Submitted to ISMB-95

Automata-Theoretic Models of Mutation and Alignment

David B. Searls and Kevin P. Murphy*
Department of Genetics, Rm. 475CRB
University of Pennsylvania School of Medicine
422 Curie Boulevard, Philadelphia, PA 19104-6145 USA
dsearls@cbil.humgen.upenn.edu, (215)573-3107, FAX -3111

Abstract

Finite-state automata called *transducers*, which have both input and output, can be used to model simple mechanisms of biological mutation. We present a methodology whereby numerically-weighted versions of such specifications can be mechanically adapted to create string edit machines that are essentially equivalent to recurrence relations of the sort that characterize dynamic programming alignment algorithms. Based on this, we have developed a visual programming system for designing new alignment algorithms in a rapid-prototyping fashion.

1 Introduction

Finite-state automata have an important place in computer science, often representing simple models of computation as the recognition or generation of strings of symbols. A wide variety of such automata have been intensively studied, including *weighted* automata which have numbers associated with transitions between states, and *transducers* which have both input and output.

Allison and co-workers [2] have proposed the use of finite-state models for mutation, and furthermore shown how some simple versions of such models (corresponding to constant and piecewise linear gap penalties) can be related to dynamic programming alignment algorithms. This builds upon early work by Karp and Held [5] establishing the relationship between weighted finite-state automata and generalized dynamic programming. We have been working to extend this notion in a biological context and to rigorously capture the relationships between simple mutational models, edit distance, and alignment. We find that not only can these transitions be neatly formalized, but that the resulting methodologies can be used to create tools to assist in the design of new algorithms.

Haussler and colleagues [6] have also made imaginative use of simple mutational models and finite-state automata, as embodied in hidden Markov models (HMMs) that they use to learn profile descriptors of multiply-aligned protein families. Here again there is an implicit relationship with dynamic programming through the algorithms associated with HMMs, although in this case the automata are used to model evolutionary relationships among specific sets of strings rather than general algorithms for comparing arbitrary strings under some model of evolution. Thus, the automata involved in describing a class of proteins directly represent, and are roughly co-extensive with, the strings themselves (i.e. the number of states and transitions is proportional to the lengths of the strings). If, on the other hand, the focus is shifted to modelling the variety of possible mutational mechanisms rather than the specific results of those mutations, we find that much more compact automata can be designed that in effect map naturally to alignment algorithms rather than strings themselves. In addition to the formal methodology described above, several novel alignment algorithms will be presented in this paper based on such finite-state automata.

*Current address: Department of Computer Science, University of California at Davis, murphyk@cs.ucdavis.edu

2 Models

The basic theoretical construct with which we will deal is the following [1]:

Definition 2.1 (Finite Transducer) *A* finite transducer *is a 6-tuple* $T = \langle Q, \Sigma, \Omega, \delta, s, F \rangle$ *where* Q *is a finite nonempty set of* states, Σ *and* Ω *are finite nonempty* input *and* ouput *alphabets, respectively,* $s \in Q$ *is a distinguished* start *state,* $F \subseteq Q$ *is a set of* final *states, and* $\delta \subset Q \times \Sigma^* \times \Omega^* \times Q$ *is a finite set of* transitions. *An input string* $u \in \Sigma^*$ *is said to be* accepted *by* T *with output* $v \in \Omega^*$ *iff* $u = u_1 u_2 \cdots u_n$, $v = v_1 v_2 \cdots v_n$, *and* $\langle q_i, u_{i+1}, v_{i+1}, q_{i+1} \rangle \in \delta$ *for* $0 \leq i < n$, *where* $q_0 = s$ *and* $q_n \in F$. *The set of all such outputs for any* $u \in \Sigma^*$ *is denoted* $T(u)$, *and the sequence of transitions employed is called a* derivation. *The following variations are also defined:*

