Is the Kolmogorov Complexity of Computational Intelligence Bounded Above?

Stuart H. Rubin
Space and Naval Warfare
Systems Center
San Diego (SSC-PAC)
CA, USA 92152-5001
stuart.rubin@navy.mil

Abstract

If we can show that the Kolmogorov complexity of computational intelligence is not bounded above, then it follows that the design process for the realization of intelligent algorithms (with emphasis on those for asynchronous concurrent control) can be programmed to some finite point of complexity. Subsequent to that point, cost overruns and eventually catastrophic failure is inevitable. The only effective solution, which follows from the theory presented herein, is twofold. First, it must be realized that there can be no non-trivial gold standard algorithm for computational intelligence. The existence of any such universal algorithm would contradict the semantic randomization problem, which is to say that it would violate the foundations of computability theory. Second, and as a consequence of the first stipulation, current evolutionary paradigms are not capable of realizing the highest levels of complexity. To do so, implies the representation and evolution of heuristics, using \( k \)-limited transformation, which drives the evolutionary process itself. Programming must become a constructive exercise in search, where the machine plays an active role in reducing the cognitive burden of the programmer. Such techniques are shown to critically depend on the evolution of heuristics, which of course involves self-reference for their non-trivial realization. An instance of this concept is genomic evolution. Other instances are not precluded.

Keywords: Computational Intelligence, Evolution, Heuristics, Kolmogorov Complexity

1. Introduction

The motivation for this paper came about through the process of trying to prove some difficult theorems on the nature of algorithms for intelligent computation. Essentially, these algorithms would start out hill climbing a solution to a problem and iteratively enlarge the search space until search was (almost) random. In that manner the search-control mechanism allowed for an admissible search [1].

The problem is that intelligence is not unique and the question that arises as a consequence of this observation is how can there be but one algorithm (e.g., neural nets, case-based reasoning, evolutionary programming, \( et \) al.), which defines it. Then, if there is indeed more than one defining algorithm, it follows from the standard insertion of vacuous Begin End; pairs that there must be a countably infinite number of them. A consequence of this result is that we necessarily change from a search for the most intelligent algorithm to a search for those algorithms (including hardware technologies), which can be most rapidly adapted to exhibit intelligent behavior in the context of a given (possibly dynamic) domain.

For example, let us constrain the definition of intelligence, for the moment, to be any algorithm that can sort numbers or words in lexicographic order. This is a relatively simple task, verifiable, and of course it has been done. However, there are \( O(n \log n) \) sorts (e.g., Quicksort), \( O(n^2) \) sorts (e.g., Insertion sort), and even \( O(n) \) sorts on parallel machines by limiting the data set size [2]. Then too, one can embed one sort algorithm in another. For example, use Quicksort until the iterative partitions become of size less than 21, then use insertion sort. Clearly, there are a very large number of possible sort programs, which are functionally appropriate.

The above description shows that we can move our search for the elusive best sort algorithm to a less pure methodology, which at the very least involves a composition of functions, or in more general terms, \( reuse \). This does not mean that the search for new sort functions must grind to a halt. Rather, the implication is quite to the contrary. That is, the search for better sort functions must be concomitant with their reuse. These processes have been characterized as being random and symmetric [3].

2. On the Complexity of Intelligence

1.1. Definition (random): A series of numbers is said to be \( random \) if the smallest algorithm capable of specifying it
to a computer has about the same number of bits of information as the series itself. A random series of digits is one whose complexity is approximately equal to its size in bits. If complexity is defined as a measure of randomness, no number can be effectively proved to be random unless the complexity of the number is less than that of the system itself [3].

1.2. Remark: It follows from the fact that all minimal programs are random that a system of greater complexity is required in order to prove that a program is a minimal one for a particular series of digits [3]. This further implies that beyond a certain complexity, every algorithmic design process must incorporate the element of chance to evolve a greater complexity.

