Algorithmic Specified Complexity in the Game of Life

Winston Ewert, William Dembski, Senior Member, IEEE, and Robert J. Marks, II, Fellow, IEEE

Abstract—Algorithmic specified complexity (ASC) measures the degree to which an object is meaningful. Neither fundamental Shannon nor Kolmogorov information models are equipped to do so. ASC uses performance context in an information theoretic framework to measure the degree of specified complexity in bits. To illustrate, we apply ASC to Conway’s Game of Life to differentiate patterns designed by programmers from those originating by chance. A variety of machines created by Game of Life hobbyists, as expected, exhibit high ASC thereby corroborating ASC’s efficacy.

Index Terms—Algorithmic specified complexity, cellular automata, Conway’s Game of Life, Kolmogorov information, Kolmogorov-Chaitin-Solomonoff information, Shannon information, specified complexity.

I. INTRODUCTION

Both Shannon et al. [1], [2] and Kolmogorov–Chaitin–Solomonoff (KCS)1 [2]–[9] measures of information are famous for not being able to measure meaning. A DVD containing the movie Braveheart and a DVD full of correlated random noise can both require the same Shannon information as measured in bytes. Likewise, a maximally compressed text file with fixed byte size can either contain a classic European novel or can correspond to random meaningless alphanumeric characters. The KCS measure of information is therefore also not able to, by itself, measure informational meaning.

We propose an information theoretic method to measure meaning [10], [11]. Fundamentally, we model meaning to be in the context of the observer. A page filled with Kanji symbols will have little meaning to someone who neither speaks nor reads Japanese. Likewise, a machine is an arrangement of parts that exhibit some meaningful function whose appreciation requires context. The distinguishing characteristic of machines is that the parts themselves are not responsible for the machine’s functionality, but rather they are only functional due to the particular arrangement of the parts. Almost any other arrangement of the same parts would not produce anything interesting. A functioning computational machine is more meaningful than a large drawer full of computer parts.

1Sometimes referred to as only Kolmogorov complexity or Kolmogorov information.
Some oscillate, some move, some produce other patterns, etc. Some of these are simple enough that they arise from random configurations of cell space. Others required careful construction, such as the very large Gemini [26]. Our goal is to formulate and apply specified complexity measures to these patterns. We would like to be able to quantify what separates a simple glider, readily produced from almost any randomly configured soup, from Gemini—a large, complex design whose formation by chance is probabilistically minuscule. Likewise, we would like to be able to differentiate the functionality of Gemini from a soup of randomly chosen pixels over a similarly sized field of grid squares.

A highly probable object can be explained by randomness, but it will lack complexity and thus not have specified complexity. Conversely, any sample of random noise will be improbable, but will lack specification and thus also lack specified complexity. In order to have specified complexity, both components must be present. The object must exhibit a describable functioning pattern while being improbable.

Our paper differs from the study of emergence in cellular automata first proposed by von Neumann [27], [28] for investigating self-reproduction. The study of dynamic cellular automata properties, popularized by Wolfram [29], deals largely with investigation of the temporal development of emergent behavior [30]. As an example, the set of all initializations that lead to the same emergent behavior, dubbed the emergent behavior [30]. As an example, the set of all initializations that lead to the same emergent behavior, dubbed the emergent behavior [30]. As an example, the set of all initializations that lead to the same emergent behavior, dubbed the emergent behavior [30]. As an example, the set of all initializations that lead to the same emergent behavior, dubbed the emergent behavior [30]. As an example, the set of all initializations that lead to the same emergent behavior, dubbed the emergent behavior [30]. As an example, the set of all initializations that lead to the same emergent behavior, dubbed the emergent behavior [30].

A small value of $K(X|\gamma) < K(X) < K(X|\gamma, \alpha)$. KCS complexity or variations thereof have been previously proposed as a way to measure the conditional KCS complexity, we would expect $K(X|Y_{\alpha}) < K(X|Y_{\gamma}) < K(X)$. The more specific the context, the smaller the elite program. Either the frequency of occurrence of the words used by Shakespeare or a concordance of words used only in Hamlet can be used to reduce the conditional KCS complexity even further. Small conditional KCS complexity can be caused by the following.