- *A* weighted *finite transducer is one for which* $\delta \subset Q \times \Sigma^* \times \Omega^* \times \Re \times Q$, *where* $\Re$ *is the set of reals. This number is called the* weight *of the transition, and the weight* n *of a derivation is the sum of the weights of its transitions.*
- *A* two-tape *finite transducer is one for which* $\delta \subset Q \times \Sigma^* \times \Sigma^* \times \Omega^* \times Q$, *and where a pair of input strings* $u, v \in \Sigma^*$ *are said to be* accepted *by* T *with output* $w \in \Omega^*$ *iff* $u = u_1 u_2 \cdots u_n$, $v = v_1 v_2 \cdots v_n$, $w = w_1 w_2 \cdots w_n$, *and* $\langle q_i, u_{i+1}, v_{i+1}, w_{i+1}, q_{i+1} \rangle \in \delta$ *for* $0 \leq i < n$, *where* $q_0 = s$ *and* $q_n \in F$.
- *A weighted, two-tape finite transducer is one for which* $\delta \subset Q \times \Sigma^* \times \Sigma^* \times \Omega^* \times \Re \times Q$, *with derivations and notation as before.*

We will employ the usual graphic conventions for such automata, for example representing a transition from the start state s to a final state $f \in F$ via some $\langle s, u, v, n, f \rangle \in \delta$ for a weighted finite transducer as shown in Figure 1.

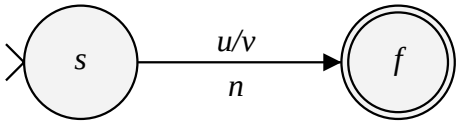

Figure 1: Graphical conventions for finite transducers

It will be convenient to use finite transducers that have essentially the same input and output alphabets, but for which the output is "labelled". For this, we define the following:

Definition 2.2 (Label) *A* labelling *of an alphabet* Σ *is a bijection* $L : \Sigma \rightarrow \overline{\Sigma}$ *in which each* $x \in \Sigma$ *is mapped to a new symbol* $\overline{x} \in \overline{\Sigma}$, *i.e.* $\overline{\Sigma} = \{\overline{x} \mid x \in \Sigma\}$. *Two strings* $u, v \in \Sigma^* \cup \overline{\Sigma}^*$ *are* equivalent up to labelling, *denoted* $u \equiv v$, *if and only if* $u = x_1 x_2 \cdots x_n$, $v = y_1 y_2 \cdots y_n$, *and for* $1 \leq i \leq n$ *either* $x_i = y_i$, $\overline{x}_i = y_i$, *or* $x_i = \overline{y}_i$.

We will now outline an automata-theoretic model of mutation, beginning with a fundamental operation meant to model a single mutational event affecting a substring of some sequence.

Definition 2.3 (Mechanism) *A* (labelled) mechanism M *is a finite transducer for which* $\Omega = \Sigma$ ($\Omega = \overline{\Sigma}$), *and where for all* $u \in \Sigma^*$ *and* $v \in M(u)$, $u \not\equiv v$. *A weighted mechanism is one based on a weighted finite transducer, as above, where in addition for any derivation the weight* n *is non-zero. Where the meaning is obvious from the context, the term* mechanism *will also refer to any derivation by such a machine.*

Note that we have required that the action of a mechanism be *detectable*, i.e. that any derivation by a mechanism produce a change in the string, and furthermore that it have non-zero weight (if any). Also, we will generally require weights to be non-negative. This definition of mutational mechanism is obviously limited, in ways that will be discussed further below. However, by accepting these limitations, we will find that the model proves to be tractable to further useful forms of analysis.[1]

[1] Even so, it must be stressed that the use of a transducer also entails theoretical limitations that do not apply to simple finite-state automata (i.e. without output). For example, we note that it can be shown that it is undecidable whether an arbitrary finite transducer is a mechanism, using a reduction of the Post Correspondence Problem.