Still, it seems counter-intuitive to say that the complexity of an ideal sort program cannot be bounded above. As it turns out, the complexity can, in all cases, be bounded above so long as the program is not capable of self-reference and in no case can it be bounded above, otherwise. These assertions require proof.

First, we prove that there is no effective enumeration of all the random computable functions [4].

1.3. Theorem (unsolvability of the randomization problem): There is no algorithm, which when presented with indices and , can decide whether is a randomization of . Thus, there is no algorithm, which when presented with the index of an arbitrary computable function , can randomize that function (i.e., transform it into , where indexes an arbitrary randomized, or random, function).

Proof: Define the total function

\[
\text{random}(i, j) = \begin{cases} 
1, & \text{if } \varphi_i \text{ randomizes } \varphi_j; \\
0, & \text{otherwise.}
\end{cases}
\]  

The randomization function is computable by definition. This function, in its simplest form, removes all redundant Begin End; pairs. Let be a generalized index for it. It should be clear that it can now be written, 

\[
\text{alg orithm}(j) = \text{random}(i, j) \quad \text{for all } j \in \mathbb{N},
\]

where is defined in equation (2).

\[
\text{alg orithm}(j) = \begin{cases} 
1, & \text{if } \varphi_i \text{ randomizes } \varphi_j; \\
0, & \text{otherwise.}
\end{cases}
\]  

The computability of random has thus been reduced to that of algorithm. Since algorithm is not computable, random cannot be either.

This proof shows that no effective method can provably minimize the space-time complexity of an arbitrary algorithm on an absolute scale. A consequence of this theorem is that all embodied representations and algorithms must be evolvable in every degree of freedom (e.g., in their hardware realizations) in the general case.

1.4. Proposition: The set \{ | : is random\} is not r.e. and, a fortiori, not recursive.

Proof: It was proven that there is no effective enumeration of all the random computable functions [4] in Theorem 1.3 (unsolvability of the randomization problem).

1.5. Proposition: Let \( f, f, \ldots, f, \ldots, i \in \mathbb{N} \), be an effective enumeration in which every \( f \) is a random computable function from \( N \) to \( N \). Then, there is a random computable function \( f : N \to N \), which does not appear in this effective enumeration.

Proof: To say that \( f, f, \ldots, f, \ldots, i \in \mathbb{N} \) is an effective enumeration of random computable functions means that there is a total computable function, \( g : N \to N \) such that \( \varphi_{i_{\varphi}} = f \).

Define \( f : N \to N \) by diagonalizing over the functions \( f, f, \ldots, f, \ldots, \); that is, let

\[
f(i) = f(i) \parallel f(i)
\]  

(3)

\( f \) is defined to be a random function because 

\[
\varphi_{i_{\varphi}} = f \parallel \varphi,
\]

where \( |\varphi_{i_{\varphi}}| \leq (|\varphi| + |\varphi|) \) as a result of the semantic randomization problem, which states:

\[
\exists k \mid \forall e \geq k > 0, \varphi_f =^e \varphi_i^{i_{\varphi}}(l) = \varphi_i^{i_{\varphi}}(l), |\varphi_f| \leq (|\varphi| + |\varphi|)
\]  

(4)

[5]. The concatenation macro, \( g \), is by appeal to Church’s Thesis. For example, the concatenation of two programs, each of which adds one to its common input variable results in a program that adds two to its common input variable. Here, \( f(i) \) is computable by the while-program:

Begin
\[
X2 := g(X1);
\]

(5)

\[
X1 := \Phi(X2, X1)
\]

End;
But, \( f \) cannot appear in the effective enumeration \( f, f, \ldots, f, \ldots \) itself, since it would differ from a randomized concatenation of itself on its own index.

**1.6. Corollary:** There can be no effective enumeration of all the random computable functions from \( N \) to \( N \). This completes the proof of Proposition 1.4 showing that the set \( \{ t \mid \varphi : N \rightarrow N \text{ is random} \} \) is not r.e.

