International Journal of Computer Mathematics
Vol. 00, No. 00, Month 200x, 1–18

RESEARCH ARTICLE

Mutation systems*

Dana Angluin[a] and James Aspnes and Raonne Barbosa Vargas

Computer Science Department, Yale University

(*Received 00 Month 200x; in final form 00 Month 200x*)

We propose *mutation systems* as a model of the evolution of a string subject to the effects of mutations and a fitness function. One fundamental question about such a system is whether knowing the rules for mutations and fitness, we can predict whether it is possible for one string to evolve into another. To explore this issue we define a specific kind of mutation system with point mutations and a fitness function based on conserved strongly k-testable string patterns. We show that for any k greater than 1, such systems can simulate computation by both finite state machines and asynchronous cellular automata. The cellular automaton simulation shows that in this framework, universal computation is possible and the question of whether one string can evolve into another is undecidable. We also analyze the efficiency of the finite state machine simulation assuming random point mutations.

Keywords: mutation systems; finite automata; cellular automata; k-testable languages; point mutations; reversible computation

1. Introduction

Biological evolution proceeds by variation and selection. Efforts to determine the evolutionary relationships of different organisms often involve comparing the DNA sequences of their genomes to find similar subsequences that have been conserved during evolution, on the assumption that the conserved subsequences affect the fitness of the organisms. In this work we propose **mutation systems** as a general model of variation and selection acting on strings of symbols. Our goal is to explore the properties of such systems, specifically what can be predicted and learned about their behavior. Variation is modeled as a mutation function that maps a string to the set of possible mutations of that string. Selection is modeled as a fitness function that determines whether each string is fit or not. The main relation we consider in this paper is whether one fit string can evolve to another fit string through a sequence of fit strings, each of which is a possible mutation of its predecessor.

As one example of how small and locally conserved patterns in genome sequences may play an important role in biological processes, we briefly describe transcription factor binding sites. Alberts et al. [1] discuss evidence that evolutionary processes have shaped the mechanisms controlling the selective expression of genes in each cell. Proteins called transcription factors bind to the DNA sequence in specific positions called transcription factor binding sites, and regulate the expression of that gene by activating or inhibiting the transcription process. These functional binding sites are called regulatory elements, and are usually small parts of the DNA sequence situated near the location where transcription of the gene sequence

*This material is based upon work supported by the National Science Foundation under Grant Number CCF-0916389.
[a]Corresponding author.

ISSN: 0020-7160 print/ISSN 1029-0265 online
© 200x Taylor & Francis
DOI: 10.1080/0020716YYxxxxxxxx
http://www.informaworld.com

starts. Estimates of the typical lengths of these regulatory elements include 10 base pairs (by Tompa et al. [17, 18]) and a range of 5 to 12 base pairs (by Wasserman and Sandelin [21].) The concept of Phylogenetic Footprinting [16] suggests that these regulatory elements tend to be conserved in the genome sequence throughout evolution, because of their functional importance in the control of gene expression. Thus cross-species comparisons may help to discover transcription factor binding sites.

In introducing the model of mutation systems, our main focus is on what an outside observer can predict or learn about an evolutionary process, not on the capacity of the process itself to learn or compute. Our model does not have a final ideal target or answer, but instead has a variety of evolution pathways and outcomes defined by the mutation operator and the fitness function. Though we describe the construction of particular mutation systems capable of simulating finite state machines or cellular automata, our primary purpose is to understand the inherent limitations of an observer in regard to a mutation system, rather than to propose practical methods of computation.

Cavaliere and Leupold[6, 7] have introduced the model of *computing by observing* in which one system (the observed) evolves through a sequence of configurations (for example, the successive sentential forms of a derivation of a terminal string in a context-free grammar), and a separate system (the observer) generates an output (a single symbol or the empty string) after processing each configuration. This setting includes the observer as a component of the model, and focuses on the sequences of incrementally produced observations. The fundamental results in this area are aimed at characterizing what classes of languages can be generated using very simple formalisms to model the observed and observer.

Valiant [19] has introduced the model of *evolvability* to explore the question of what functions can be efficiently approximated through a polynomial-time evolution process. In this case, fitness or performance is measured by approximation to a single ideal target function, and the class of evolvable functions is contained in the class of PAC-learnable functions. The motivation is to study the capabilities and limitations of evolution.

In contrast these approaches, the large and flourishing area of Evolutionary Computation[8] is concerned with understanding and using evolutionary mechanisms to design algorithms to optimize functions and solve other computational problems. In this setting, the algorithm designer chooses the problem representation, the mutation and recombination operations, the fitness function, the details and parameters of how the population evolves and various other aspects of the model in order to solve a particular computational problem effectively and efficiently.

After a short section of preliminaries, we introduce our new model of a mutation system composed of an alphabet, a mutator and a fitness function in Section 3 and give several examples. We then specialize to the case of point mutations (in Section 3.1) and fitness functions defined by strictly k-testable patterns (in Section 3.3.) The combination of point mutations and a strictly k-testable fitness function is termed a k-simple mutation system. In Section 3.4 we introduce the technique of symbol duplication to help control the effects of point mutations. In Section 4 we show that for any nondeterministic finite state machine, there is a 2-simple mutation system that simulates its computations. We consider the overhead of this simulation for deterministic finite state machines under the assumption of random point mutations in Section 4.3. In Section 5 we show that for any one-dimensional asynchronous reversible cellular automata with insertions and deletions, there is a 2-simple mutation that simulates its computations, thus showing that 2-simple mutation systems are a universal model of computation. In Section 6 we give conclusions and discuss possible future directions.

2. Preliminaries

An alphabet Σ is a finite nonempty set of symbols. Σ^* denotes the set of all finite strings of symbols from Σ. The empty string is denoted λ. A string v is a substring (or factor) of a string x if there exist strings u and w such that $x = uvw$. A language is any subset of Σ^*. Σ^k denotes those elements of Σ^* of length k. The symbols in a string s of length n are indexed from 1 to n and $s[i]$ denotes the i^{th} symbol of s.

We consider non-deterministic finite state machines with no accepting states, defined as follows. A finite state machine (FSM) is a quadruple $M = (\Sigma, Q, q_0, \delta)$, where Σ is the alphabet of input symbols, Q is the set of states, q_0 is the initial state, and δ is the transition function, which maps $Q \times \Sigma$ to subsets of Q. If every $\delta(q, a)$ contains exactly one state, then M is deterministic. In this case we may write $\delta(q, a) = q'$ instead of $\delta(q, a) = \{q'\}$.

3. Mutation systems

We propose a model of the evolution of a string subject to the effects of mutations and a fitness function. A single step consists of a mutation of the current string followed by an application of the fitness function. If the fitness function determines that the mutated string is fit, the mutated string replaces the current string; otherwise the mutated string is discarded and the current string is kept.

Definition 3.1 A **mutation system** $S = (\Sigma, \mu, f)$ is composed of an alphabet Σ, a mutator μ that maps Σ^* to subsets of Σ^* and a fitness function $f : \Sigma^* \to \{0, 1\}$. The mutator μ specifies the set of strings to which a given string can mutate in one step. The fitness function f determines whether a given string s is fit ($f(s) = 1$) or not ($f(s) = 0$).

Our model permits the separation of the effects of the mutator μ and the fitness function f. For example, in some situations we may be observing a system with a known mutator and an unknown fitness function.

Given a mutation system S and two fit strings s_1 and s_2, we are interested in the question of whether s_1 can evolve to s_2 through a sequence of steps permitted by S.

Definition 3.2 Let a mutation system $S = (\Sigma, \mu, f)$ and two strings $s_1, s_2 \in \Sigma^*$ be given. We say that s_1 **can mutate to** s_2 **in one step**, denoted $s_1 \to_\mu s_2$, if $s_2 \in \mu(s_1)$. We say that s_1 **can evolve to** s_2 **in one step**, denoted $s_1 \to_S s_2$, if $f(s_1) = f(s_2) = 1$ and s_1 can mutate to s_2 in one step.

