Mutation Systems*

Dana Angluin, James Aspnes, and Raonne Barbosa Vargas

Department of Computer Science, Yale University

Abstract. 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 $k \geq 2$, 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.

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 simple model of variation and selection acting on strings of symbols, with the goal of exploring the properties of such systems, specifically what we can predict and learn 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.

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 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

* Research supported by the National Science Foundation under Grant CCF-0916389.

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 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 ($f(s) = 0$).*

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 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.

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 *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. $\mu_p(s) = \mu_d(s) \cup \mu_i(s) \cup \mu_r(s)$.

The mutator μ_p permits any single point mutation of a string. Reversibility is a relevant property of mutators and mutation systems.

Definition 4 *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 following lemma is immediate.

Lemma 1 *If μ is stepwise reversible then $S = (\Sigma, \mu, f)$ is reversible.*

3.2 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 [3, 7, 10]. Head [5] and Yokomori and Kobayashi [13] describe applications of k-testable languages to modeling biological phenomena.

Definition 5 *Let Σ be an alphabet. A strictly k-testable pattern $P = (PRE, MID, SUF)$ is composed of three sets of strings with $PRE \subseteq \Sigma^{k-1}$, $MID \subseteq \Sigma^k$, and $SUF \subseteq \Sigma^{k-1}$. 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.*

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 $s \in L_P$. A **k-simple mutation system** is 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.

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: Symbol Duplication. Let $\Sigma = \{a, b\}$. We define a 2-simple mutation system $S_2 = (\Sigma_2, \mu_p, f_2)$ that protects strings against point mutations. The alphabet Σ_2 is $D(\Sigma) = \{a^1, a^2, b^1, b^2\}$ and the strictly 2-testable fitness function f_2 is defined by the prefix strings $\{a^1, b^1\}$, the suffix strings $\{a^2, b^2\}$, and the middle strings

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

The set of strings that are fit with respect to f_2 are exactly those of the form $d(s)$ for some nonempty $s \in \Sigma^*$, for example, $a_1a_2b_1b_2b_1b_2$. If a fit string undergoes any non-identity point mutation, the resulting string is not fit with respect to f_2.

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 [1]. 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 $0 \leq i \leq n$, where for each $1 \leq j < i$, $q_{j+1} \in \delta(q_j, s[j])$. 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: M_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

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

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: $a_x^1 a_x^2$ for all $a \in \Sigma$.
2. Initial boundary: $a_x^2 b_x^1$ for all $a, b \in \Sigma$.
3. Duplicate update needed: $a_q^1 a_x^2$ for all $a \in \Sigma$ and $q \in Q$.
4. Updated duplicate: $a_q^1 a_q^2$ for all $a \in \Sigma$ and $q \in Q$.
5. Updated boundary: $a_q^2 b_x^1$ for all $a, b \in \Sigma$ and $q \in Q$.
6. State transition: $a_q^2 b_{q'}^1$ 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 1. Each FSM step requires two mutations.

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

Fig. 1. 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 1; their proofs are omitted for lack of space.

Lemma 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 $a^1_x a^2_x$ or $a^1_q a^2_x$ or $a^1_q a^2_q$ for some $a \in \Sigma$ and $q \in Q$.*
3. *If two consecutive symbols of s' are indexed* 2 *and* 1*, they must be $a^2_x b^1_x$ or $a^2_q b^1_x$ or $a^2_q b^1_{q'}$ 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 $h_1(s')$.*

Lemma 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.*

Lemma 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 $d(I_x(s))$.*

Theorem 1 *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 above. 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 $d(I_x(s))$.*

In case M is nondeterministic, the strings evolvable from $d(I_x(s))$ in S give 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. In case 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 history in which all symbols have state components in Q at the other end.

We consider **random point mutations**, in which each type of mutation (deletion, insertion, replacement) is selected with some probability, and for each type, a string position to apply it is selected equiprobably, and a symbol is selected equiprobably from the alphabet for an insertion or a replacement. For a deterministic machine M, the result is a Markov chain of $2n$ vertices that moves forward when a mutation causes another symbol to have state component $q \in Q$ and backward when a mutation causes another symbol to have state component x.

In the construction described above, the probability of a forward mutation and a backward mutation is the same, namely $p_r/(2n|\Sigma'|)$ where p_r is the probability of choosing replacement. By standard results on random walks, this implies that the expected number of attempted mutations for the simulation to reach the final string is $O(|\Sigma'|n^3/p_r)$. However, by biasing the random walk in the forward direction, this can be reduced to $O(|\Sigma'|n^2/p_r)$, as suggested by Bennett [2]. For example, if we make an additional copy of every symbol a^i_q such that $q \in Q$, and treat them as equivalent in the simulation, then the probability of a forward mutation is twice that of a backward mutation.

5 Simulating Cellular Automata

Cellular automata are a well known model of computation introduced by Von Neumann [12], motivated by physical and biological problems. In a recent survey paper,

Kari [6] 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 consider one-dimensional asynchronous reversible cellular automata with insertions and deletions because they support universal computation [9].