This corollary has rather profound implications. Were the set \( \{ t \mid \varphi : N \rightarrow N \text{ is random} \} \) r.e., then one could hill-climb ever-better heuristics for solving an arbitrary computable problem. (Of course, nothing in the theory precludes the success of such an approach for an algorithm that is not capable of self-reference.) However, as it turns out, the infeasibility of evolving a general heuristic solution implies that there can be no proof of such – for otherwise the proof would serve as a recursive characterization – in direct contradiction of Corollary 1.6.

It follows that all non-trivial (i.e., self-referential) algorithms are inherently heuristic. That is, the notion of an absolute non-trivial and valid algorithm is a misnomer.

Having proven that the complexity of intelligence is not bounded (under the assumption that intelligence is defined to be capable of self-reference), it remains for us to investigate various mechanics for the successful evolution of programs subject to domain-specific constraints.

### 3. Paradigms for Successful Evolution

No matter the level of programming language, the complexity of programs that may be expressed in it are limited in practice by such factors as the skill and capabilities of the programmer(s), the need to (asynchronously) coordinate with other systems in parallel, the capability of the domain to be recursively decomposed and solved using hill-climbing techniques, and in theory by the complexity of the driving system (or System of Systems).

In particular, at least one pragmatic observation is that when coding complex algorithms, invariably there will be two types of spurious behaviors of implementation; namely, (a) the realization of unintended behaviors and (b) the absence of desired behaviors. Usually, these may be further characterized as deleterious behaviors. They are, as a rule, discovered during testing.

This leads to the concept of programming by testing. It may be explained as follows. Recall writing a program of average complexity – say a sort program without loss of generality. You are writing an outer loop and you are not sure if a program variable is to range from 1 to \( n \), or is it to \( n-1 \), or \( n+1 \)? You know that if you are off by one the sort will not operate properly. You have a choice. You can study hard and try to predict the proper value, or you can simply realize that if a choice produces the incorrect result for some test vector, then this alternative is to be eliminated with only a small chance of a double fault (i.e., it would have produced the correct result were it not for some other latent error). Indeed, by using multiple tests, which employ orthogonal (i.e., mutually random) test vectors, the chance of a double fault is brought well below that of a cognitive error in program design.

Functional programming makes it easy to design CASE tools, where the power of the processor(s) can be brought to bear in the form of a programmer’s assistant. However, this power is not limited to a single articulation point as was used for this illustration. Nor is it limited to functional programming (although this would not have been a major impediment), or even to Boolean logic.

Up until this point, we’ve tacitly assumed that \( O, O \) where \( O \) stands for the output vector produced by a given input vector and \( O \) is the estimated output vector, is zero. This is the so-called Boolean approach. However, functional programming is also amenable to min \( |O-O| \) using the 2 or \( \infty \) norms. Such programming is said to be fuzzy to the degree that these norms differ from zero. Thus, *fuzzy functional programming* (FFP) can be used to heuristically solve all manner of optimization problems (e.g., the TSP problem [6], [7]). In particular, the optimization of heuristics is of practical import.

Suppose that one were to apply this programming paradigm to an ergonomic design problem. In particular, suppose that one were to supply as input the various instruments on a dashboard such as a speedometer, a gas gauge, an indicator of engine temperature, et al. The goal here is to arrange these components on the dashboard so as to maximize user-utility while minimizing cognitive stress. The problem here pertains to how to evaluate such outputs. No mathematically precise function exists here as it does say for the output of a sort function. A general solution is to use an expert system to perform the fuzzy evaluation. Notice that this paradigm offers the designer a key advantage. Instead of focusing on how to solve the problem, one can now be liberated to focus on how to evaluate the solution of the problem. This allows one to work at a higher level and hence be more productive in terms of the quantity as well as the quality of output.