1) Placing $X$ in the context of $Y$ and/or
2) A small (unconditional) KCS complexity, $K(X)$.

A small value of $K(X|Y)$ can therefore arise from the small complexity of $X$ and/or from the available context, $Y$.

A. ASC: Actual and Observed

Algorithmic specified complexity [41] is defined as

$$ASC(X, C, P) = I(X) - K(X|C)$$

where

1) $X$ is the object or event under consideration;
2) $C$ is the context (given information) which can be used to describe the object;
3) $K(X|C)$ is the KCS complexity of object $X$ given context $C$;
4) $P(X)$ is the probability of $X$ under the given stochastic model;
5) $I(X) = -\log_2(P(X))$ is the corresponding self-information.

The ASC measure bears a resemblance to both Shannon [1], [2] and KCS [23] mutual information. ASC is probabilistically rare in the sense that [42]

$$\Pr[ASC(X, C, P) \geq \alpha] \leq 2^{-\alpha}.$$  \hspace{1cm} (2)

For example, the chance of observing ten or more bits of ASC does not exceed $2^{-10} \approx$ one chance in a thousand. ASC provides evidence that a stochastic outcome modeled by the distribution, $P(X)$, does not explain a given object. ASC is incomputable because KCS complexity is incomputable [2]. However, the true KCS complexity is always equal to or less than any achieved lossless compression. This means that the true ASC is always equal to or more than an estimate. We will refer to the known estimate as the observed observed algorithmic specified complexity (OASC). We know that

$$ASC(X, C, P) \geq OASC(X, C, P).$$ \hspace{1cm} (3)
The inequality in (2) applies to OASC. From (3), we conclude there is a $k \geq 0$ such that

$$OASC = ASC - k.$$ 

Thus

$$\Pr[OASC \geq \alpha] = \Pr[ASC - k \geq \alpha]$$

$$= \Pr[ASC \geq \alpha + k]$$

$$\leq 2^{-\alpha - k}$$

$$\leq 2^{-\alpha}.$$ 

(4)

OASC therefore obeys the same bound as does ASC in (2).

ASC can be nicely illustrated using various functional patterns in Conway’s Game of Life. The Game of Life and similar systems allow a variety of fascinating behaviors [29]. In the game, determining the probability of a pattern arising from a random configuration of cells is difficult. The complex interactions of patterns arising from such a random configuration makes it difficult to predict what types of patterns will eventually arise. It would be straightforward to calculate the probability of a pattern arising directly from some sort of random pattern generator. However, once the Game of Life rules are applied, determining what patterns would arise from the initial random patterns is nontrivial. In order to approximate the probabilities, we will assume that the probability of a pattern arising is about the same whether or not the rules of the Game of Life are applied, i.e., the rules of the Game of Life do not make interesting patterns much more probable then they would otherwise be.

Objects with high ASC defy explanation by the stochastic process model. Thus, we expect objects with large ASC are designed rather than arising spontaneously. Note, however, we are only approximating the complexity of patterns and the result is only probabilistic. We expect that patterns requiring more design will have higher values of ASC. Smaller designed patterns exist, but it is not possible to conclude that they were not produced by random configurations.

Section III documents the methodology of the paper. We define a mathematical formulation to capture the functionality of various patterns. This can be encoded as a bitstring and a program written to generate the original pattern from this functional description. Section IV uses this methodology to calculate ASC for a variety of patterns found in the Game of Life.

III. METHODS

A. Specification

The Game of Life is played on grid of square cells. A cell is either alive (a one) or dead (a zero). A cell’s status is determined by the status of other cells around it. Only four rules are followed.

1) Under-Population: A living cell with fewer than two live neighbors dies.

2) Family: A living cell with two or three live neighbors lives on to the next generation.

3) Overcrowding: A living cell with more than three living neighbors dies.

4) Reproduction: A dead cell with exactly three living neighbors becomes a living cell.