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 $s_1 \to_C s_2$, if applying one transition rule to s_1 yields s_2. And s_1 **can reach** s_2 **in** C if $s_1 \to_C^* s_2$. 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 fourteen point mutations. 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 $(*)$ 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 $P = (\text{PRE}, \text{MID}, \text{SUF})$ 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.

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 $s \in \Sigma^*$.

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 $b^1 \cdot b^2_-$.

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 2 shows the strings added to MID for the substitution rule $axb \leftrightarrow ayb$. Each line connects two symbols forming a string in MID.

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 $a^2 \cdot c^1_1$. 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 3 shows the strings in MID for the insertion/deletion rule $ac \leftrightarrow abc$. Again each line connecting two symbols indicates a string in MID.

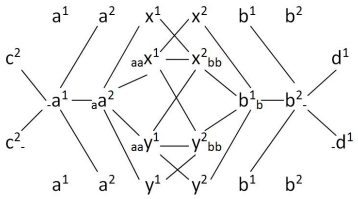

Fig. 2. MID strings allowing substitutions for the rule $axb \leftrightarrow ayb$.

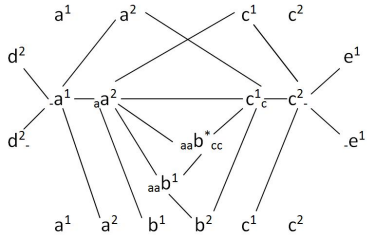

Fig. 3. MID strings allowing insertions and deletions for the rule $ac \leftrightarrow abc$.

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 *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 4 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 5 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 reverse of this sequence indicates how ac can be replaced by abc. □

5.3 Correctness of the cellular automaton simulation

Theorem 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.

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

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

Fig. 4. Sequence of mutations to achieve $d(axb) \leftrightarrow^*_S d(ayb)$.

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

Fig. 5. Sequence of mutations to achieve $d(abc) \leftrightarrow^*_S d(ac)$.

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 $h_1(s') \rightarrow_C h_1(t)$.

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 $h_1(s') \rightarrow_C h_1(t)$.

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 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 $h_1(s') \rightarrow_C h_1(t)$.

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 $h_1(s') \rightarrow_C h_1(t)$.

In case (3), the result in t is

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

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: $bad \rightarrow_C bcd \rightarrow_C bed$, so that $h_1(s') \rightarrow^*_C h_1(t)$, which concludes the proof of Theorem 2. □

6 Discussion

We have introduced mutation systems to model the evolution of a string subject to the effects of mutations and a fitness function. Some possible generalizations of our definition may be fruitful to explore: a population of evolving strings, a probabilistic or time-varying fitness function, or a fitness function that depends on comparing strings in the current population.

Comparing our mutation systems to Valiant's concept of evolvability [11] we note that his model is designed to explore the question of what functions can be efficiently approximated through a polynomial-time evolution process, while our model does not have a final ideal target, but instead has a variety of evolution pathways and outcomes defined by the mutation operator and the fitness function.

We have shown that mutation systems with point mutations and strictly 2-testable fitness functions can represent general computation, and therefore it is in general undecidable to predict whether one string can evolve into another in such systems. By contrast, 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 [4, 8]. 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.

Acknowledgements. 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. Bennett, C.: Logical reversibility of computation. IBM J. RES. DEVELOP. (November 1973) 525–532
2. Bennett, C.H.: The thermodynamics of computation – a review. International Journal of Theoretical Physics **21** (1982) 905–940
3. Brzozowski, J., Simon, I.: Characterizations of locally testable events. Discrete Mathematics **4** (1973) 243–271
4. García, P., Vidal, E.: Inference of k-testable languages in the strict sense and application to syntactic pattern recognition. IEEE Trans. Pattern Anal. Mach. Intell. **12** (1990) 920–925
5. Head, T.: Splicing representations of strictly locally testable languages. Discrete Appl. Math. **87** (1998) 139–147
6. Kari, J.: Theory of cellular automata: A survey. Theoretical Computer Science **334**(1–3) (April 2005) 3–33
7. Kim, S.M., McNaughton, R., McCloskey, R.: A polynomial time algorithm for the local testability problem of deterministic finite automata. Algorithms and Data Structures **382** (1989) 420–436
8. Kobayashi, S., Yokomori, T.: Learning concatenations of locally testable languages from positive data. In Arikawa, S., Jantke, K., eds.: Algorithmic Learning Theory. Volume 872 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1994) 407–422
9. Lindgren, K., Nordahl, M.: Universal computation in simple one-dimensional cellular automata. Complex Systems **4** (1990) 299–318
10. McNaughton, R.: Algebraic decision procedures for local testability. Theory of Computing Systems **8**(1) (March 1974) 60–76
11. Valiant, L.G.: Evolvability. J. ACM **56** (2009) 3:1–3:21
12. Von Neumann, J.: Theory of self-reproducing automata. Editor A.W. Burks, University of Illinois Press (1966)
13. Yokomori, T., Kobayashi, S.: Learning local languages and their application to DNA sequence analysis. IEEE Trans. Pattern Anal. Mach. Intell. **20** (1998) 1067–1079