Some common notions of mutation can be represented by transducers that are easily seen to be mechanisms, such as those shown in Figure 2. These represent single element substitution, deletion, and insertion; note that the latter accepts only an empty string as input, while the others affect only single bases at a time. The symbols x and y here represent any member of the alphabet, so that in the literal machine there would actually be sixteen substitution transitions in the case of DNA, and four each of deletion and insertion transitions. These might be represented as separate mechanisms, particularly if this was thought to represent some important biological distinction. On the other hand, note that these three mechanisms might be combined into a single mechanism, by simply merging the respective start and final states; what constitutes a distinct mechanism is at the discretion of the modeller. In practice, we will see that weighted, labelled mechanisms will prove to be the most useful form, and that separate or combined mechanisms will preserve the weights and labels in the desired manner.

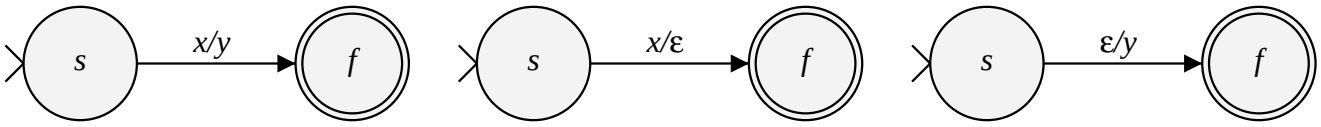

Figure 2: Mechanisms for single-base (a) substitution, (b) deletion, and (c) insertion, where $x, y \in \Sigma$ and $x \neq y$. The symbol ϵ represents the *empty* string, or string of zero length.

In fact, we will now proceed to combine collections of mechanisms for a specific purpose. We will wish to be able to apply a mechanism anywhere within a string, rather than just to a given substring (or, in the previous example, a single base) in its entirety. In order to do this, we will construct a machine that, given a set of mechanisms, applies exactly one of those mechanisms at any permissible point in a string. We do this as follows:

Definition 2.4 (Mutator) *A* (labelled) mutator $\mathcal{M}_\mu$ *is a finite transducer constructed from a set of (labelled) mechanisms* μ *that have common alphabets* Σ_μ *and* Ω_μ*, but pairwise disjoint sets of states* Q_i*, as follows:*

- *the set of states* Q *is the union of the states* Q_i *in each* $M_i \in \mu$*, together with a new start state* s *and a single, new final state,* $F = \{f\}$
- *the alphabets are* $\Sigma = \Omega = \Sigma_\mu$
- *the transitions* δ *comprise the following:*
  - *each transition in* δ_i *of each mechanism in* μ
  - *a new transition* $\langle s, \epsilon, \epsilon, s_i \rangle$ *for each start state* s_i *of each mechanism in* μ
  - *new transitions* $\langle f_i, \epsilon, \epsilon, f \rangle$ *for each final state* $f_i \in F_i$ *of each mechanism in* μ
  - *new transitions* $\langle s, x, x, s \rangle$ *and* $\langle f, x, x, f \rangle$ *for each* $x \in \Sigma_\mu$
  - *if* μ *is labelled, a new transition* $\langle s, \overline{x}, \overline{x}, s \rangle$*, for each* $\overline{x} \in \overline{\Sigma}_\mu$

A weighted mutator is constructed as above, with zero weights attached to each new transition. A (labelled) mutation *is any derivation by a (labelled) mutator.*

The construction of a mutator is perhaps more easily understood using the graphical representation, as shown in Figure 3a. For an unlabelled mechanism set, the reflexive transitions on the start and end states merely serve to consume and produce identical input and output, so that any substring of the overall string may be "presented" to the appropriate mechanism. We will refer to such transitions $\langle q_i, x, x, q_j \rangle \in \delta$, where $x \in \Sigma$, as *scanning* transitions. It is clear, by the construction of a mutator, that only one mechanism's transitions are invoked in any particular mutation; we will refer to that mechanism within a given mutation as, simply, the mechanism of the mutation. The *effect* of a given mutation will refer to the output produced by the mechanism of that mutation, which is a substring

