are heading towards—no damage is inflicted. As the cells shrink they decrease the number of sites producing spaceships and eventually the cell disappears all together. A grid produced by the *Shrinking Cells with Spaceships* update rule is shown in Fig. 5 and is represented by the following *MergeLife* update rule:

```
8503-5eb6-084c-04df-7657-a5b3-6044-3524
```

**Table 5** GOL truth table

| Result | Living neighbors | Dead neighbors | Neighbor count |
|---|---|---|---|
| White | 0 | 8 | 2040 |
| White | 1 | 7 | 1785 |
| No change | 2 | 6 | 1530 |
| Black | 3 | 5 | 1275 |
| White | 4 | 4 | 1020 |
| White | 5 | 3 | 765 |
| White | 6 | 2 | 510 |
| White | 7 | 1 | 255 |
| White | 8 | 0 | 0 |

### 6.4 Still life and oscillators

The *Still Life and Oscillators* update rule quickly stabilizes into still life and oscillators. In this regard it is similar to GOL, except that this update rule usually converges much quicker than GOL for a given random initial grid. Unlike the previous update rules, it does not appear to produce spaceships. The *Still Life and Oscillators* update rule is shown in Fig. 6 and is represented by the following *MergeLife* update rule:

```
6769-5dd6-7d03-564e-a5ec-cae2-54c4-810c
```

### 6.5 Sustained cellular

The *Sustained Cellular* update rule produces several blob-like areas that will often last for 10,000–15,000 CA generations before disappearing. During the CA generations, the borders of the blob structures will change, giving the appearance of slow movement. Still life structures are also produced by this update rule. A grid created by the *Sustained Cellular* update rule is shown in Fig. 7 and is represented by the following *MergeLife* update rule:

```
df1d-bba1-8e06-aa66-48ff-7414-6a2f-6237
```

## 7 Turing completeness and the relationship to Conway's game of life

CA are capable of complex computation and emergent behavior. Turing completeness was proven for both ECA rule 110 [10] and GOL [38]. This section demonstrates that *MergeLife* is a generalization of GOL by showing that GOL can be simulated by the following *MergeLife* update rule:

```
a07f-c000-0000-0000-0000-0000-ff80-807f
```

Therefore, it is possible for the *MergeLife* GA to discover GOL. By extension, *MergeLife* can represent CA that are capable of functioning as Universal Turing Machines. Because MergeLife CA can be Turing Machines, they also exhibit the halting problem, which states that whether a Turing Machine will halt or not cannot be determined by analysis of the program code. Because of this the MergeLife update rules must generally be executed to see how long they will take to converge or halt to a stable grid. This has important ramifications to the design of the objective function that was discussed previously in this paper.

This section explores how a *MergeLife* rule can be constructed that emulates the four rules that specify GOL state transition [13]:

- Any live cell with fewer than two live neighbors dies, as if caused by underpopulation.
- Any live cell with two or three live neighbors lives on to the next CA generation.
- Any live cell with more than three live neighbors dies, as if by overpopulation.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

A living neighbor will be considered to be black ([0, 0, 0]) and a dead neighbor is white ([255, 255, 255]). Four *MergeLife* sub-rules can be created that perform the above four GOL rules. Before representing GOL as a *MergeLife* update rule, it is first useful to consider GOL as Table 5 that specifies the result of all 9 possible neighbor counts between 0 and 8.

The above table has four distinct result bands. The cell becomes white for 0 to 1 living neighbors, no change for 2, black for 3, and white again for 4 to 8 living neighbors. There are three distinct outcomes for every cell on the grid: change the cell to black, change the cell to white, or leave the cell at its current value. By its definition, GOL contains only white or black cells—dead or living. Therefore the *percent* column for every sub-rule that sets a cell to black/white will use 100%. The sub-rule that does not affect the current state of the cell (GOL survived) will use a *percent* value of 0% to simply leave the existing value alone. In addition to these 3 sub-rules a fourth is needed. As can be seen from Table 5 there are two overlapping bands where a white value must be explicitly set: 0 or 1 neighbors and 4 or more neighbors. Because of this, it is necessary to assign the cyan sub-rule a negative percent to cause it to target the next key-color—white. This is why it was necessary to have the ability to map multiple sub-rules to the same key-color to achieve GOL with white and black pixels.