FFP can be simultaneously applied to multiple articulation points in a program where indecision is by design (as we saw for the previous example of a single articulation point involving \( n-1, n, \) and \( n+1 \)). The problem does not lie in the evaluation of the solutions; but rather, in the number of combinations of candidate solutions. A faster and/or multi-processor always helps, but consider the combinatorics. An increase in the number of processors will result in a linear speedup at best; whereas, the space of alternatives grows multiplicatively. This means that it is necessary to restrict the growth in simultaneous articulation points. We term this the *triangle inequality* (e.g., \( |A| + |B| \geq |A + B| \)).
Consider the definition of an arbitrary algorithm. If one has some knowledge as to what one variable(s) are, then that knowledge can be used to minimize the search for what remaining variable(s) might be if they are in the form of articulation points. Each such constraint can result in an exponential speedup of the synthesis process. Notice that here the constraints are not in the form of rules, but rather in the form of schema that define searchable alternatives. Higher-level programming languages can be designed to facilitate the manual and later the heuristic definition of such schema. Most importantly, such languages permit the machine to do what it does best (i.e., exercise search algorithms) and the human in the loop to do what they do best (i.e., insert search-control knowledge). Moreover, the space of alternative articulation points may consist of Functions of Functions. This requires more processor power for speed, but serves to increase the complexity of that which may be synthesized.

4. Evolution is Defined by Heuristics of Heuristics

Heuristics represent a condensation of the domain-specific knowledge embodied in (1) the search involved in finding the heuristic and (2) the knowledge that may be brought to bear for minimizing the complexity of the search paradigm. This knowledge must, by definition, include heuristics in the general case. This inclusion is necessary for a mathematical fixed point to arise, which in the case of an indefinite sequence has a random sequence as the image of its randomization. It follows that in the general case there can be no such thing as logics for discovery – only heuristic search, including heuristics for discovering heuristics, ad infinitum, until a domain-specific fixed point is reached. Indeed, it is the absence of search-control heuristics that principally led to the failure of the Japanese Fifth Generation project [8].

As supporting evidence, we next prove that heuristic acquisition is inherent to any non-trivial randomization of knowledge. Let, \( f \) be a total computable search function, which maps an arbitrary context, \( c \), to an arbitrary case antecedent, \( a \), where \( c \) and \( a \) are state vectors. Equality (=) is never a map, which is designated by an implication \( (\rightarrow) \). Then,

\[
 f(c) = \begin{cases} 
 a, & \text{if } \|f(c) - a\| < \delta; \\
 \text{NIL, otherwise.} & \end{cases}
\]

defines a qualitatively fuzzy match. Next, let \( g(a) \rightarrow b \) and \( f(c) \rightarrow h(b) \), where \( b \) is an arbitrary state vector; \( g \) and \( h \) are total computable functions; and, \( g \) may not substitute for \( h \), where it exists, because \( 0 < \delta \). Note that the use of the function \( h \) in \( h(b) \) denotes the case adaptation problem, which is as yet unsolved in the literature for the general case. We may write, \( f(c) \rightarrow h(g(a)) \), and since \( g \) and \( h \) are total computable functions, it follows that

\[
a \rightarrow \begin{cases} 
 k(a), & \text{if } \|f(c) - a\| < \delta; \\
 \bot, & \text{otherwise.}
\end{cases}
\]

where \( k \) is the total computable composition, \( h \circ g \). Now, where \( \delta \) is arbitrarily large, we may write, \( a \rightarrow k(a) \), \( a \neq k(a) \). Next, define \( F \) to be a Function of Functions such that \( F = \{f_1, f_2, \ldots, f_n\} \). It should be clear that we can write, \( F \rightarrow g(F) \), where \( g \) is a redefine total computable function. Then, through the use of pairing and projection functions, one may equivalently write, \( f \rightarrow g(f) \), \( f \neq g(f) \). Observe that, \( a \rightarrow f(a) \) and \( a \rightarrow g(f(a)) \), where \( f \) is a redefine total computable function. Thus, it follows that there must be inherent non-determinism in a general case adaptation, since \( a \) is arbitrary and \( f \neq g(f) \). It also follows that the alternatives, selected at random, are equivalent under transformation (i.e., analogous) – under the image of the arbitrary functional, \( g \).