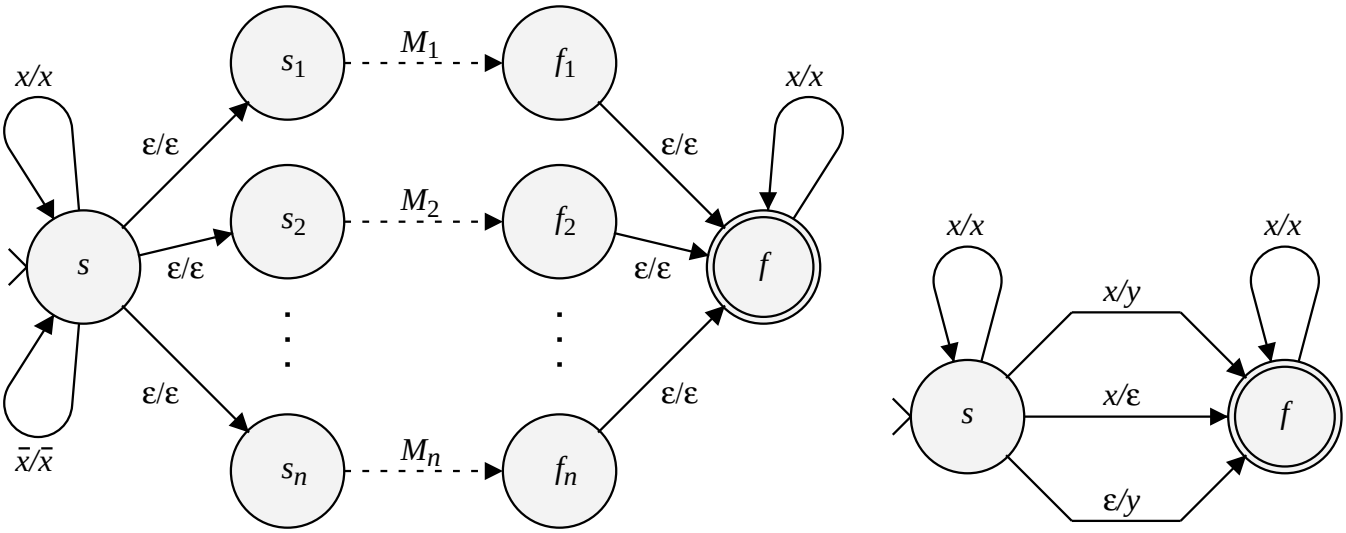

Figure 3: (a) Construction of a mutator from mechanisms M_i (left) and (b) an equivalent transducer to the mutator for the mechanisms of Figure 2 (right).

of the overall mutator's output – in fact, for a labelled mutator acting on unlabelled input, the effect is precisely that substring of the output that is labelled. The substring of the overall input that is consumed by the mechanism will be called the *affected* substring.

A mutator is constructed so as to keep the different mechanisms contributing to it clearly separate and distinct, but in fact an equivalent transducer can be constructed, without empty transitions, by merging states. Such an equivalent mutator, for the mechanisms given in Figure 2, is illustrated in Figure 3b. We note that the purpose of labels is to ensure that *evolutions*, or multiple applications of a mutator to a string, are provably equivalent to another form of automaton derived from it, as described next. Briefly, labelling ensures that in running a string through a mutator multiple times, the effect of any mutation will never be affected in a subsequent mutation. This formalizes an important (and biologically simplistic) assumption in the derivation of edit distances, and can also serve to establish that the number of labelled evolutions of any string is finite.