As is usual, we denote the reflexive transitive closure of these relations by a superscripted * on the arrow. We say that s_1 **can mutate to** s_2 if $s_1 \to_\mu^* s_2$, that is, there is a finite sequence of zero or more mutation steps that carries s_1 to s_2. Similarly, we say that s_1 **can evolve to** s_2 if $s_1 \to_S^* s_2$, that is, there is a finite sequence of zero or more evolution steps that carries s_1 to s_2. Note that in the latter case, s_1, s_2 and any intermediate strings in some evolution must be fit.

Example 3.3 Consider a mutation system $S_1 = (\Sigma_1, \mu_1, f_1)$ over the alphabet $\Sigma_1 = \{a, b, c\}$ where $\mu_1(s)$ is the set of strings obtained by interchanging two adjacent symbols in s and $f_1(s) = 1$ if and only if no two adjacent symbols of s are equal.

Then we have the following mutation steps.

$$abcbc \rightarrow_{\mu_1} abccb \rightarrow_{\mu_1} acbcb \rightarrow_{\mu_1} cabcb.$$

However these are not all evolution steps because the string *abccb* is not fit according to f_1. An alternative path of evolution steps demonstrates that *abcbc* can evolve to *cabcb*.

$$abcbc \rightarrow_{S_1} bacbc \rightarrow_{S_1} bcabc \rightarrow_{S_1} bcacb \rightarrow_{S_1} cbacb \rightarrow_{S_1} cabcb.$$

Example 3.4 Consider the mutation system $S_2 = (\Sigma_2, \mu_2, f_2)$ with alphabet $\Sigma_2 = \{a, b, c\}$, defined as follows. For any string s, $\mu_2(s)$ is the set of strings that may be obtained from s by one of the following operations: replace a contiguous substring of a's by an equal number of b's, or replace a contiguous substring of b's by an equal number of c's, or replace a contiguous substring of c's by an equal number of a's. The following is a sequence of one step mutations in this system.

$$aacca \rightarrow_{\mu_2} bbcca \rightarrow_{\mu_2} bbaaa \rightarrow_{\mu_2} bbaab.$$

Define $f_2(s) = 1$ if and only if s does not contain occurrences of all three symbols: a, b and c. The initial element of the above sequence (*aacca*) is fit according to f_2, but the second element (*bbcca*) is not. The following alternative sequence of mutations shows that *aacca* can evolve to *bbaab* in S_2.

$$aacca \rightarrow_{S_2} aaaaa \rightarrow_{S_2} bbaaa \rightarrow_{S_2} bbaab.$$

In fact, in S_2 any fit string s of length n can evolve to any other fit string of length n as follows. If s contains two different symbols x and y such that $x \rightarrow_{\mu_2} y$, then blocks of x's can be converted to blocks of y's until the string is of the form y^n. This can be converted to z^n for any $z \in \{a, b, c\}$ in at most two more mutation steps. To produce a fit string with occurrences of different symbols x and y such that $x \rightarrow_{\mu_2} y$, first produce x^n, and then covert blocks of x's to y's to achieve the desired target string.

It may be helpful to visualize the mutation and evolution relations as directed graphs. The **mutation graph** of a mutation system is the graph with vertices Σ^* and directed edges (s_1, s_2) such that s_1 can mutate to s_2 in one step. The **evolvability graph** of a mutation system is the subgraph of the mutation graph induced by the set of fit strings (i.e., those s with $f(s) = 1$). Thus, the evolvability graph is obtained from the mutation graph by removing all the vertices corresponding to unfit strings. The string s_1 can mutate to s_2 if and only if there is a directed path from s_1 to s_2 in the mutation graph, while s_1 can evolve to s_2 if and only if there is a directed path from s_1 to s_2 in the evolvability graph.

Example 3.5 Suppose $S_3 = (\Sigma_3, \mu_3, f_3)$ has alphabet $\Sigma_3 = \{0, 1, 2, 3\}$ and the mutator μ_3 can transform a string by adding 1 to any single symbol that is not 3. The portion of the mutation graph of S_3 restricted to strings of length two is pictured in Figure 1. Suppose the fitness function f_3 assigns 0 to any string that is a decimal representation of a prime number. Vertices corresponding to unfit strings are removed, and the evolution graph of S_3 restricted to strings of length two is pictured in Figure 2.

In what follows we focus on mutators and fitness functions defined very locally on strings.

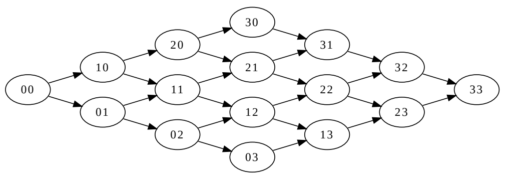

Figure 1. The mutation graph of S_3 restricted to strings of length two.

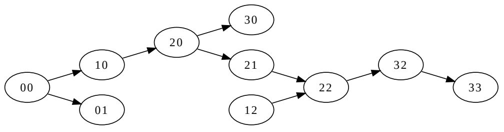

Figure 2. The evolution graph of S_3 restricted to strings of length two. Only vertices corresponding to fit strings remain.

3.1 *Point mutations*

A point mutation of a string is obtained by deleting or inserting a single occurrence of a symbol or by replacing a single occurrence of a symbol by any symbol.

Definition 3.6 Let s be any string. The mutators μ_d, μ_i, μ_r, and μ_p are defined as follows.

(1) $\mu_d(s)$ is the set of strings that can be obtained by deleting exactly one occurrence of a symbol from s.
(2) $\mu_i(s)$ is the set of strings that can be obtained from s by inserting exactly one occurrence of a symbol from Σ into s.
(3) $\mu_r(s)$ is the set of strings that can be obtained from s by replacing exactly one occurrence of a symbol in s by any symbol from Σ.
(4) $\mu_p(s) = \mu_d(s) \cup \mu_i(s) \cup \mu_r(s)$. Thus, the mutator μ_p permits any single point mutation of a string.

For example, over the alphabet $\{a, b, c\}$ we have the following.

$$\begin{aligned} \mu_d(bcb) &= \{cb, bb, bc\} \\ \mu_r(bcb) &= \{acb, ccb, bab, bbb, bca, bcc\} \\ \mu_i(bcb) &= \{abcb, bbcb, cbcb, bacb, bccb, bcab, bcbb, bcba, bcbc\} \end{aligned}$$

For comparison, the mutators μ_1 and μ_2 of Examples 3.3 and 3.4 permit mutations that are not point mutations, while the mutator μ_3 of Example 3.5 permits a subset of the single symbol replacements.

3.2 *Reversibility*

Reversibility is an important property of mutators and mutation systems.

Definition 3.7 A mutator μ is **stepwise reversible** if for all strings s_1 and s_2,

$$s_2 \in \mu(s_1) \Leftrightarrow s_1 \in \mu(s_2).$$

That is, if s_1 can mutate to s_2 in one step, then s_2 can mutate back to s_1 in one step. A mutation system $S = (\Sigma, \mu, f)$ is **reversible** if for all strings s_1 and s_2,

$$(s_1 \rightarrow_S^* s_2) \Leftrightarrow (s_2 \rightarrow_S^* s_1).$$

That is, if s_1 can evolve to s_2, then s_2 can evolve to s_1.

The point mutator μ_p is stepwise reversible: an insertion can be reversed by a deletion, a deletion by an insertion, and a replacement by the opposite replacement. The mutator μ_1 of Example 3.3 is stepwise reversible, but the mutators μ_2 and μ_3 of Examples 3.4 and 3.5 are not.

If a mutator μ is stepwise reversible then its mutation graph can be considered to be undirected, because every edge (s, t) has a corresponding reverse edge (t, s). If a mutation system is reversible then every weakly connected component of its evolvability graph is strongly connected.

Lemma 3.8 If μ is stepwise reversible then $S = (\Sigma, \mu, f)$ is reversible.