The GOL update rule makes use of the white, black, red and cyan key-color sub-rules. The white and black sub-rules specify when a cell should be moved to either white or black. The cyan sub-rule handles the additional white band. The red sub-rule has a 0 *percent* and therefore specifies that a cell should survive (not change). The algorithm does not use the other sub-rules and as a result they have *percent* and *range* values set to zero. Transitions in both directions between black and white are absolute, therefore the only absolute values for percent are 100% or 0%.

**Table 6** Decoded GOL update rule (`a07f-c000-0000-0000-0000-0000-ff80-807f`)

| High ($\alpha$) | Range | Key-color | Pct.($\beta$) (%) | Index ($\gamma$) | Octet-1 | Octet-2 |
|---|---|---|---|---|---|---|
| 0 | NA | Green | 0 | 3 (green) | `00 (0)` | `00 (0)` |
| 0 | NA | Yellow | 0 | 4 (yellow) | `00 (0)` | `00 (0)` |
| 0 | NA | Blue | 0 | 5 (blue) | `00 (0)` | `00 (0)` |
| 0 | NA | Purple | 0 | 6 (purple) | `00 (0)` | `00 (0)` |
| 1024 | 0–1023 | White | 100 | 8 (white) | `80 (128)` | `7f (127)` |
| 1280 | 1024–1279 | Black | 100 | 1 (black) | `a0 (160)` | `7f (127)` |
| 1536 | 1280–1535 | Red | 0 | 2 (red) | `c0 (192)` | `00 (0)` |
| 2048 | 1536–2047 | White | – 100 | 7 (cyan) | `ff (255)` | `80 (-128)` |

The ranges specify the number of neighbors present for each transformation. *MergeLife* will take the average RGB colors for each cell to compare against each row's range. Each neighbor that is white will be 255, because white is RGB [255, 255, 255]. Likewise each neighbor that is black will be 0 [0, 0, 0]. This requires looking at neighbor counts backwards, because the neighbor count will be of the dead/white neighbors, not the living/black neighbors. A cell that had no living neighbors has a *MergeLife* neighbor count of 2040. Similarly, a cell that had all living neighbors has a neighbor count of 0, as indicated by Table 5.

The white sub-rule range of 0–1023 specifies between 0–4 dead neighbors. This is equivalent to 4 or more living neighbors. Such a condition will be set to white. This is equivalent to GOL's overpopulation transition.

The black row range of 1024–1279 specifies 5 dead neighbors. This is equivalent to 3 living neighbors. Such a condition will be set to black. This is equivalent to GOL's birth transition.

The red row range of 1280–1535 specifies 6 dead neighbors. This is equivalent to 2 living neighbors. Such a condition will be left as it is. This is equivalent to GOL's survival transition.

Finally, the cyan row's range of 1536–2047 specifies 7 or more dead neighbors. This is equivalent to 1 or fewer living neighbors. Such a condition will be set to white, due to the cyan sub-rule's negative percent. This is equivalent to GOL's underpopulation transition.

The sub-rules necessary to simulate GOL are given in Table 6.

## 8 Related work

The unique contribution of this paper is the automatic production of generative art through the use of a CA evolved from an objective function that can be customized to what the individual user considers visually appealing. It is also important to note that the generative art is considered to be the animation produced by a running CA, not simply a single image frame. The human-user provides initial guidance to the objective function; however, the algorithm's performance is not

hindered by needing human interaction while running. Related work in the area of generative art is considerable; however, the algorithm presented in this paper builds upon the related work described in this section.