We now proceed to extend this general model to encompass the notion of string edit and edit distance, by constructing the next in a series of machines:

Definition 2.5 (Editor) *An* editor $\mathcal{E}_\mu$ *is a finite transducer constructed from a mutator* $\mathcal{M}_\mu$ *as follows:*

- *the states* Q*, input and output alphabets* $\Sigma = \Omega$*, and start state* s *are as in* $\mathcal{M}_\mu$
- *the final states are changed from* $F = \{f\}$ *to* $F = \{s\}$
- *the transitions* δ *are those in* $\mathcal{M}_\mu$ *together with a new transition* $\langle f, \epsilon, \epsilon, s\rangle$

A weighted editor is constructed as above from a weighted mutator, with zero weight attached to the new transition. An editing *is any derivation by an editor, and an* edit *is an output of an editor.*

Figure 4a illustrates this construction graphically. An editor goes a step further than a mutator by applying any number of allowable mutation mechanisms to a string. It accomplishes this by, first, allowing the transducer to in fact perform no mutations, by making the start state a final state, and second, to iterate after each mutation by returning to the start state s from what would otherwise be the final state f. Thus, after each mechanism is applied, the input may be further scanned and another mechanism applied at any subsequent point, and so on. Note that the scanning transition on the former final state is now superfluous, and in fact an equivalent editor can generally be constructed by simply *merging the start and final states* of a mutator. Such an equivalent editor, for the mutator of Figure 3b, is illustrated in Figure 4b.

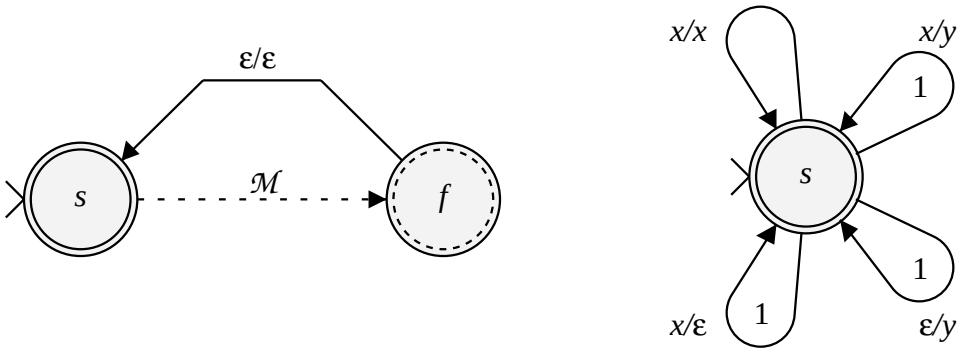

Figure 4: (a) Construction of an editor from a mutator $\mathcal{M}$, of which only the start and (formerly) final states are shown (left), and (b) an equivalent editor for a weighted version of the mutator of Figure 3b (right).

The editor of Figure 4b in addition has weights of "1" attached to each transition formerly associated with a mutation mechanism (i.e. substitution, insertion, and deletion). It can now be seen that this editor in fact constitutes a calculator of the number of mutation mechanisms invoked to transform one string to another in any particular editing. This, of course, is the fundamental notion behind the measure of *edit distance*. While it may seem that we have gone to a great deal of trouble to arrive at this simple calculator, we emphasize again the generality of this approach. For example, it has long been recognized that the simplistic notion of calculating edit distance by assessing a "penalty" for insertions and deletions that is strictly proportional to the length of the resulting gaps, which is inherent in the editor of Figure 4b, is biologically naive. More realistic sequence alignment algorithms recognize that insertions and deletions of any size generally constitute single mutational events, and the metrics imposed involve a constant penalty for any gap, plus some smaller incremental penalty that is proportional to the size of the gap. We can easily adapt to this new model of mutation by substituting for the mechanisms for deletion and insertion ones like that illustrated in Figure 5a.

Here, reflexive arcs accomplish single deletions with weights of "1", as many as are desired, but in order to complete the application of the mechanism, the last transition to the final state is assessed a much greater weight. Thus, a single such event entails one large weight plus an incremental weight in proportion to the length of the indel. (There are several variations to this architecture which would serve equally well.) Such mechanisms can be combined with the previous substitution mechanism (now given an intermediate weight) to create an equivalent mutator, as the reader may confirm, and the mutator's start and end states may then be merged to produce the equivalent editor illustrated in Figure 5c. We also introduce at this point the use of Greek letters to denote weights on transitions.