Proof Suppose $s_1 \to_S^* s_2$. Then there is a sequence $u_0, \ldots, u_t$ of fit strings such that

$$s_1 = u_0 \to_\mu u_1 \to_\mu \ldots \to_\mu u_{t-1} \to_\mu u_t = s_2.$$

Because μ is stepwise reversible, we have also

$$s_2 = u_t \to_\mu u_{t-1} \to_\mu \ldots \to u_1 \to_\mu u_0 = s_1,$$

which shows that $s_2 \to_S^* s_1$. Thus the mutation system $S = (\Sigma, \mu, f)$ is reversible. ■

The mutation system S_2 in Example 3.4 is a counterexample to the converse of this lemma. In particular, the mutator μ_2 is not stepwise reversible, but the mutation system S_2 is reversible because any fit string can evolve to any other fit string of the same length.

3.3 *Conservation of strictly k-testable patterns*

We consider fitness functions defined by very local properties of a string, namely properties characterized by strictly k-testable languages [5, 12, 15]. McNaughton [15] introduced the notion of local testability and defined strict k-testability. Brzozowski and Simon [5] gave an alternative definition that differs slightly from the original by McNaughton, and proved the two definitions equivalent. This alternative definition was also later used by McNaughton in Kim et al. [12], and it is the definition we use in this paper.

Head [10] and Yokomori and Kobayashi [22] describe applications of k-testable languages to modeling biological phenomena. Another reason to consider k-testable languages is evidence for the tractability of learning them. In particular, the k-testable languages are learnable in the limit from positive data [9] and also concatenations of k-testable languages are learnable in the limit from positive data [13].

Definition 3.9 Let Σ be an alphabet. A **strictly k-testable pattern**

$$P = (\text{PRE}, \text{MID}, \text{SUF})$$

is composed of three sets of strings with $\text{PRE} \subseteq \Sigma^{k-1}$, $\text{MID} \subseteq \Sigma^k$, and $\text{SUF} \subseteq \Sigma^{k-1}$. The language of P, denoted L_P, is the set of all strings s of length at least k such that the prefix of s of length $k-1$ is in PRE, every substring of s of length k is in MID, and the suffix of s of length $k-1$ is in SUF.

Thus PRE specifies the permissible length $k-1$ prefixes, SUF specifies the permissible length $k-1$ suffixes, and MID specifies the permissible length k substrings. Note that if P is a strictly k-testable pattern, then L_P is a regular set.

Example 3.10 Let $\Sigma = \{a, b, c\}$ and $P = (\text{PRE}, \text{MID}, \text{SUF})$ with $\text{PRE} = \{a, b, c\}$, $\text{MID} = \{ab, ac, ba, bc, ca, cb\}$ and $\text{SUF} = \{a, b, c\}$. Then P is a strictly 2-testable pattern and L_P consists of all strings of length at least 2 over Σ in which no two adjacent symbols are equal.

A fitness function f is defined to be **strictly k-testable** if there exists a strictly k-testable pattern P such that for every string s, $f(s) = 1$ iff $s \in L_P$. Note that this implies that $f(s) = 0$ for strings s of length less than k.

The fitness function f_1 of Example 3.3 is strictly 2-testable because it simply forbids two adjacent equal symbols. The fitness function f_2 of Example 3.4 is not strictly k-testable for any k because it may be necessary to see a substring of unbounded length to witness that all three symbols a, b, and c are present in a string. The fitness function f_3 of Example 3.5 is also not strictly k-testable for any k because the set of decimal representations of primes is not a regular set.

A **k-simple mutation system** is a mutation system with mutation operator μ_p and a strictly k-testable fitness function. In what follows we focus on 2-simple mutation systems.

3.4 *Symbol duplication*

The technique of symbol duplication is useful in preventing unwanted point mutations in a 2-simple mutation system. If the alphabet is Σ, then the **duplicated alphabet** $D(\Sigma)$ consists of two copies of each symbol $a \in \Sigma$, one with index 1, denoted a^1, and one with index 2, denoted a^2. We define the **duplication map** d from Σ^* to $D(\Sigma)^*$ such that $d(s)$ is obtained from s by replacing every occurrence of a symbol a in s by the string a^1a^2. We define a **projection map** h_1 from $D(\Sigma)^*$ to Σ^* such that $h_1(s)$ replaces every index 1 symbol a^1 by a and every index 2 symbol a^2 by the empty string. For example, $d(abb) = a^1a^2b^1b^2b^1b^2$ and $h_1(a^1b^1b^2a^2a^1) = aba$. Clearly $h_1(d(s)) = s$.

Example 3.11 We define a 2-simple mutation system $S_4 = (\Sigma_4, \mu_4, f_4)$ that protects strings against point mutations. Let $\Sigma = \{a, b\}$. The alphabet Σ_4 is $D(\Sigma) = \{a^1, a^2, b^1, b^2\}$ and the strictly 2-testable fitness function f_4 is defined by the prefix strings $\{a^1, b^1\}$, the suffix strings $\{a^2, b^2\}$, and the middle strings

$$\{a^1a^2, a^2a^1, a^2b^1, b^1b^2, b^2a^1, b^2b^1\}.$$

The set of strings that are fit with respect to f_4 are exactly those of the form $d(s)$ for some nonempty $s \in \Sigma^*$, for example, $a^1a^2b^1b^2b^1b^2$.

It is not difficult to show that if a fit string undergoes any non-identity point mutation, the resulting string is not fit with respect to f_4. Thus the evolution graph of S_4 consists of isolated vertices.

4. Simulating FSM computation

To represent FSM computation using a reversible mutation system, we choose a reversible representation: FSM computation histories, analogous to Bennett's construction to make Turing machines reversible [3]. Let $M = (\Sigma, Q, q_0, \delta)$ be a finite state machine. Choose an element $x \notin Q$ and define the **state-annotated alphabet** Σ_Q as the set of all symbols a_q such that $a \in \Sigma$ and $q \in Q \cup \{x\}$. The symbol a_q represents the state q of M after reading the symbol a, with x indicating that the symbol is unread. The main symbol component of a_q is a and the state component is q.

Given a string $s \in \Sigma^*$ of length n, a **computation history of M on s** is a string $s' \in (\Sigma_Q)^*$ of length n such that the string of main symbol components of s' is s, and the sequence of state components consists of $q_1, q_2, \ldots, q_i \in Q$ followed by $(n-i)$ x's for some $0 \le i \le n$, where for each $1 \le j < i$, $q_{j+1} \in \delta(q_j, s[j])$. In this case, s' represents the computation in which M has read the first i symbols of s and for each j gives the state reached after reading the j^{th} input symbol. The **initial computation history of M on** s, denoted $I_x(s)$, is obtained from s by replacing each a by a_x, signifying that all the input symbols of s are unread.

Example 4.1 Define a deterministic finite state machine $M_1 = (\{a, b\}, \{0, 1\}, 0, \delta_1)$ with transition function δ_1 given by $\delta_1(0, a) = 1$, $\delta_1(0, b) = 0$, $\delta_1(1, a) = 0$, and $\delta_1(1, b) = 1$. The state of M_1 indicates whether it has read an odd (1) or even (0) number of a's. The computation histories of M_1 on the input string $abaa$ are the following: $a_xb_xa_xa_x$, $a_1b_xa_xa_x$, $a_1b_1a_xa_x$, $a_1b_1a_0a_x$, $a_1b_1a_0a_1$.

4.1 *From FSMs to mutation systems*

Given a FSM $M = (\Sigma, Q, q_0, \delta)$, we describe how to construct a 2-simple mutation system $S = (\Sigma', \mu_p, f)$ such that for any non-empty input string s for M, the computation histories of M on input s are represented by the strings that $d(I_x(s))$ may evolve to in S. The alphabet Σ' is $D(\Sigma_Q)$. In the symbol a_q^i, the main symbol component is a, the state component is q and the index is i. For the example FSM M_1,

$$\Sigma' = \{a_x^1, a_x^2, a_0^1, a_0^2, a_1^1, a_1^2, b_x^1, b_x^2, b_0^1, b_0^2, b_1^1, b_1^2\}.$$

Corresponding to the initial computation history $I_x(s)$ of M on input s is the initial string $d(I_x(s))$ with every symbol replaced by its duplicates indexed 1 and 2. For the FSM M_1, we have

$$d(I_x(abaa)) = a_x^1a_x^2b_x^1b_x^2a_x^1a_x^2a_x^1a_x^2.$$

We define a strictly 2-testable fitness function f that allows the initial string $d(I_x(s))$ to evolve under point mutations to just those strings representing the computation histories of M on input s. The goal is to permit those mutations of symbols indexed 1 that represent reading the corresponding symbol of the input and correctly updating the state component, and also those mutations that copy this update to the immediately following symbol indexed 2.