Several projects employ human-operators as the primary component of the objective function for evolved generative art. An early example of human-operator guided evolution of CA rules, presented by Sims [43], used a GA to evolve CA update rules. The method employed by Sims differs from MergeLife in that it is monochrome and requires the human-user to evaluate the CA at every GA step. *Picbreeder* [40, 41] is a website that evolves still images using CPPNs and an objective function that is guided by Internet users. A related project, *Endless Forms* [7] allows Internet users to direct CPPNs to evolve objects to be created by a 3D printer. Both of these techniques require human interaction for each GP generation. The Capow [39] project evolves continuous valued CA from wave equations to produce a variety of visually interesting still images.

The generation of CA update rules is a natural application to evolutionary algorithms [31]. Evolutionary programming has been applied to the CA [6, 25]. Morphogenesis is a problem that has CA rule evolution applications [6, 34, 43]. While *Picbreeder* and *Endless Forms* made use of CPPNs to actually produce generative art, it is also possible to use CPPNs as the update rule of a CA. Nichele, Ose, Risi, and Tufte [34] employed a `NeuroEvolution of Augmenting Topologies` (NEAT) [46] GA to evolve CA update rules as a solution to morphogenesis and replication. Because a neural network is the update rule for such a CA their update rule becomes complex. One of the goals for *MergeLife* was to allow the update rule to be represented in a short string that is executed by a relatively simple program.

Biological plants are a frequent source of inspiration for artificial life based artistic algorithms. Lindenmayer Systems (L-Systems) [26, 36] describe the behavior of plant cells and model the growth processes of plant development. L-System based algorithms have been adapted beyond their 2D origins to 3D [4]. These artificial plants are similar to MergeLife in the fact that the animation of their growth and progression is often considered either all or part of the resulting creative artifact.

A number of projects evolved update rules to cause the CA to produce output that closely matches specific training data. Evolutionary algorithms evolved rules [59] for the standard formulation of the density classification problem. Research has also shown that evolutionary algorithms can fit CA rules to achieve specific result data from a CA [47]. The researchers were able to reproduce Wolfram ECA Rule 126 from only its output. Though similar, these techniques are not useful to evolve update rules for a generative art producing CA because we do not have data that specifically describes what the output of the generative art CA will actually look like.

Objective function design is one of the most complex aspects of any evolutionary algorithm [27, 45]. Wolfram's four classes of CA [57] lack the granularity needed for an objective function to effectively sort a population. Other techniques make use of an existing image to guide the evolution [17, 34, 42]. *MergeLife* evolves update rules based on multiple objectives governed by human subjective weightings of several statistical measures over multiple CA generations.

## 9 Conclusions and next steps

We have already translated *MergeLife* into several computer programming languages, which are available on GitHub. Currently *MergeLife* is available in Python, Java, and JavaScript. The Java version of *MergeLife* uses an Encog GA [16]. Contributions of code ported to other languages are welcome and can be submitted via GitHub. The current Python implementation achieves decent performance through the use of Numpy [52] and Scipy [20]. More advanced, GPU-enabled frameworks such as TensorFlow [1] would likely enhance performance. TensorFlow can also offload some of the computation work to Graphical Processing Units (GPUs) which could be well aligned with *MergeLife* calculation needs.

The current code base for *MergeLife* uses a finite grid of cells. Conceptually, GOL is often described in terms of updating a grid of cells. Though easy to understand, this implementation has two flaws. The first flaw is that all processing must be contained inside of this fixed grid. If part of the pattern moves outside of the fixed grid, that part of the pattern will be lost. The second limitation is that even though much of the grid might be empty, the GOL update rule must recalculate the entire grid. These limitations led to the development of the *HashLife* algorithm [14]. The *HashLife* algorithm, developed for GOL, could be applied to *MergeLife* to provide both performance improvements as well as an infinite grid.