As witnessed by videos on YouTube, astonishing functionality can be achieved with these few simple rules [43]–[45]. If the reader is unfamiliar with the diversity achievable with these operations, we encourage them to view these and other short videos demonstrating the Game of Life. The static pictures in this paper do not do justice to the remarkable underlying dynamics. There is also an active users group [46].

The rules for the Game of Life are deterministic. Performance is therefore dictated only by the initially chosen pattern. In order to demonstrate the compression of functional Game of Life patterns, we first devise a contextual mathematical formulation for describing functionality. A method for interpreting this formulation is considered to be part of the context. Let $X$ be some arbitrary pattern corresponding to a configuration of living and dead pixels. Let $X \oplus$ be the result of one iteration of the Game of Life applied to $X$. Suppose that the following equality holds:

$$X = X \oplus.$$ 

This says that a pattern does not change from one iteration to the next. This is known as a still-life [25], and an example is presented in Fig. 1. A more interesting pattern can be described as

$$X = X \oplus \oplus$$

which can be a pattern that returns to its original state after two iterations. The relationship is also valid for two iterations of a still-life. In order to differentiate a two-iteration flip-flop from a still life form, two equations are required

$$X \not= X \oplus$$

$$X = X \oplus^2.$$ 

(5)

We often need to specify that a rule holds only for some parameter and not for any smaller version of that. We therefore adopt the notation

$$X = X \oplus^i$$ 

(6)

to mean a pattern that repeats in $i$ iterations, but not in less than $i$ iterations. An example for $i = 2$, shown in Fig. 2, is a period-2 oscillator [46] or a flip-flop [25].

One of the more famous Game of Life patterns is the glider. This is a pattern which moves as it iterates. A depiction is shown in Fig. 3. In order to represent movements we introduce arrows, so $X \uparrow$ is the pattern $X$ shifted up one row. Since four
Fig. 2. Blinker, a simple period-2 oscillator.

Fig. 3. Glider, a simple spaceship.

iterations regenerate the glider shifted one unit to the right and one unit down, we can write
\[ X \downarrow \rightarrow = X \oplus 4. \] (7)
This defines the functionality of moving in the direction and speed of the glider.

We can also describe a pattern as the set-difference of two other patterns. Since \( A \setminus B \) denote elements in \( A \) not in \( B \), we have for example
\[ X = X \setminus \bar{X}. \] (8)

At times, it may be useful to define variables. For example
\[ Y := X \oplus 32 \] (9)
\[ Y = Y \oplus 32 \] (10)
where : = denotes “equal to by definition.” This reduces to
\[ X \oplus 32 = X \oplus 64. \]

Table I provides a listing of operations. The selected set of operation was chosen in attempt to cover all bases which might be useful, but is still arbitrary. Another set of operations with more or less power could be chosen. By using a consistent set of operations between the various examples explored in this paper, we obtain comparable OASC values.

More that one \( X \) will display two step oscillation in accordance to \( X = X \oplus 2 \). In fact, this and other equations will admit an infinite set of patterns that satisfy the description. In order to make an equation description apply to a unique pattern, we can lexicographically order all patterns obeying a description. This can be done by defining a box that bounds the initial pattern. To do so, we will constrain initialization to a finite number of living cells to avoid an infinite number of living cells.

The full ordering can be defined by the following priority set of rules with lower-numbered rules listed first.

1) Smaller number of living cells.
2) Smaller bounding box area.
3) Smaller bounding box width.
4) Lexicographically ordering according to the encoding of cells within a box bounding the living cells. For example, bounding the living cells in the upper left configuration in Fig. 3 and reading left to right then down gives 010001111 = (143)\(_{10}\).