5. On Feature Evolution

Evolutionary feature definition requires the manual specification of schema for evolutionary instantiation. At this point, we address the construction and instantiation of feature schemata. Features are based on a composition of functions, which are subject to realization on parallel/distributed processors. Each function is said to be immutable because it may embody the full range of features in the programming language used for realization; yet, once specified, it exists in a library of functions and may only be subject to manual update. Such functions may be iteratively composed using one or more pairs of nested braces, \{\}. An example will serve to clarify the design and instantiation of a schema. Consider:

---

**Fig. 1 A Generalized Schema for Feature Synthesis**

The schema shown in Fig. 1 embodies \( m \times n \) programs (candidate feature definitions). Instances are not created sequentially because the resulting search space may be intractable. Rather, the search space is uniformly covered by randomly instantiating a feature definition, which has
advantages when coordinating distributed processors. While some feature definitions will be created more than once, the capability for asynchronous execution vies in favor of this approach for scalability.

DE feature (arg)

Projection function;

\{\{FN1, FN2, \ldots, FN_m\}\};

\{FN1, FN2, \ldots, FN_n\}\};

\ldots

Statement(s)

End.

Fig. 2 A Delimited Generalized Schema for Feature Synthesis

The schema shown in Fig. 2 is identical to the one shown in Fig. 1 – except for the extra level of braces. Fig. 2 embodies \(\frac{m+n}{2}\) programs, which is much less than the \(mn\) programs shown in Fig. 1. This follows because there are two inner braces and half the time the first will be selected and half the time the second will be. This one change has reduced the search space by an order of magnitude.

DE feature (arg)

Projection function;

\{\{FNI, FNj, Fnk\}\};

\{FNI, FNj, Fnk\};

\{FNI, FNj, Fnk\}\};

\ldots

Statement(s)

End.

Fig. 3 A Delimited Optimal Schema for Feature Synthesis

More generally, it can be proven that the optimal schema occurs where the number of function groups and functions are all equal (e.g., three in Fig. 3). The size of the generalized program space in Fig. 1 is \(\prod_{i=1}^{s} |f_i|\), where there are \(s\) sets or groupings of functions. However, the size of the generalized program space in Fig. 2 is \(\frac{1}{s} \sum_{i=1}^{s} |f_i|\); whereas, the size of the optimal program space in Fig. 3 is \(\lfloor \sqrt{|F|} \rfloor\). It should be clear that \(\lfloor \sqrt{|F|} \rfloor \ll \frac{1}{s} \sum_{i=1}^{s} |f_i|\) \ll \prod_{i=1}^{s} |f_i|\).

The triangle inequality tells us that simply put, the less that is left to chance the more efficient the model will be at producing viable features. In other words, it is better to have several program schemata with few articulation points each than to place all of the articulation points in one schema, where the requisite search time can easily overwhelm any computer. A key concept in writing the features is that instead of the user searching for the correct code construct or function at various points in the program, the user specifies a space of alternative constructs at a limited number of articulation points over a maximal number of schemata. In any case, the user specifies reasonable alternatives, which are captured in the form of a set – possibly tagged with a pneumonic id. Here, the user need not contemplate the details – details that would detract from his/her capability to be an efficient programmer/debugger. The computer will find for programs that satisfy all of the test vectors, if possible using a Boolean and/or fuzzy paradigm. Again, it is the schema that implicitly contains the knowledge, which delimits the search space.

6. On the Use of \(k\)-limited Transformation

The inherent need for heuristic evolution has been made clear. In this section, we advance a practical application of such need in the form of production systems. Here, a context is hill-climbed for a solution to a problem in the form of one or more (iteratively) matched rules. The difficulty arises where hill-climbing is too specific to capture a viable solution and exhaustive search is too general a methodology to be tractable. It is likely that an ordered heuristic search may be evolved, similar if not identical to that defined below, which maintains tractability while utilizing generalized hill climbing to search for a solution.