The strictly 2-testable pattern $P = (\text{PRE}, \text{MID}, \text{SUF})$ that determines the fitness function f is defined as follows. The set PRE contains all symbols of the form a_x^1 and a_q^1 such that $a \in \Sigma$ and $q \in \delta(q_0, a)$. The set SUF contains all symbols of the form a_x^2 and a_q^2 such that $a \in \Sigma$ and $q \in Q$. The set MID contains several types of strings of length 2, as follows.

(1) Initial duplicate: $a_x^1a_x^2$ for all $a \in \Sigma$.
(2) Initial boundary: $a_x^2b_x^1$ for all $a, b \in \Sigma$.
(3) Duplicate update needed: $a_q^1a_x^2$ for all $a \in \Sigma$ and $q \in Q$.
(4) Updated duplicate: $a_q^1a_q^2$ for all $a \in \Sigma$ and $q \in Q$.
(5) State transition needed: $a_q^2b_x^1$ for all $a, b \in \Sigma$ and $q \in Q$.
(6) State transition made: $a_q^2b_{q'}^1$ for all $a, b \in \Sigma$, $q \in Q$, and $q' \in \delta(q, b)$.

For example, the sequence of steps of M_1 on input $abaa$ can be achieved by the mutation steps shown in Figure 3. Each FSM step requires two mutations.

$$\begin{array}{cccccccc}
a_x^1 & a_x^2 & b_x^1 & b_x^2 & a_x^1 & a_x^2 & a_x^1 & a_x^2 \\
a_1^1 & a_x^2 & b_x^1 & b_x^2 & a_x^1 & a_x^2 & a_x^1 & a_x^2 \\
a_1^1 & a_1^2 & b_x^1 & b_x^2 & a_x^1 & a_x^2 & a_x^1 & a_x^2 \\
a_1^1 & a_1^2 & b_1^1 & b_x^2 & a_x^1 & a_x^2 & a_x^1 & a_x^2 \\
a_1^1 & a_1^2 & b_1^1 & b_1^2 & a_x^1 & a_x^2 & a_x^1 & a_x^2 \\
a_1^1 & a_1^2 & b_1^1 & b_1^2 & a_0^1 & a_x^2 & a_x^1 & a_x^2 \\
a_1^1 & a_1^2 & b_1^1 & b_1^2 & a_0^1 & a_0^2 & a_x^1 & a_x^2 \\
a_1^1 & a_1^2 & b_1^1 & b_1^2 & a_0^1 & a_0^2 & a_1^1 & a_x^2 \\
a_1^1 & a_1^2 & b_1^1 & b_1^2 & a_0^1 & a_0^2 & a_1^1 & a_1^2
\end{array}$$

Figure 3. Sequence of mutations for computation of M_1 on input *abaa*.

4.2 *Correctness of the FSM simulation*

To see that S correctly represents computation by M, we establish certain properties of evolvability in S. Note that for every $a, b \in \Sigma$, $a_x^1 \in$ PRE, $a_x^2 \in$ SUF, $a_x^1 a_x^2 \in$ MID and $a_x^2 b_x^1 \in$ MID, and therefore for every nonempty $s \in \Sigma^*$ we have $f(d(I_x(s))) = 1$, that is, $d(I_x(s))$ is fit in S. The following lemmas prove Theorem 4.5.

Lemma 4.2 Let $s' \in (\Sigma')^*$ be any nonempty string such that $f(s') = 1$. Then s' has the following properties.

(1) The indices of s' alternate between 1 and 2, beginning with 1 and ending with 2.
(2) If two consecutive symbols of s' are indexed 1 and 2, they must be $a_x^1 a_x^2$ or $a_q^1 a_x^2$ or $a_q^1 a_q^2$ for some $a \in \Sigma$ and $q \in Q$.
(3) If two consecutive symbols of s' are indexed 2 and 1, they must be $a_x^2 b_x^1$ or $a_q^2 b_x^1$ or $a_q^2 b_{q'}^1$ for some $a, b \in \Sigma$ and $q, q' \in Q$ such that $q' \in \delta(q, b)$.
(4) The state components of s' consist of a sequence of elements of Q followed by a sequence of x's.
(5) The string $h_1(s')$ is a computation history of M on the input s composed of the sequence of main symbol components of $h_1(s')$.

Proof PRE contains only symbols with index 1 and SUF contains only symbols with index 2, therefore s' must start with an index 1 symbol and end with an index 2 symbol. MID contains pairs of symbols with index 1 and then 2 or index 2 and then 1. Thus the indices of symbols in s' must strictly alternate: $1, 2, 1, 2, \ldots$.

Properties (2) and (3) are immediate from the definition of MID and imply property (4).

In the string $h_1(s')$ suppose positions i and $i+1$ have state components q_i and q_{i+1} in Q and main symbol components a_i and a_{i+1}. Then $s'[2i-1]$ has index 1, main symbol component a_i and state component q_i, and $s'[2i+1]$ has index 1, main symbol component a_{i+1} and state component q_{i+1}. The symbol between them, $s'[2i]$, must have index 2, main symbol component a_i and state component q_i. This is because it must have the same main symbol component as the symbol to its left in s', and either the same state component or x. But x is not possible, because the symbol to its right has a non-x state component. Thus by case (6) of MID, we must have $q_{i+1} \in \delta(q_i, a_{i+1})$. Therefore $h_1(s')$ is a computation history of M on input s. ■

Lemma 4.3 Let s be a nonempty input string for M. Let s' be any string evolvable from $d(I_x(s))$ in S. Then $h_1(s')$ is a computation history of M on input s.

Proof By Lemma 4.2, we know that $h_1(s')$ is a computation history of M on the input string that consists of the concatenation of the main symbol components of $h_1(s')$, but we need to show that this string is indeed s. This is easily seen by induction on number of the evolution steps from $d(I_x(s))$, because $h_1(d(I_x(s))) = I_x(s)$ and mutations that insert or delete a symbol, or change the main symbol component of any symbol will be rejected by f, so every string evolvable from $d(I_x(s))$ preserves the input string. ■

Lemma 4.4 Let s be a nonempty input string for M. If t is any computation history of M on input s, then $d(t)$ is evolvable in S from $d(I_x(s))$.

Proof Let $s = a_1a_2\cdots a_n$. Consider the computation history $t = (a_1)_{q_1}(a_2)_{q_2}\cdots(a_n)_{q_n}$. Let r denote the number of symbols read in t, that is, the maximum i such that $q_i \neq x$. The proof is by induction on r.

If $r = 0$ then t is the initial computation history of M on input s, that is, $t = I_x(s)$. Because $d(I_x(s))$ is fit, $d(t)$ is evolvable from $d(I_x(s))$.

If $r > 0$, then consider the computation history t' of M on input s in which the symbol $(a_r)_{q_r}$ at position r is replaced with $(a_r)_x$. In t' there are only $r-1$ symbols read, so by induction, $d(t')$ is evolvable in S from $d(I_x(s))$. There is one computation step of M from t' to t which consists of reading a_r and changing the state to $q_r \in \delta(q_{r-1}, a_r)$. The effect of this step can be accomplished by two point mutations to $d(t')$, namely, to replace the symbol $(a_r)^1_x$ at position $2r-1$ by the symbol $(a_r)^1_{q_r}$, which is accepted by f, and then to replace the symbols $(a_r)^2_x$ at position $2r$ by the symbol $(a_r)^2_{q_r}$, which is also accepted by f. Thus $d(t)$ is evolvable in S from $d(I_x(s))$. ■

The following Theorem is an immediate consequence of Lemmas 4.3 and 4.4.