*MergeLife* currently uses a 2D, rectangular grid, with a Moore Neighborhood. Expansion of these design characteristics would not be difficult. Moving from 2D to 3D, or higher, would involve a change to the neighbor count. Other lattice types could also be employed and would require corresponding changes to the grid storage and neighbor counting. Both the neighborhood radius and type could be changed, requiring corresponding changes to the neighbor count function.

*MergeLife* could benefit from further refinement of the objective function. The objective function is noisy due to the random initial states of the grid. A better objective score might be achieved by considering neighboring GA population members [29]. This includes both fully automated objective functions and hybrid human supplemented objective functions. Automatic objective functions could be created that detect spaceships, guns, rakes, oscillators, and other desirable features. Detection of such features could be added to the current set of statistics that are calculated by the current multi-objective function. This could involve interaction with the user as the objective function evaluates more and more cases. Allowing the user to add their own objective weightings to the top members of the population could allow even further refinement upon the specific aesthetic qualities that a user is looking for.

Novelty search has recently been incorporated into a number of artistic optimization problems [50]. It might be possible to search for more novel update rules than the ones previously discovered. The hexadecimal strings for many MergeLife update rules appear quite different, yet produce very similar visual patterns. Additionally, the study of individual *MergeLife* update rules could yield information about other patterns supported and Turing Completeness. Many of the interesting patterns discovered by GOL are engineered and do not occur naturally from a

random grid initialization. Similar engineering might create interesting structures for *MergeLife* update rules such as *Yellow World* (Fig. 4).

**Open Access** This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

## References

1. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, Tensorflow: a system for large-scale machine learning. OSDI **16**, 265–283 (2016)
2. A. Adamatzky, G.J. Martínez, *Designing Beauty: The Art of Cellular Automata*, vol. 20 (Springer, Berlin, 2016)
3. B. Alexander, J. Kortman, A, Neumann, Evolution of artistic image variants through feature based diversity optimisation, in *Proceedings of the Genetic and Evolutionary Computation Conference* (ACM, New York, 2017), pp. 171–178
4. S. Bergen, B.J. Ross, Aesthetic 3d model evolution. Genet. Program. Evolvable Mach. **14**(3), 339–367 (2013)
5. E. Berkelamp, J.H. Conway, R.K. Guy, *Winning-Ways for Your Mathematical Plays* (Academic Press, London, 1982)
6. A. Chavoya, Y. Duthen, Using a genetic algorithm to evolve cellular automata for 2d/3d computational development, in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (ACM, New York, 2006), pp. 231–232
7. J. Clune, H. Lipson, Evolving three-dimensional objects with a generative encoding inspired by developmental biology, in *ECAL* (2011), pp. 141–148
8. S. Colton, Creativity versus the perception of creativity in computational systems, in *AAAI Spring Symposium: Creative Intelligent Systems*, vol. 8 (2008)
9. S. Colton, J.W. Charnley, A. Pease, Computational creativity theory: the face and idea descriptive models, in *ICCC* (2011), pp. 90–95
10. M. Cook, Universality in elementary cellular automata. Complex Syst. **15**(1), 1–40 (2004)
11. K.M. Evans, Larger than life: digital creatures in a family of two-dimensional cellular automata, in *DM-CCG* (2001), pp. 177–192
12. K.M. Evans, Larger than life: threshold-range scaling of lifes coherent structures. Phys. D Nonlinear Phenom. **183**(1), 45–67 (2003)
13. M. Gardner, The fantastic combinations of John Conway's new solitaire game life. Sci. Am. **223**, 120–123 (1970)
14. R.W. Gosper, Exploiting regularities in large cellular spaces. Phys. D Nonlinear Phenom. **10**(1–2), 75–80 (1984)
15. D.A. Hart, Toward greater artistic control for interactive evolution of images and animation, in *Workshops on Applications of Evolutionary Computation* (Springer, Berlin, 2007), pp. 527–536
16. J. Heaton, Encog: library of interchangeable machine learning models for java and c#. J. Mach. Learn. Res. **16**, 1243–1247 (2015)
17. E.D. Heijer, A. Eiben, Using scalable vector graphics to evolve art. Int. J. Arts Technol. **9**(1), 59–85 (2016)
18. T.J. Hutton, Evolvable self-reproducing cells in a two-dimensional artificial chemistry. Artif. Life **13**(1), 11–30 (2007). https://doi.org/10.1162/artl.2007.13.1.11
19. C.G. Johnson, Fitness in evolutionary art and music: a taxonomy and future prospects. Int. J. Arts Technol. **9**(1), 4–25 (2016)
20. E. Jones, T. Oliphant, P. Peterson, et al., SciPy: open source scientific tools for Python (2001–). http://www.scipy.org/
21. A. Jordanous, A standardised procedure for evaluating creative systems: computational creativity evaluation based on what it is to be creative. Cogn. Comput. **4**(3), 246–279 (2012)