The first rule could be removed, leaving a consistent system. However, among Game of Life hobbyists, patterns with fewer living cells are considered smaller and maintain this for
TABLE II

<table>
<thead>
<tr>
<th>Nullary operations</th>
<th>Symbol</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>00000</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>00001</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>00010</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>00011</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>00100</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>00101</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>00110</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>00111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unary Operations</th>
<th>Symbol</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊕</td>
<td>01000</td>
<td></td>
</tr>
<tr>
<td>↑</td>
<td>01001</td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td>01010</td>
<td></td>
</tr>
<tr>
<td>←</td>
<td>01111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binary Operations</th>
<th>Symbol</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊗</td>
<td>01101</td>
<td></td>
</tr>
<tr>
<td>⊖</td>
<td>01110</td>
<td></td>
</tr>
<tr>
<td>⊖</td>
<td>01111</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>10000</td>
<td></td>
</tr>
<tr>
<td>−</td>
<td>10001</td>
<td></td>
</tr>
<tr>
<td>×</td>
<td>10010</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>10011</td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>10100</td>
<td></td>
</tr>
<tr>
<td>≠</td>
<td>10101</td>
<td></td>
</tr>
<tr>
<td>(repeat)</td>
<td>10110</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Literals</th>
<th>Type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>11001</td>
<td></td>
</tr>
<tr>
<td>Pattern</td>
<td>11010</td>
<td></td>
</tr>
<tr>
<td>Special</td>
<td>11111</td>
<td></td>
</tr>
</tbody>
</table>

consistency. This does add some complications to the model which is discussed in the Appendix.

We will append each equation with a number, in the form #i indicating that we are interested in the ith pattern to fit the equation. Thus, the glider becomes

\[ X \downarrow \rightarrow = X \oplus^4, \#0 \]

as the smallest pattern which fits the description.

In our discussions of ASC, establishing rules for lexicographical ordering is important whereas assessing the computational resources needed to explicitly populate the list is not.

B. Binary Representation

In order to use the ASC results, we need to encode the mathematical representation as a binary sequence. Each symbol is assigned a 5-bit binary code as specified in Table II. Any valid formula will be encoded as a binary string using those codes. All such formulas will be encoded as prefix-free codes.

Firstly, a number of the operations have zero arguments, known as nullary operators. These are listed first in Table II. Such operations are simply encoded using their 5-bit sequence. Since they have no arguments, their sequence is completed directly after the five bits. As noted, a different set of operations could be chosen that would require a different number of bits to specify. Thus, X will be encoded as 00000 and W will be encoded as 00010. All the nullary operations are trivially prefix free since all have exactly five bits.

An operation that takes a single argument, known as a unary operation, can be encoded with its 5-bit code followed by representation of the subexpression. Thus, \( X \uparrow \) can be represented as 0101000000. Since the subexpression can be represented in a prefix free code, we can determine the end of it, and adding five bits to the beginning maintains the prefix-free property.

Operations with two arguments, or binary operations, are encoded using the 5-bit sequence followed by the sequence for the two subexpressions. So \( X = X \oplus \) can be recorded as 101010000011000000. \( \oplus^4 \) can be recorded as 1011110100001100. Note that \( \oplus \) usually takes an argument, but this is not needed when it is used as the target of a repeat. As with the unary case, the prefix free nature of the subexpressions allows the construction of the large formula.

The literals in Table II are denoted by the 5-bit code along with an encoding of the integer or pattern. Any positive integer \( n \) can be encoded using \( \lceil \log_2(n+1) + \log_2 n \rceil + 1 \) bits, hereafter \( l(n) \) bits in a prefix free code using the Levenstein code [47]. See Section III-C for a discussion of binary encodings for arbitrary patterns.

To declare there are no more operations to be had, we will use the five bit sequence, 11111. Simply concatenating all the equations would not be a prefix-free code since the binary encoding would be a valid prefix to other codes. After the last equation, 11111 is appended as a suffix preventing any longer codes from being valid and making the system prefix free.

To calculate the length of the encoding we add up the following.

1) Five bits for every symbol.
2) \( l(n) \) bits for each number \( n \) in the equation.
3) The length of the bit encoding of any pattern literals.
4) Five bits for the stop symbol.
5) \( l(n) \) bits for the parameters and sequence numbers.