Theorem 4.5 Let a finite state machine $M = (\Sigma, Q, q_0, \delta)$ be given, and let $S = (\Sigma', \mu_p, f)$ be the 2-simple mutation system constructed from M according to the method described in Section 4.1. Let $s \in \Sigma^*$ be a nonempty input string for M. For every string s' evolvable in S from $d(I_x(s))$, $h_1(s')$ is a computation history of M on input s. For every computation history t of M on input s, $d(t)$ is evolvable from $d(I_x(s))$.

If the FSM M is nondeterministic, the strings evolvable from $d(I_x(s))$ in the mutation system S correspond to all possible computation histories of M on input s because S is a reversible mutation system and may evolve backward to the initial string from any string it reaches, and then forward again along another computation path. If M is deterministic, the strings evolvable from $d(I_x(s))$ form a line graph of $2n$ vertices, with $d(I_x(s))$ at one end and the final history, in which all symbols have been read and have state components in Q, at the other end.

4.3 *Random point mutations and simulation running time*

To quantify the overhead of the FSM simulation, we consider the case of a deterministic FSM M and random point mutations. On an input of length n, the machine M reads one symbol per step, for a total of n steps before it reaches the final configuration.

In a **random point mutation** of a string s, the type of mutation is selected according to probabilities p_d (for a deletion), p_i (for an insertion) and p_r (for a replacement) where each probability is nonzero and their sum is 1. Then, depending on the type of mutation selected, a position in the string is selected equiprobably (with $|s|$ possible positions for a deletion or replacement and $|s|+1$ for an insertion.) Finally, for a replacement or insertion, a symbol is selected equiprobably from the alphabet of the mutation system to be the replacing symbol or the symbol to insert. Once the mutation has been selected and applied to produce a mutated string s', the fitness function is applied to s'. If s' is fit, it replaces s (a successful mutation), and if s' is not fit, s remains the string to be mutated (an unsuccessful mutation.) We are interested in the expected total number of mutations (successful and unsuccessful) for the mutation system simulating M on input s to evolve from $d(I_x(s))$ to strings representing every computation history of M on s.

In the simulation described in Section 4.1, when the input string s has length n we can consider the strings reachable from $d(I_x(s))$ as a Markov chain of $2n$ states, one for each reachable string, with initial state $d(I_x(s))$ and final state with no unread symbols, representing the final configuration of M on input s. Each other state of the chain has one predecessor and one successor, with a forward transition when a successful mutation causes another symbol to have state component $q \in Q$ and a backward transition when a successful mutation causes another symbol to have state component x.

The probabilities of a forward transition and a backward transition from a state of the chain that is not initial or final are each equal to

$$p_r/(2n|\Sigma'|),$$

where p_r is the probability of choosing a mutation of type replacement, n is the length of the input to M, and Σ' is the alphabet of the simulating mutation system. In order to be successful, a mutation must be of type replacement (which has probability p_r) and choose a replacing symbol that leads to a fit string. For a forward transition, the mutation must choose the first symbol with state component x and the unique element of Σ' that has the same main symbol, the same index, and correct state component in Q, which happens with probability $1/(2n|\Sigma'|)$. For a backward transition, the mutation must choose the last symbol with a non-x state component and the unique element of Σ' that has the same main symbol, the same index, and state component x, which happens with probability $1/(2n|\Sigma'|)$.

By standard results on random walks, this implies that the expected total number of mutations for the simulation described in Section 4.1 to evolve from $d(I_x(s))$ to all $2n$ reachable strings is

$$\Theta(|\Sigma'|n^3/p_r).$$

Intuitively, this bound can be viewed as the $\Theta(n^2)$ expected number of steps for an unbiased random walk to move a distance of n from the initial location, multiplied by the waiting time $\Theta(n|\Sigma'|/p_r)$ for a mutation that succeeds and makes a forward or backward transition.

4.4 *Modified simulation running time*

By using a somewhat different construction for the simulation, we can bias the walk in the forward direction, making forward transitions much more likely than backward ones. This is a technique used by Bennett [4] to improve the speed of reversible computation. In the modified simulation we make an additional copy of every symbol a^i_q such that $q \in Q$, and treat the copies as equivalent in the simulation. Note that there is still only one copy of each symbol a^i_x.

In particular, h_1 maps both copies of a^1_q to a_q and both copies of a^2_q to the empty string for each $q \in Q$. Thus, there are 2^r different strings reachable from $d(I_x(s))$ in which the first r symbols have non-x state components and the rest have state component x, and they all represent just one computation history of M on input s. We can consider the $2^{2n+1} - 1$ strings reachable from $d(I_x(s))$ as a Markov chain with $2^{2n+1} - 1$ states, with each state that is not initial having one predecessor (in which the last symbol with non-x state component is replaced by the symbol with same main symbol component, same index, and state component x) and each state that is non-final having two successors (in which the first symbol with state component x is replaced by one of the two symbols with same main symbol component, same index and the correct state component from Q.)

The result is that the probability of a forward transition is twice that of a backward transition in this Markov chain. Therefore the expected total number of mutations for the simulation to evolve from $d(I_x(s))$ to some string in which no symbol has state component x (representing the final configuration of M on input s) is reduced by a factor of n to

$$\Theta(|\Sigma'|n^2/p_r).$$

Note that we do not require the simulation to evolve to every reachable string (an exponential number), just to strings representing every computation history of M on input s. Intuitively, this construction spends random bits (in the choice of which copy of a_q^i to use) in order to make the choice of a backward transition much less likely.

5. Simulating cellular automata

Cellular automata are a well known model of computation introduced by Von Neumann [20], motivated by physical and biological problems. In a recent survey paper, Kari [11] notes that cellular automata have several fundamental properties of the physical world: they are massively parallel, homogeneous, and reversible, have only local interactions, and facilitate formulation of conservation laws based on local update rules. These properties match well with the features of our mutation system model, and a detailed comparison sheds light on the power and expressiveness of our new model. We choose to simulate one-dimensional asynchronous reversible cellular automata with insertions and deletions because they share features with mutation systems and support universal computation [14].

A **cellular automaton** $C = (\Sigma, \delta)$ is composed of an alphabet of symbols Σ and a set δ transition rules of the form $axb \leftrightarrow ayb$ for substitutions or $ab \leftrightarrow axb$ for insertions and deletions, where $a, b, x, y \in \Sigma$. The idea is that the value of a given cell of the automaton may change only when both its neighbors have specific values.

For $s_1, s_2 \in \Sigma^*$, s_1 **can reach** s_2 **in one step of** C, denoted $s_1 \rightarrow_C s_2$, if applying one transition rule to s_1 yields s_2. And s_1 **can reach** s_2 **in** C if $s_1 \rightarrow_C^* s_2$. Given an input string $s \in \Sigma^*$, a **snapshot of** C **on input** s is any string s' such that s can reach s' in C. For example if we have the rules $\{abc \leftrightarrow adc, dce \leftrightarrow dfe, fe \leftrightarrow fge\}$, and an input $abce$, the snapshots of the computation on this input are

$$\{abce, adce, adfe, adfge\}.$$

5.1 *From cellular automata to mutation systems*

Given a cellular automaton $C = (\Sigma, \delta)$, we describe how to construct a 2-simple mutation system $S = (\Sigma', \mu_p, f)$ such that for every nonempty input string $s \in \Sigma^*$, the snapshots of C on input s are represented by the strings evolvable from $d(s)$ in S.

The simulation of a cellular automaton is more complex than the simulation of a FSM; one step of the cellular automaton may require as many as fourteen point mutations, as shown in Fig 6. To ensure the correct coordination of these mutations, we duplicate the symbols and also allow them to store information about one or two symbols to the left or right. The idea is that before performing a transition of the cellular automaton, the system "locks" the left and right neighbors of the symbol to be changed. The additional symbol $(-)$ marks the left and right edges of the transition. To permit insertions and deletions in the string, there is an extra index (denoted $*$) besides 1 and 2. As an example, the following string

$$a^1 \cdot a^2 \cdot {}_{-}b^1 \cdot {}_{b}b^2 \cdot {}_{bb}c^1 \cdot c^2_{dd} \cdot d^1_d \cdot d^2_{-} \cdot e^1 \cdot e^2$$

represents the string $abcde$ where c has locked its left and right neighbors preparing for a transition. The explicit concatenation operator $(\cdot)$ separates individual symbols above. After a transition has been performed, symbols may unlock their neighbors and return to having empty neighbor information.