Attempts to synthesize a match for the context are \(k\)-limited. This means that at most \(k\) transformations of the context are permitted at each step, which is defined by the (transformed) context matching at least one more rule antecedent predicate than was the case for the previous step. In other words, we iteratively hill-climb matching the context, where the roads are not very steep and can go up and down like a roller coaster, but must strictly increase in elevation after each step, which consists of \(k\) transformations. Feature set to feature set (FF) maps have as their goal, mapping a context to a rule antecedent. Feature set to procedural sequence (FP) maps have as their goal, mapping a rule antecedent to a rule consequent. The transformed context is evaluated for a match against the FF-base after each transformation (but not for more than \(k\) successive transformations). An associative memory may be used to speedup the matching process. If ever the number of predicates covered by an iteratively transformed context increases by at least one, then reset the count of \(k\) to zero and the current maximum of matched antecedent predicates to the number just matched.
If \( k \) successive transformations do not yield at least one more matched rule antecedent predicate than the current maximum (i.e., a more-specific match), backtrack to the state vector for the previous step. If backtracking takes us back to the context (start node), ask the user if they would like to supply a rule consequent or stop and if neither, double the value of \( k \) and repeat the search process with provision for an interrupt exit. Conversely, if the context is mapped to cover at least one rule antecedent, halve the value of \( k \) and optionally repeat the search process. Rules are thus fired and corrected in accordance with the non-deterministic algorithm just described.

Each successive transformation of the context is stacked. If ever a stacked state vector is repeated, the transformation leads to a cycle so return to the node associated with that state vector (deleting the intervening state vectors) and reset the count of \( k \) to zero. There will be exactly one such node.

More generally, if ever a stacked state vector is a superset of a previous state vector on the stack, the transformation leads to right recursion (e.g., \( f_i \rightarrow f_j f_k \)) so return to the node associated with the lowest (first) subset state vector (deleting the intervening state vectors) and reset the count of \( k \) to zero. There may be more than one such node. It would not be practical to use associative memory here.

Whenever an (iterative) transformation of a context covers and thus leads to the firing of a rule, one or more FF maps are acquired from the stacked state vectors. For example, if the acyclic stack appeared from bottom to top, \( \langle F_i, F_j, F_k \rangle \), where \( F_i \) is the context, \( F_j \) may or may not cover some rule antecedent, and \( F_k \) must cover at least one rule antecedent, then two non-right-recursive FF maps are non-redundantly acquired as follows.

1. \( F_i \rightarrow F_j \)
2. \( F_j \rightarrow F_k \)

One or more than two such maps may similarly be non-redundantly acquired. The granularity [9] of the FF maps will decrease with use over time as new FF maps are acquired until some domain-specific fixed point is attained. Each such FF map (including finer-grained replacements) serves as a heuristic when ordered (see below) and given sufficient processing power. Thus, even \( k \)-limited Type 0 transformational search can be sped up, where symmetric constructs such as the temporal locality of transformation can be extracted.

**7. On Ordered Search**

Ordered search is perhaps somewhat paradoxically based on random search in view of the Type 0 generality of the FF maps. Such a search paradigm distributes well over fine-grained parallel processors because there is no need to coordinate data structures. Nevertheless, there is a need to heuristically order the FF maps according to their probability in successfully mapping the current state vector to a maximal coverage of an antecedent(s).

First, the FP-base is maintained by inserting new FP maps at the head of the base and moving properly fired cases (FP maps) to the head of its base. FP maps can only be determined to be improperly fired if no FF maps were involved – in which case they are expunged from the base.