C. Binary Encoding for Patterns

In order to use OASC we need to define the complexity or probability of the patterns. We would like to define the probability based on the actual probability of the pattern arising from a random configuration. We will model the patterns as being generated by a random sequence of bits.

In order to use a random encoding of bits, we need to define the bit encoding for a Game of Life pattern. Section III-B contains a definition of an encoding, but it is based on functionality. The probability of a pattern arising is clearly not related to its functionality, and thus this measure is not a useful encoding for this purpose.

There are different ways to define this encoding. We can encode the width and height of the encoding using Levenstein encoding and each cell encoded as a single bit indicating whether it is living or not. This gives a total length of

\[ \alpha(p) = l(p_w) + l(p_h) + p_w p_h \]

where \( p_w \) is the width of the pattern \( p \) and \( p_h \) is the height of the pattern. We will call this the standard encoding.
Fig. 4. Gosper gliding gun.

However, in many cases patterns consist of mostly dead cells. A lot of bits are spent indicating that a cell is empty. We can improve this situation by recording the number of live cells and then we can encode the actual pattern using less bits.

\[ \beta(p) = l(p_w) + l(p_h) + l(p_a) + \left\lceil \log_2 \left( \frac{p_w p_h}{p_a} \right) \right\rceil \]

where \( p_a \) is the number of alive cells. We will call this compressed encoding.

To demonstrate these methods, consider the Gosper gliding gun in Fig. 4. Using the standard encoding this requires

\[ \alpha(p) = l(p_w) + l(p_h) + p_w p_h \]
\[ = l(36) + l(9) + 36 \times 9 \]
\[ = 12 + 8 + 324 \]
\[ = 344 \text{ bits}. \]

Using the compressed encoding requires

\[ \beta(p) = l(p_w) + l(p_h) + l(p_a) + 2 \left\lfloor \log_2 \left( \frac{p_w p_h}{p_a} \right) \rceil \rceil \]
\[ = 12 + 8 + 12 + 160 = 192 \text{ bits}. \]

The compressed method will not always produce smaller descriptions as it does here. However, we can use both methods, and simply add an initial bit to specify which method was being used. Thus, the length of the encoding for a pattern, \( p \) is then

\[ P(p) = 1 + \min(\alpha(p), \beta(p)) \]

(11)

where the 1 is to account for the extra bit used to determine which of the two methods was used for encoding.

However, we need to determine the Shannon information for a pattern, \( p \). There are two ways to encode each pattern and both need to be considered

\[ \Pr[X = p] = \Pr[X = p|C] \Pr[C] + \Pr[X = p|\overline{C}] \Pr[\overline{C}] \]

where \( X \) is the random variable of the chosen pattern, and \( C \) is the random event which is true when the compressed encoding is used. Since either method is chosen with 50% probability

\[ \Pr[X = p] = \frac{2^{-\alpha(p)}}{2} + \frac{2^{-\beta(p)}}{2} = \frac{2^{-\alpha(p)} + 2^{-\beta(p)}}{2}. \]

Our primary purpose in this paper is to demonstrate OASC for functional machines in the Game of Life. However, the process also serves as a test of the hypothesis that the approximation to the probability of a pattern and its corresponding information in (11) arising is reasonably close. Are there features of random Game of Life patterns that tend to produce additional functionality? If so, we expect that we will obtain larger than expected values of ASC.

D. Computability

The contextual mathematical formulation thus far developed here for the Game of Life is less powerful than a Turing complete language. For example, there is no conditional looping mechanism. The Game of Life itself is Turing complete [48]; however, our equations using the components in Table II describing the Game of Life are not. There are concepts that cannot be described using the operations we have defined. A large array of a billion closely spaced albeit noninteracting blinkers has low KCS complexity akin to the celebrated low KCS complexity of a repetitive crystalline structure. A looping or a \texttt{REPEAT} command is required to capture low KCS complexity bound in such cases. The list in Table II of course, can be expanded to include these and other cases. However, the proof on the bound of ASC only requires that the language used to describe the pattern is prefix-free. Thus, the ASC bounds using the context in Table II still apply to the language defined here.