Let $J = \{1, 2, *\}$ be the set of indices and $N = \{\lambda\} \cup \{-\} \cup \Sigma \cup \Sigma^2$ be the set of possible neighbor strings. Define the alphabet Σ' for the mutation system as follows.

$$\Sigma' = \{{}_u a^i_v : a \in \Sigma, i \in J, u \in N, v \in N\}.$$

In the symbol ${}_u a^i_v$, a is the main symbol component, i is the index, u (resp. v) is the left (resp. right) neighbor information. Let Σ_1 denote the set of symbols of the form a^i with empty neighbor information and index $i \in \{1, 2\}$.

The symbol duplication map d maps Σ^* to $(\Sigma_1)^*$ by replacing each occurrence of a symbol a by the string $a^1 \cdot a^2$. We define a projection h_1 from $(\Sigma')^*$ to Σ^* that maps each symbol with index 1 to its main symbol component, and maps all others to the empty string. Thus $h_1(d(s)) = s$ for all $s \in \Sigma^*$. Also, for example,

$$h_1({}_{-}a^1 \cdot {}_{a}a^2 \cdot {}_{aa}d^1 \cdot b^2_{cc} \cdot c^1_c \cdot c^2_{-}) = adc.$$

5.2 *Defining the fitness function*

We describe the strictly 2-testable pattern $P = (\mathrm{PRE}, \mathrm{MID}, \mathrm{SUF})$ that determines the fitness function f of the mutation system. PRE consists of all symbols a^1 and ${}_{-}a^1$ such that $a \in \Sigma$. SUF consists of all symbols a^2 and a^2_{-} such that $a \in \Sigma$. The set MID contains strings of length two to deal with the situations: (1) empty neighbor information, (2) substitution rules, and (3) insertion/deletion rules.

5.2.1 Empty neighbor information

To permit duplicated symbols we have $a^1 \cdot a^2$ for all $a \in \Sigma$. To permit a boundary between symbols we have $a^2 \cdot b^1$ for all $a, b \in \Sigma$. Together with PRE and SUF, these cases ensure that $f(d(s)) = 1$ for every nonempty string $s \in \Sigma^*$.

5.2.2 Substitution rules

For each substitution rule $axb \leftrightarrow ayb$ we add strings to MID that permit $d(axb)$ and $d(ayb)$ to mutate to each other as follows.

To add left neighbor information $-$ to a^1 we have $c^2 \cdot {}_{-}a^1$ and $c^2_{-} \cdot {}_{-}a^1$ for all $c \in \Sigma$, as well as ${}_{-}a^1 \cdot a^2$. To add right neighbor information $-$ to b^2 we have $b^2_{-} \cdot d^1$ and $b^2_{-} \cdot {}_{-}d^1$ for all $d \in \Sigma$, as well as $b^1 \cdot b^2_{-}$.

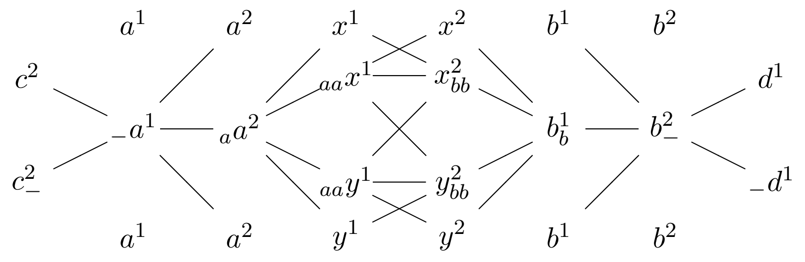

Figure 4. MID strings allowing substitutions for the rule $axb \leftrightarrow ayb$.

To add left neighbor information a to the symbol a^2 we have ${}_{-}a^1 \cdot {}_{a}a^2$, as well as ${}_{a}a^2 \cdot x^1$ and ${}_{a}a^2 \cdot y^1$. To add right neighbor information b to the symbol b^1 we have $b^1_b \cdot b^2_{-}$, as well as $x^2 \cdot b^1_b$ and $y^2 \cdot b^1_b$.

To add left neighbor information aa to the symbol x^1 or y^1 we have ${}_{a}a^2 \cdot {}_{aa}x^1$ and ${}_{aa}x^1 \cdot x^2$, as well as ${}_{a}a^2 \cdot {}_{aa}y^1$ and ${}_{aa}y^1 \cdot y^2$. To add right neighbor information bb to the symbol x^2 or y^2 we have $x^2_{bb} \cdot b^1_b$ and $x^1 \cdot x^2_{bb}$, as well as $y^2_{bb} \cdot b^1_b$ and $y^1 \cdot y^2_{bb}$. The strings that permit both left neighbor information of aa on x^1 and right neighbor information bb on x^2 (and similarly for y^1 and y^2) are ${}_{aa}x^1 \cdot x^2_{bb}$ and ${}_{aa}y^1 \cdot y^2_{bb}$.

The above strings permit consecutive symbols indexed 1 and 2 only if they have the same main symbol. However, we need to permit x to be replaced by y and vice versa. The strings that permit this are ${}_{aa}x^1 \cdot y^2_{bb}$ and ${}_{aa}y^1 \cdot x^2_{bb}$. Figure 4 shows the strings added to MID for the substitution rule $axb \leftrightarrow ayb$. Each line connects two symbols forming a string in MID.

5.2.3 *Insertion/deletion rules*

For each insertion/deletion transition rule $ac \leftrightarrow abc$ we add the following strings to MID. To add left neighbor information $-$ to a^1 we have $d^2 \cdot {}_{-}a^1$ and $d^2_{-} \cdot {}_{-}a^1$ for all $d \in \Sigma$, as well as ${}_{-}a^1 \cdot a^2$. To add right neighbor information $-$ to c^2 we have $c^2_{-} \cdot e^1$ and $c^2_{-} \cdot {}_{-}e^1$ for all $e \in \Sigma$, as well as $c^1 \cdot c^2_{-}$.

To add left neighbor information a to a^2 we have ${}_{-}a^1 \cdot {}_{a}a^2$ as well as ${}_{a}a^2 \cdot b^1$ and ${}_{a}a^2 \cdot c^1$. To add right neighbor information c to c^1 we have $c^1_c \cdot c^2_{-}$ as well as $b^2 \cdot c^1_c$ and $a^2 \cdot c^1_c$. The string that permits both left neighbor information of a on a^2 and right neighbor information of c on c^1 when a^2 and c^1 are adjacent is ${}_{a}a^2 \cdot c^1_c$.

To add left neighbor information aa to b^1 when a^2 is adjacent to b^1, we have ${}_{a}a^2 \cdot {}_{aa}b^1$ and ${}_{aa}b^1 \cdot b^2$.

To allow b to be deleted or inserted, we add strings using the $*$ index that permit b^2 to become ${}_{aa}b^*_{cc}$ and vice versa, namely ${}_{aa}b^1 \cdot {}_{aa}b^*_{cc}$ and ${}_{aa}b^*_{cc} \cdot c^1_c$. Finally we add a string that permits the insertion/deletion of b^1 and ${}_{aa}b^*_{cc}$, namely ${}_{a}a^2 \cdot {}_{aa}b^*_{cc}$. Figure 5 shows the strings in MID for the insertion/deletion rule $ac \leftrightarrow abc$. Again each line connecting two symbols indicates a string in MID.

This completes the construction of MID and the mutation system S. To see that f permits the transitions of C to be simulated, we prove the following.

Lemma 5.1 If $s \in \Sigma^*$ is nonempty and $s \rightarrow_C t$ then $d(s) \rightarrow^*_S d(t)$.