Second, the FF-base is maintained by inserting new FF maps at the head of the base and moving all properly fired FF maps (as determined by the validity of the resultant FP map(s)) to the head of its base. A sequence of FF maps are said to have been improperly fired if the resultant FP map is judged to be erroneous. In this case, if the sequence consists of exactly one FF map, it is clearly erroneous and thus expunged. If however the sequence consists of exactly two distinct FF maps (a single FF map will never apply twice in immediate sequence), then either one or both FF maps will be in error. Since this situation cannot be resolved, both involved FF maps are moved to the bottom of the FF base. Note that these maps are first to fall off the cache upon the acquisition of new FF maps when the base attains it limiting size. If however the sequence consists of more than two distinct FF maps, then similarly each involved FF map is moved down in proportion to the number of involved FF maps. This is because the more maps in sequence, the less likely any one of them will be in error. Let the sequence be of length \( m \) and the FF base have a limiting size of \( n \), where \( m \geq 2 \).

Move the involved FF maps down by \( \left\lfloor \frac{n-m+2}{m} \right\rfloor \) positions, where the floor function is interpreted to mean, “Or insert at the end of the list – whichever comes first”.

The FP-base is logically ordered to maintain the most-recently-used (MRU) cases in cache and allow the least-recently-used (LRU) cases to fall off the bottom of the cache, where they may fall off (i.e., be at least temporarily forgotten). Forgotten FF and FP maps can often be synthesized again through the application of other resident maps, which are more-recently-used. The FF-base has its maps visited in probabilistic order, where the MRU maps are more frequently visited than the LRU ones. These probabilities are assigned using the 3-2-1 skew.

**8. The 3-2-1 Skew**

This assignment of probabilities favors the logical head of the FF-base in keeping with Denning’s principle of temporal locality [10]. That is, FF maps, which were most recently acquired or fired (and thus appear at the head of the FF-base), are proportionately more likely to be fired under the 3-2-1 skew. If multiple processors are employed, each processor independently realizes this
skew for its local memory. Such distributed processing also serves to insure tractability. It similarly insures that the following 3-2-1 skew is subdivided into relatively short blocks so that all FF maps remain relatively accessible in practice.

The closer an FF map is to the top of its linked list, the greater the likelihood of its selection. A good scheme (i.e., the 3-2-1 skew) for achieving this with an application domain base of \( F \) maps is to assign the head FF map a probability of being selected of \( \frac{2F}{F(F + 1)} \). The map just below the head map has a probability of being selected of \( \frac{2(F - 1)}{F(F + 1)} \). Finally, the tail map of the FF-base has a probability of being selected of \( \frac{2}{F(F + 1)} \). The \( i \)th map from the head has a probability of being selected of \( \frac{2(F - i + 1)}{F(F + 1)} \), for \( i = 1, 2, \ldots, F \).

A most-efficient algorithm for realizing the 3-2-1 skew is given in Fig. 4. This algorithm has the added advantage of favoring just the relative head of the FF-base during very short execution times, which is proportionately most likely to yield a successful FF transformation of the context. Otherwise, statistically switching candidates for (further) mapping serves to offset the possibility of deleterious hill-climbing in the mapping process.

```plaintext
i = 1;
Repeat
    Select an FF map as a candidate transform using a uniform random number generator (e.g., Mersenne Twister, LispWorks mt-random) with integers in \([1, i]\);
    If \( i < \) current number of FF maps in the processor base, \( i \leftarrow i + 1 \)
Until
    Interrupt or Wake-Up;
```

Fig. 4 An Efficient Algorithm for Realizing the 3-2-1 Skew — the Distribution is Skewed in Favor of the Most-Recently Acquired and/or Fired FF Maps

9. Conclusions

By establishing that randomization is inherently heuristic, it follows that the best heuristics for randomization cannot be effectively enumerated. That is to say that the evolution of heuristics in general inherently depends on heuristics, which necessarily include chance as their fixed-point case. The evolution of heuristics must also depend on symmetry and self-reference (also heuristics) for tractability. Thus, while the complexity of intelligence can not be numerically bounded above, it is strongly tied to the representational formalisms employed and to those heuristics, which can emerge as a consequence of those formalisms.

Acknowledgements

The author thanks the Space and Naval Warfare Systems Center, San Diego, California for financial support. This work was produced by a U.S. government employee as part of his official duties and no copyright subsists therein. It is approved for public release with an unlimited distribution.

References