In order to use ASC, we must algorithmically derive the machine from the equations describing it. A program would systematically test all pattern in order of increasing size while checking whether they pass the test. We term this program the interpreter. Since the pattern specified whether it is the first, second, third, etc., pattern to pass the test, the process can stop and output the pattern once it is reached. Thus, a constant length interpreter program can derive the pattern from the equations, and ASC using a standard Turing machine is a constant longer than the OASC results presented here. If an alternate formulation is used to describe the pattern, then a different constant would apply as a different interpreter would be required.

The language used here is motivated in part for simplicity in understanding. It allows the comparison of the complexity of various specifications without constants which is difficult in standard KCS complexity.

Essentially, we have included the interpreter for our formulation as part of the context. The interpreter has details on the Game of Life, but not on the nature of patterns in it. This allows the description of the pattern in the Game of Life without any undue bias toward the patterns found in the Game of Life.

IV. Results

A. Oscillators

The simplest oscillator is one which does not actually change, i.e., a still life. An example is depicted in Fig. 1. This object can be described as

\[ X = X \otimes, \#0 \]

(12)
Table III

| Name              | Period | Complexity | \(K(X|C)\) | OASC  | Bound         | \(Pr[X]\) |
|-------------------|--------|------------|------------|-------|---------------|-----------|
| block             | 1      | 12.68      | -25.32     | 4.189\*10^{-9} | 3.232\*10^{-01} |
| blinker           | 2      | 10.68      | -29.32     | 6.702\*10^{-8} | 3.292\*10^{-01} |
| caterer           | 3      | 61.68      | 20.68      | 5.953\*10^{-7} | 7.692\*10^{-11} |
| mazing            | 4      | 60.83      | 18.83      | 2.146\*10^{-6} | 4.545\*10^{-09} |
| pseudo-barberpole | 5      | 95.0       | 52.0       | 2.220\*10^{-16} |                  |
| unix              | 6      | 75.96      | 32.96      | 1.197\*10^{-10} | 5.882\*10^{-10} |
| burloaferimeter   | 7      | 117.0      | 74.0       | 5.294\*10^{-23} |                  |
| figure eight      | 8      | 50.91      | 6.91       | 8.315\*10^{-03} | 3.030\*10^{-08} |
| 29p9              | 9      | 113.96     | 68.96      | 1.742\*10^{-21} |                  |

As this is the smallest pattern that can fit the test. There are four symbols taking 20 bits to describe. There are five bits for the stop symbol and one bit to describe the sequence number. This gives a total of 26 bits to describe this pattern. Using standard encoding will require \(l(2) + l(2)\times 2 + 1 = 4 + 4 + 4 = 13\). Thus, the ASC is at least 32 bits, because that is the overhead required to simply embed the pattern in its own description.

A flip-flop or a period two oscillator can be described as

\[X = X \oplus i, \quad i = 2, \#0. \tag{13}\]

This takes six symbols (the repeat counts as a symbol) plus the stop symbol the parameter and the sequence number. That is a total of 35 + \(l(2) + l(0) = 35 + 4 + 1 = 40\) bits. The blinker encoded using standard encoding will take \(l(1) + l(3) + 3 + 1 = 2 + 5 + 3 + 1 = 11\) bits. The OASC is then 11 - 40 = -29 bits. Again, this pattern fits the modeled stochastic process well.

A spaceship is a pattern in life which travels across the grid. It continually returns back to its original state but in a different position. The first discovered spaceship was the glider depicted in Fig. 3. We previously showed in (7) that it could be described as

\[X \downarrow \rightarrow = X \oplus 4, \#0. \tag{14}\]

which has three symbols, and will require 11 bits for the pattern. The \#0 is required, despite there being only one pattern which fits the equation, for consistency with the search process described in Section III-D. Thus, the length is 3 \(\times 5 + 5 + l(0) + 11 = 20 + 1 + 11 = 32\) giving at least 11 - 32 = -21 bits of ASC. In fact any pattern can be said to have at least -21 bits of ASC, because that is the overhead required to simply embed the pattern in its own description.