Proof If t is obtained from s by using a substitution rule to substitute ayb for axb in s, then the sequence of point mutations in Figure 6 applied to the relevant portion of $d(s)$ shows that $d(t)$ is evolvable from $d(s)$. Symbols (if any) to the left and right of this portion of $d(s)$ are unchanged.

If t is obtained from s by using an insertion/deletion rule to replace abc in s by ac, then the sequence of point mutations in Figure 7 applied to the relevant portion of $d(s)$ shows that $d(t)$ is evolvable from $d(s)$. Symbols (if any) to the left and right of this portion of $d(s)$ are unchanged. Because point mutations are reversible, the

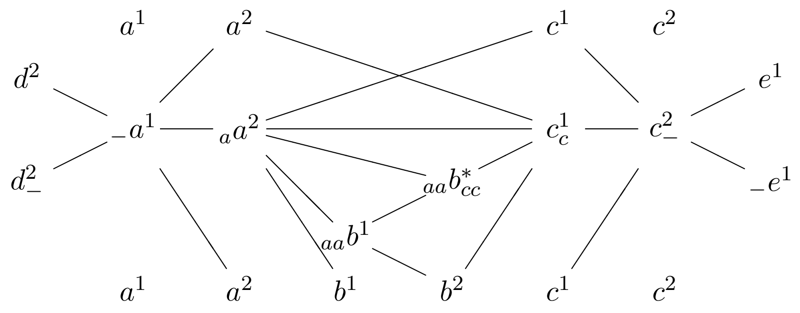

Figure 5. MID strings allowing insertions and deletions for the rule $ac \leftrightarrow abc$.

$$\begin{array}{cccccccccccc}
a^1 & \cdot & a^2 & \cdot & x^1 & \cdot & x^2 & \cdot & b^1 & \cdot & b^2 \\
\_a^1 & \cdot & a^2 & \cdot & x^1 & \cdot & x^2 & \cdot & b^1 & \cdot & b^2 \\
\_a^1 & \cdot & a^2 & \cdot & x^1 & \cdot & x^2 & \cdot & b^1 & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & x^1 & \cdot & x^2 & \cdot & b^1 & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & x^1 & \cdot & x^2 & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}x^1 & \cdot & x^2 & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}x^1 & \cdot & x^2_{bb} & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}y^1 & \cdot & x^2_{bb} & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}y^1 & \cdot & y^2_{bb} & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}y^1 & \cdot & y^2 & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & y^1 & \cdot & y^2 & \cdot & b^1_b & \cdot & b^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & y^1 & \cdot & y^2 & \cdot & b^1 & \cdot & b^2_{\_} \\
\_a^1 & \cdot & a^2 & \cdot & y^1 & \cdot & y^2 & \cdot & b^1 & \cdot & b^2_{\_} \\
\_a^1 & \cdot & a^2 & \cdot & y^1 & \cdot & y^2 & \cdot & b^1 & \cdot & b^2 \\
a^1 & \cdot & a^2 & \cdot & y^1 & \cdot & y^2 & \cdot & b^1 & \cdot & b^2
\end{array}$$

Figure 6. Sequence of mutations to achieve $d(axb) \leftrightarrow_S^* d(ayb)$.

$$\begin{array}{ccccccccccc}
a^1 & \cdot & a^2 & \cdot & b^1 & \cdot & b^2 & \cdot & c^1 & \cdot & c^2 \\
\_a^1 & \cdot & a^2 & \cdot & b^1 & \cdot & b^2 & \cdot & c^1 & \cdot & c^2 \\
\_a^1 & \cdot & a^2 & \cdot & b^1 & \cdot & b^2 & \cdot & c^1 & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & b^1 & \cdot & b^2 & \cdot & c^1 & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & b^1 & \cdot & b^2 & \cdot & c^1_c & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}b^1 & \cdot & b^2 & \cdot & c^1_c & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & {}_{aa}b^1 & \cdot & {}_{aa}b^*_{cc} & \cdot & c^1_c & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & & & {}_{aa}b^*_{cc} & \cdot & c^1_c & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & & & & & c^1_c & \cdot & c^2_{\_} \\
\_a^1 & \cdot & {}_aa^2 & \cdot & & & & & c^1 & \cdot & c^2_{\_} \\
\_a^1 & \cdot & a^2 & \cdot & & & & & c^1 & \cdot & c^2_{\_} \\
\_a^1 & \cdot & a^2 & \cdot & & & & & c^1 & \cdot & c^2 \\
a^1 & \cdot & a^2 & \cdot & & & & & c^1 & \cdot & c^2
\end{array}$$

Figure 7. Sequence of mutations to achieve $d(abc) \leftrightarrow_S^* d(ac)$.

reverse of this sequence indicates how ac can be replaced by abc. ■

5.3 *Correctness of the cellular automaton simulation*

Theorem 5.2 Let $C = (\Sigma, \delta)$ be a cellular automaton and let $S = (\Sigma', \mu_p, f)$ be the 2-simple mutation system constructed from C as described above. Let $s \in \Sigma^*$ be a nonempty string. For any string t reachable from s in C, the string $d(t)$ is evolvable from $d(s)$. Conversely, for any string s' evolvable in S from $d(s)$, $h_1(s')$ is reachable from s in C.

Proof The first part follows by induction on the number of transitions to reach t from s in C, using Lemma 5.1.

For the converse, it suffices to show that if $d(s) \to_S^* s'$ and $s \to_C^* h_1(s')$ and $s' \to_S t$ then $h_1(s') \to_C^* h_1(t)$.

Suppose a^1 is the first symbol and b^2 is the last symbol of $d(s)$. To maintain fitness, these symbols cannot be deleted, and no symbol can be inserted before the first or after the last. The only changes they can undergo that result in fit strings is that a^1 can be replaced by $\_a^1$ and vice versa, and b^2 can be replaced by $b^2_{\_}$ and vice versa. Thus we need only consider changes to interior symbols.

Let $s' \in (\Sigma')^*$ be any nonempty fit string. The indices of any three consecutive symbols in s' must be one of the seven possibilities: $(1,2,1)$, $(2,1,2)$, $(1,2,*)$, $(2,1,*)$, $(1,*,1)$, $(2,*,1)$, and $(*,1,2)$.

If the deletion of a symbol from the interior of s' yields another fit string t, then the symbol deleted must be the middle symbol in one of the index sequences: $(1,2,*)$, $(2,1,*)$ or $(2,*,1)$. In the first and third cases the symbols of index 1 are unchanged and $h_1(t) = h_1(s')$. In the case of $(2,1,*)$, the three symbols in s' must be

$$ {}_{a}a^{2} \cdot {}_{aa}b^{1} \cdot {}_{aa}b^{*}_{cc}, $$

which implies that $abc \leftrightarrow ac$ is a rule in C. Moreover, the symbol before this triple must be ${}_{\_}a^{1}$ and the symbol after it must be c^{1}_{c}, which means that $h(t)$ is obtained from $h_1(s')$ by replacing abc by ac, and $h_1(s') \to_C h_1(t)$.

Analogously, if an insertion of a symbol in the interior of s' yields another fit string t, then only an insertion into $(2,*)$ (yielding $(2,1,*)$) results in $h_1(t) \neq h_1(s')$. This implies that the inserted symbol and its two neighbors to the left and right in t are as follows:

$$ {}_{\_}a^{1} \cdot {}_{a}a^{2} \cdot {}_{aa}b^{1} \cdot {}_{aa}b^{*}_{cc} \cdot c^{1}_{c}. $$

Thus, $abc \leftrightarrow ac$ is a rule of C and $h_1(t)$ is obtained from $h_1(s')$ by replacing ac by abc and $h_1(s') \to_C h_1(t)$.

If a replacement of one interior symbol of s' by another yields a fit string t, then either the replacement changes the index of the symbol or not. The only possible kinds of replacements that change the index of the symbol are of the form $(1,2,1) \leftrightarrow (1,*,1)$. This leaves the symbols of index 1 unchanged, and $h_1(t) = h_1(s')$.