22. A. Jordanous, Four PPPPerspectives on computational creativity, in *AISB 2015 Symposium on Computational Creativity* (2015), pp. 16–22
23. K. Kaneko, Pattern dynamics in spatiotemporal chaos: pattern selection, diffusion of defect and pattern competition intermettency. Phys. D Nonlinear Phenom. **34**(1–2), 1–41 (1989)
24. A. Kosorukoff, Human based genetic algorithm, in *2001 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 5 (IEEE, 2001), pp. 3464–3469
25. M. Lewis, Evolutionary visual art and design, in *The Art of Artificial Evolution*, ed. by J. Romero, P. Machado (Springer, Berlin, 2008), pp. 3–37
26. A. Lindenmayer, Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. J. Theor. Biol. **18**(3), 280–299 (1968)
27. P. Machado, T. Martins, H. Amaro, P.H. Abreu, An interface for fitness function design, in *International Conference on Evolutionary and Biologically Inspired Music and Art* (Springer, Berlin, 2014), pp. 13–25
28. O. Martin, A.M. Odlyzko, S. Wolfram, Algebraic properties of cellular automata. Commun. Math. Phys. **93**(2), 219–258 (1984)
29. R. Miikkulainen, H. Shahrzad, N. Duffy, P. Long, How to select a winner in evolutionary optimization? in *Proceedings of the IEEE Symposium Series in Computational Intelligence* (IEEE, 2017)
30. M. Mitchell, *An Introduction to Genetic Algorithms* (MIT Press, Cambridge, 1998)
31. M. Mitchell, J.P. Crutchfield, R. Das, Evolving cellular automata with genetic algorithms: a review of recent work, in Proceedings of the First International Conference on Evolutionary Computation and Its Applications (EvCA'96) (Russian Academy of Sciences, Moskva, 1996)
32. A. Neumann, B. Alexander, F. Neumann, Evolutionary image transition using random walks, in *International Conference on Evolutionary and Biologically Inspired Music and Art* (Springer, Berlin, 2017), pp. 230–245
33. C.P. Newland, H.R. Maier, A.C. Zecchin, J.P. Newman, H. van Delden, Multi-objective optimisation framework for calibration of cellular automata land-use models. Environ. Model. Softw. **100**, 175–200 (2018)
34. S. Nichele, M.B. Ose, S. Risi, G. Tufte, CA-NEAT: evolved compositional pattern producing networks for cellular automata morphogenesis and replication. IEEE Trans. Cogn. Dev. Syst. (2017). https://doi.org/10.1109/TCDS.2017.2737082
35. M.C. Olmedo, Multi-objective land allocation (MOLA), in *Geomatic Approaches for Modeling Land Change Scenarios*, ed. by M.T. Camacho Olmedo, M. Paegelow, J.F. Mas, F. Escobar (Springer, Berlin, 2018), pp. 457–460
36. P. Prusinkiewicz, A. Lindenmayer, *The Algorithmic Beauty of Plants* (Springer, Berlin, 2012)
37. S. Rafler, Generalization of Conway's "game of life" to a continuous domain-smoothlife. arXiv preprint arXiv:1111.1567 (2011)
38. P. Rendell, *Turing Machine Universality of the Game of Life* (Springer, Berlin, 2016)
39. R. Rucker, *Continuous-Valued Cellular Automata in Two Dimensions* (Oxford University Press, Oxford, 2003)
40. J. Secretan, N. Beato, D.B. D Ambrosio, A. Rodriguez, A. Campbell, K.O. Stanley, Picbreeder: evolving pictures collaboratively online, in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (ACM, New York, 2008), pp. 1759–1768
41. J. Secretan, N. Beato, D.B. D'Ambrosio, A. Rodriguez, A. Campbell, J.T. Folsom-Kovarik, K.O. Stanley, Picbreeder: a case study in collaborative evolutionary exploration of design space. Evol. Comput. **19**(3), 373–403 (2011)
42. J.K. Shin, *Application of Cellular Automata for a Generative Art System* (Leonardo, Cambridge, 2016)
43. K. Sims, Interactive evolution of dynamical systems, in *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life* (1992), pp. 171–178
44. K.O. Stanley, Compositional pattern producing networks: a novel abstraction of development. Genet. Program. Evolvable Mach. **8**(2), 131–162 (2007)
45. K.O. Stanley, J. Lehman, *Why Greatness Cannot be Planned: The Myth of the Objective* (Springer, Berlin, 2015)
46. K.O. Stanley, R. Miikkulainen, Evolving neural networks through augmenting topologies. Evol. Comput. **10**(2), 99–127 (2002)
47. X. Sun, P.L. Rosin, R.R. Martin, Fast rule identification and neighborhood selection for cellular automata. IEEE Trans. Syst. Man Cybern. Part B (Cybern.) **41**(3), 749–760 (2011)