As with oscillators we can readily describe the smallest version of a spaceship. In addition to varying with respect to the

\[\text{TABLE III}

\text{ASC FOR THE SMALLEST KNOWN OSCILLATORS IN EACH CATEGORY}\]
period, spaceships vary with respect to the speed and direction. Speeds are rendered as fractions of $c$, where $c$ is one cell per iteration. First we will consider spaceships that travel diagonally like the glider. In general to travel with a speed of $c/s$ with period $p$ can be described as

$$X \downarrow \frac{c}{s} \rightarrow X \oplus p, \#0.$$  

(15)

This describes a spaceship moving down and the right. Due to the symmetry of the rules of the Game of Life, the same spaceships could all be reoriented to point in different directions. That would change the direction of the arrows, but not the length of the description. The length of this is $5 \times 12 + 5 + l(\frac{c}{s}) + l(p) + l(0) = 66 + l(\frac{c}{s}) + l(p)$.

Fig. 6 shows the smallest known diagonally moving spaceships for different speeds. If we assume that these are the smallest spaceships for these speeds, then (15) describes them.

**Table IV**

| Name        | Period | Speed | Complexity | $K(X|C)$ | OASC |
|-------------|--------|-------|------------|----------|------|
| glider      | 4      | 19.96 | 74.0       | -54.04   |
| 58P5H1V1    | 5      | 296.0 | 75.0       | 221.0    |
| 77P6H1V1    | 6      | 459.0 | 75.0       | 384.0    |
| 83P7H1V1    | 7      | 733.0 | 75.0       | 658.0    |
| Four engine cordership | 96     | 962.0 | 89.0       | 873.0    |

**Table V**

| Name              | Period | Speed | Complexity | $K(X|C)$ | OASC |
|-------------------|--------|-------|------------|----------|------|
| lightweight spaceship | 2      | 33.99 | 57.0       | -23.01   |
| 25P3HV1V0.2       | 3      | 97.0  | 58.0       | 39.0     |
| 37P4H1V0          | 4      | 177.0 | 59.0       | 118.0    |
| 30P5H2V0          | 5      | 133.0 | 62.0       | 71.0     |
| Spider            | 5      | 211.0 | 60.0       | 151.0    |
| 56P6H1V0          | 6      | 242.0 | 60.0       | 182.0    |
| Weekender          | 7      | 158.0 | 62.0       | 96.0     |

Table IV shows the ASC for these various spaceships. The glider has negative ASC, it is simple enough to be readily explained by a random configuration. The remaining diagonal spaceships exhibit a large amount of ASC, fitting the fact that they are all complex designs. This is expected from look at Fig. 6 where the remaining patterns are much larger than the glider.

In addition to the diagonally moving spaceships we can also consider orthogonally moving spaceships. These move in only one direction, and so can be described as

$$X \uparrow \frac{c}{s} = X \oplus p, \#0.$$  

(16)

The length of this is $5 \times 9 + 5 + l(\frac{c}{s}) + l(p) + l(0) = 51 + l(\frac{c}{s}) + l(p) + l(0)$. As with the diagonal spaceships, the same designs can be reoriented to move in any direction. The equation can be updated by simply changing the arrow. Fig. 7 shows the smallest known spaceship for a number of different speeds. Table V shows the ASC for the various spaceships. The simplest orthogonal spaceship, the lightweight spaceship, has negative bits of ASC. This matches the observation that these spaceships do arise out of random configurations [51]. The remaining spaceships exhibit significant amounts of ASC, although not as much as the diagonal spaceships, and are not reported to have been observed arising at random.
C. Guns

Fig. 8 shows the Gosper gun running through 31 iterations. The 30th iteration is the same as the original configuration except that it also includes a glider. The glider will escape and the gun will continue to produce gliders indefinitely. This is known as a gun. We can describe this gun as

\[ X \oplus^{30} = X \cup \mathbb{G} \rightarrow 24 \downarrow 10, \#0. \]  

(17)

That is, the configuration after 30 iterations is equal to the original configuration with a glider added at a particular position. There are 60 bits for the symbols and it will require 20 bits to describe the glider, so \(60 + 20 + l(30) + l(24) + l(10) + l(0)\) which is \(60 + 20 + 11 + 11 + 8 + 1 = 111\) bits. The complexity is 196 bits. This gives us \(196 - 111 = 85\) bits of OASC. At a probability of \(2^{-85}\), we conclude the Gosper gun is unlikely to be produced by a random configuration.