Thus the only replacements that we must consider are replacements of symbols of index 1 by symbols of index 1 with a different main symbol, so that $h_1(t) \neq h_1(s')$. The indices of the replaced symbol and its two neighbors must be either $(2,1,*)$ or $(2,1,2)$. In the first case, the three symbols of s' are of the form

$$ {}_{a}a^{2} \cdot {}_{aa}b^{1} \cdot {}_{aa}b^{*}_{cc}, $$

and there is no other symbol that can replace ${}_{aa}b^{1}$ and yield a fit string t. In the case of $(2,1,2)$ the possibilities for the symbol of index 1 are a^{1}, ${}_{\_}a^{1}$, a^{1}_{a}, and ${}_{bb}a^{1}$. When the symbol to its right is one of a^{2}, $a^{2}_{\_}$, or ${}_{a}a^{2}$, replacing the symbol of index 1 in s' by a symbol of index 1 and main symbol other than a does not yield a fit string. Thus, the only possibilities in s' for the symbol of index 1 and its right neighbor are the following: (1) $a^{1} \cdot a^{2}_{bb}$, (2) ${}_{bb}a^{1} \cdot a^{2}_{cc}$, (3) ${}_{bb}a^{1} \cdot c^{2}_{dd}$.

In case (1) the only replacement for a^{1} that changes the main symbol component is of the form ${}_{dd}c^{1}$ and yields

$$ {}_{\_}d^{1} \cdot {}_{d}d^{2} \cdot {}_{dd}c^{1} \cdot a^{2}_{bb} \cdot b^{1}_{b} $$

in t. Then $dcb \leftrightarrow dab$ is a rule in C and $h_1(t)$ is obtained from $h_1(s')$ by replacing dab by dcb, so that $h_1(s') \to_C h_1(t)$.

In cases (2) and (3) the symbols to the left of ${}_{bb}a^{1}$ must be ${}_{\_}b^{1} \cdot {}_{b}b^{2}$. The only possible replacement for ${}_{bb}a^{1}$ that changes the main symbol component is of the form ${}_{bb}e^{1}$.

In case (2), the result in t is

$$ {}_{\_}b^{1} \cdot {}_{b}b^{2} \cdot {}_{bb}e^{1} \cdot a^{2}_{cc} \cdot c^{1}_{c}. $$

Thus $bec \leftrightarrow bac$ is a rule in C and $h_1(t)$ is obtained from $h_1(s')$ by replacing bac by bec, so that $h_1(s') \to_C h_1(t)$.

In case (3), the result in t is

$$_{-}b^1 \cdot {}_bb^2 \cdot {}_{bb}e^1 \cdot c^2_{dd} \cdot d^1_d.$$

Thus both $bed \leftrightarrow bcd$ and $bad \leftrightarrow bcd$ are rules in C, and $h_1(t)$ is obtained from $h_1(s')$ by replacing bad by bed. Though this is not necessarily a single step of C, it is accomplished by two steps: $bad \to_C bcd \to_C bed$, so that $h_1(s') \to^*_C h_1(t)$, which concludes the proof of Theorem 5.2. ■

6. Conclusion

We have introduced mutation systems to model the evolution of a string subject to the effects of mutations and a fitness function. We have shown that 2-simple mutation systems, defined as having point mutations and a strictly 2-testable fitness function, are sufficiently powerful to simulate computation by nondeterministic finite automata. Under the assumption of random point mutations, the number of steps to simulate a deterministic finite automaton is bounded by a polynomial in the number of steps in the computation. We have also shown that 2-simple mutation systems can simulate one-dimensional asynchronous reversible cellular automata with insertions and deletions. Because this is a universal model of computation, it is in general undecidable to predict whether one string can evolve into another in a 2-simple mutation system.

Some possible generalizations of our definitions may be fruitful to explore. Instead of just one evolving string, we could consider a population of evolving strings. Rather than a deterministic, time-invariant fitness function, we could consider fitness functions that were probabilistic and/or time-varying, possibly depending on comparisons with other strings in the current population.

A promising future direction is to explore the learnability of fitness functions given positive data derived from the evolution of one or more strings in a mutation system. The intuition is that observation of strings related by mutation and evolution may provide additional information for a learning process. One reason for considering k-testable fitness functions in connection with learnability is that for any k the class of strictly k-testable languages, and even the class of concatenations of strictly k-testable languages, are learnable in the limit from positive data [9, 13].

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant Number CCF-0916389. A preliminary version of this paper appears in the proceedings of LATA 2011 [2]. Raonne Barbosa Vargas is now employed by Microsoft Corporation. The authors thank David Eisenstat and Sarah Eisenstat for help with aspects of this paper.

References

[1] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell.* Garland Publishing, 1994.

[2] Dana Angluin, James Aspnes, and Raonne Barbosa Vargas. Mutation systems. In *Language and Automata Theory and Applications: 5th International Conference, LATA 2011, Tarragona, Spain,*

May 26-31, 2011. Proceedings, volume 6638 of *Lecture Notes in Computer Science*, pages 92–104. Springer-Verlag, May 2011.

[3] C.H. Bennett. Logical reversibility of computation. *IBM J. RES. DEVELOP.*, pages 525–532, November 1973.

[4] Charles H. Bennett. The thermodynamics of computation – a review. *International Journal of Theoretical Physics*, 21:905–940, 1982.

[5] J.A. Brzozowski and Imre Simon. Characterizations of locally testable events. *Discrete Mathematics*, 4:243–271, 1973.

[6] Matteo Cavaliere and Peter Leupold. Evolution and observation – a non-standard way to generate formal languages. *Theoretical Computer Science*, 321:233–248, 2004.

[7] Matteo Cavaliere and Peter Leupold. Computing by observing: Simple systems and simple observers. *Theoretical Computer Science*, 412:113–123, 2010.

[8] Kenneth A. De Jong. *Evolutionary Computation: A Unified Approach.* MIT Press, Cambridge, MA, 2006.

[9] P. García and E. Vidal. Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12:920–925, 1990.

[10] Tom Head. Splicing representations of strictly locally testable languages. *Discrete Appl. Math.*, 87:139–147, 1998.

[11] Jarkko Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334(1–3):3–33, April 2005.

[12] Sam M. Kim, Robert McNaughton, and Robert McCloskey. A polynomial time algorithm for the local testability problem of deterministic finite automata. *Algorithms and Data Structures*, 382:420–436, 1989.

[13] Satoshi Kobayashi and Takashi Yokomori. Learning concatenations of locally testable languages from positive data. In Setsuo Arikawa and Klaus Jantke, editors, *Algorithmic Learning Theory*, volume 872 of *Lecture Notes in Computer Science*, pages 407–422. Springer Berlin / Heidelberg, 1994.

[14] K. Lindgren and M.G. Nordahl. Universal computation in simple one-dimensional cellular automata. *Complex Systems*, 4:299–318, 1990.

[15] Robert McNaughton. Algebraic decision procedures for local testability. *Theory of Computing Systems*, 8(1):60–76, March 1974.

[16] John Quackenbush. Computational analysis of microarray data. *Genetics*, 2:418–427, June 2001.

[17] Martin Tompa, Nan Li, and Timothy Bailey. Assessing computational tools for the discovery of transcriptional factor binding sites. *Nature Biotechnology*, 23(1):137–144, January 2005.

[18] Martin Tompa and Amol Prakash. Discovery of regulatory elements in vertebrates through comparative genomics. *Nature Biotechnology*, 23(10):1249–1256, October 2005.

[19] Leslie G. Valiant. Evolvability. *J. ACM*, 56:3:1–3:21, 2009.

[20] J. Von Neumann. Theory of self-reproducing automata. Editor A.W. Burks, University of Illinois Press, 1966.

[21] Wyeth W. Wasserman and Albin Sandelin. Applied bioinformatics for the identification of regulatory elements. *Genetics*, 5:276–287, April 2004.

[22] Takashi Yokomori and Satoshi Kobayashi. Learning local languages and their application to DNA sequence analysis. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20:1067–1079, 1998.