48. H. Takagi, Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. Proc. IEEE **89**(9), 1275–1296 (2001)
49. A. Tantoushian, Scaling larger than life bugs: to range 25 and beyond. Ph.D. thesis, California State University, Northridge (2017)
50. A. Vinhas, F. Assunção, J. Correia, A. Ekárt, P. Machado, Fitness and novelty in evolutionary art, in *International Conference on Evolutionary and Biologically Inspired Music and Art* (Springer, Berlin, 2016), pp. 225–240
51. J. Von Neumann, A.W. Burks, Theory of self-reproducing automata. IEEE Trans. Neural Netw. **5**(1), 3–14 (1966)
52. S. Walt, S.C. Colbert, G. Varoquaux, The numpy array: a structure for efficient numerical computation. Comput. Sci. Eng. **13**(2), 22–30 (2011)
53. E.W. Weisstein, Elementary cellular automaton. From MathWorld—a wolfram web resource. http://mathworld.wolfram.com/ElementaryCellularAutomaton.html. Last visited on 11/6/2017
54. M. Wojtówicz, Cellular automata rules lexicon. http://psoup.math.wisc.edu/mcell/rullex_lgtl.html. Last visited on 11/6/2017
55. S. Wolfram, Cellular automata. Los Alamos Sci. **9**, 2–27 (1983)
56. S. Wolfram, Statistical mechanics of cellular automata. Rev. Mod. Phys. **55**(3), 601 (1983)
57. S. Wolfram, *A New Kind of Science*, vol. 5 (Wolfram Media, Champaign, 2002)
58. S. Wolfram et al., *Theory and Applications of Cellular Automata*, vol. 1 (World Scientific, Singapore, 1986)
59. D. Wolz, P.P. De Oliveira, Very effective evolutionary techniques for searching cellular automata rule spaces. J. Cell. Autom. **3**(4), 289–312 (2008)

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.