D. Eaters

Most of the time when a glider hits a still life, the still life will react with the glider and end up being changed into some other pattern. However, with patterns known as eaters, such as that displayed in Fig. 9, the pattern “eats” the incoming glider resulting it returning to its original state. There are two aspects that make it an eater. Firstly, it must be a still life

\[ X = X \oplus. \]  

(18)

Secondly, it must recover from eating the glider

\[ \left( X \cup \mathbb{G} \uparrow^{4} \downarrow^{4} \right) \oplus^{4} = X. \]  

(19)

The two equation have a total of 18 symbols, and the glider will require 20 bits to encode. Thus, the total length will be \(5 \times 18 + 5 + 20 + l(3) + l(4) + l(4) + l(0) = 5 \times 18 + 5 + 20 + 4 + 7 + 7 + 1 = 134\) bits. The complexity of the eater is 29 bits. The OASC is thus \(29 - 134 = -105\) bits. The eater is thus simple enough to be explain by a random configuration.

E. Ash Objects

Within the Game of Life, it is possible to create a random soup of cells and observe what types of objects arise from the soup. The resulting stable objects, still-lifes and oscillators, are known as ash [46]. Experiments have been performed to measure the frequencies of various objects arising from this soup [50]. Fig. 10 shows the ten most common ash objects, together comprising 99.6% of all ash objects. We observe that these objects are fairly small, and thus will not exhibit much complexity. The largest bounding box is \(4 \times 4\) which will require at most \(1 + l(4) + l(4) + 16 = 1 + 7 + 7 + 16 = 31\)
A large amount of ASC. We are not interested in any pattern where there is a gap larger than the size of the observable universe. Let $U = L + T + 1$ where $L$ is the number of living cells in a pattern, and $T$ is the number of $\oplus$ operations. Given a bounding-box larger than $U \times U$, there must exist a gap larger than the size of the observable universe. Consequently there is a finite number of interesting patterns for a given number of living cells, and we can number them.

ACKNOWLEDGMENT

The authors would like to thank the reviewers’ suggestion of contrasting our paper with the study of temporal emergence properties of cellular automata.

REFERENCES


Winston Ewert received the Ph.D. degree from Baylor University, Waco, TX, USA.

He has written a number of papers relating to search, information, and complexity. He was a Research Assistant at the Evolutionary Informatics Laboratory. He is currently a Software Engineer in Kirkland, WA, USA.

William Dembski (M’07–SM’07) received the Ph.D. degrees in mathematics with the University of Chicago, Chicago, IL, USA, in 1988, and in philosophy with the University of Illinois at Chicago, Chicago, IL, in 1996.

He is a mathematician and a philosopher. He has authored over 20 books, and is a Senior Fellow with Discovery Institute, Seattle, WA, USA, and a Senior Research Scientist with the Evolutionary Informatics Laboratory. His current research interests include information-theoretic underpinnings of the natural sciences. His newest book is *Being as Communion: A Metaphysics of Information* (Ashgate Pub. Co., August, 2014).

Robert J. Marks, II (S’71–M’72–SM’83–F’94) is a Distinguished Professor of electrical and computer engineering at Baylor University, Waco, TX, USA.

His most recent books are *Handbook of Fourier Analysis and Its Applications* (Oxford University Press, 2009), and *Biological Information—New Perspectives* (Singapore: World Scientific, 2013), co-edited by M. J. Behe, W. A. Dembski, B. L. Gordon, and J. C. Sanford. He has an Erdős-Bacon number of five.