As suggested above, a series of consecutive deletions or insertions is commonly called a *gap*. A *gap penalty* is the total weight assessed by the mechanisms creating a gap, and is characterized by a function γ_k of the gap length k. Thus, for the naive model of mutation, $\gamma_k = \beta \cdot k$ for transition weight β, but for models such as that of Figure 5 we have a so-called *affine* gap penalty, $\gamma_k = \alpha + \beta \cdot (k-1)$, as may be readily inferred from the automaton.

In fact, the cost of any editing may be inferred from the construction of these automata, and thus also the minimal cost *via* the following inductive definition:

Definition 2.6 (Functional Edit Distance) *For each state $q_i \in Q$ of a weighted editor $\mathcal{E}$ construct a function of the same name from pairs of strings to reals, $q_i : \Sigma^* \times \Sigma^* \to \Re$, such that*

- *for each final state $f \in F$, $f(\epsilon, \epsilon) = 0$*
- *for all $x, y \in \Sigma^*$, $q_i(x,y) = \min \{ q_j(r,s) + n \mid \langle q_i, u, v, n, q_j \rangle \in \delta,\ x = ur \text{ and } y = vs \}$*

Then for any $x, y \in \Sigma^*$ the functional edit distance *from x to y is the value for the start state, $s(x,y)$.*

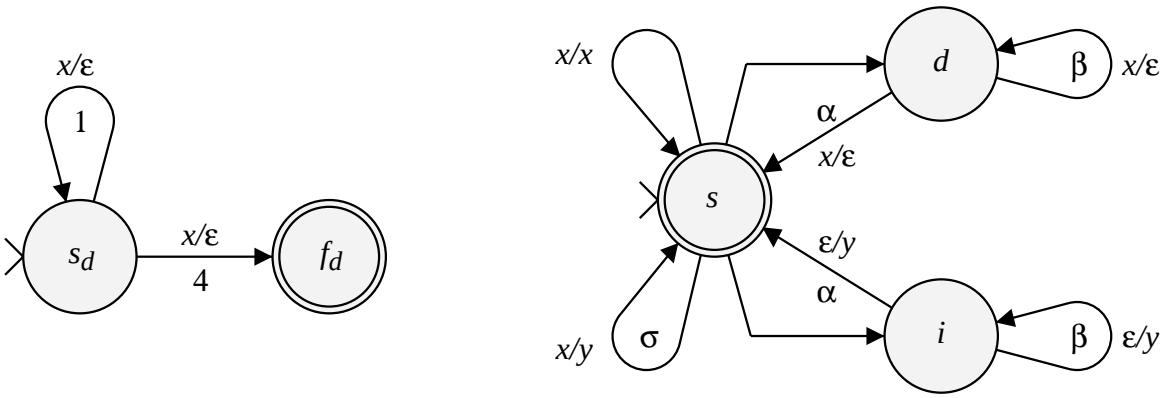

Figure 5: (a) Weighted mechanism for single-event deletion (left) and (b) an equivalent editor for single-event deletion and insertion, plus substitution as before (right).

Applying this definition to the editor of Figure 4b leads to the classic edit distance recurrence,

$$s(xu, yv) = \min \begin{cases} s(u,v) + \sigma_{x,y} & \text{where } \sigma_{x,y} = 0 \text{ if } x = y \text{ and } \sigma_{x,y} = 1 \text{ otherwise} \\ s(u, yv) + 1 & \\ s(xu, v) + 1 & \end{cases}$$

$$s(u,\epsilon) = |u| \qquad s(\epsilon, v) = |v|$$

This follows from the architecture of the automaton when the single state s is interpreted as the recursive function, while transitions represent conditions under which prefixes of the input and output are consumed and generated, respectively, and the total cost incremented so as to arrive in a new machine configuration. Boundary conditions are established by requiring the automaton to arrive in a final state with empty input and completed output. Note that this is also, in essence, the basis of the Needleman-Wunsch-Sellers dynamic programming algorithm [9, 10], as can be seen more clearly by way of an exactly analogous definition for a matrix interpretation of distance:

Definition 2.7 (Matrix Edit Distance) *Given a weighted editor $\mathcal{E}$ and a pair of strings $x, y \in \Sigma^*$, for each state $q_i \in Q$ construct a matrix $q_i[0..|x|, 0..|y|\,]$ of reals such that*

- *for each final state $f \in F$, $f[0,0] = 0$*
- *for $0 \le a \le |x|$ and $0 \le b \le |y|$,*

$$q_i[a,b] = \min \{\, q_j[a - |u|, b - |v|\,] + n \mid \langle q_i, u, v, n, q_j \rangle \in \delta,\ u = x_{(a-|u|+1)..a} \text{ and } v = y_{(b-|v|+1)..b} \}$$

Then the matrix edit distance *between x and y is $s[\,|x|, |y|\,]$.*

Relating these distances to alignment is awkward because of the distinction between input and output. We can require that the output generated match a given string, but it is more natural and ultimately more useful to reorient our view of the automaton with one final construction:

Definition 2.8 (Aligner) *An* aligner $\mathcal{A}_\mu$ *is a (weighted) two-tape finite transducer constructed from a (weighted) editor $\mathcal{E}_\mu$ as follows:*

- *the states Q, start state s, and final states F are as in $\mathcal{E}_\mu$*
- *the input alphabet is that of $\mathcal{E}_\mu$, $\Sigma = \Sigma_\mu$*
- *the output alphabet Ω is a new set of symbols in a bijective mapping from transitions in $\mathcal{E}_\mu$, $a : \delta \to \Omega$*
- *for each transition $d = \langle q_i, x, y, (n,)\, q_j \rangle$ in $\mathcal{E}_\mu$ there is a $\langle q_i, x, y, z, (n,)\, q_j \rangle$ in $\mathcal{A}_\mu$ such that $a(d) = z$*

An aligning *is any derivation by an aligner, and an* alignment *is an output of an aligner.*

By default and by convention, the mapping a for transitions with input pairs $x, y \in \Sigma_\mu \cup \{\epsilon\}$ will simply be a token with the inputs written one above the other, using a dash in the case of ϵ. Thus, for the transitions of the three simple mechanisms of Figure 2, we would have $a(\langle s, x, y, f \rangle) = \frac{x}{y}$, $a(\langle s, x, \epsilon, f \rangle) = \frac{x}{-}$, and $a(\langle s, \epsilon, y, f \rangle) = \frac{-}{y}$. It can be seen that formal alignments will thus correspond

to alignments as they are usually understood. Not only is alignment thus explicitly related to editing (as well as mutation, etc.), but to the definition of minimal edit distance as well.

We emphasize that this also constitutes a declarative computational model, and that we may infer recurrences directly from the structures of the models. For example, viewing the states in Figure 5b this time as matrices according to Definition 2.7, we derive the following recurrences by minimizing over all outgoing transitions, decrementing indices by the lengths of the inputs consumed and incrementing the cost by their attached weights (with appropriate conditions applied):

$$s[a,b] = \min \begin{cases} s[a-1,b-1] & \text{if } x_a = y_b \\ s[a-1,b-1] + \sigma & \text{if } x_a \neq y_b \\ d[a,b] & \\ i[a,b] & \end{cases} \qquad d[a,b] = \min \begin{cases} s[a-1,b] + \alpha \\ d[a-1,b] + \beta \end{cases}$$

$$s[0,0] = 0 \qquad i[a,b] = \min \begin{cases} s[a,b-1] + \alpha \\ i[a,b-1] + \beta \end{cases}$$

together with other boundary conditions that are easily inferred. These, in fact, are exactly the recurrences for affine gaps derived by Gotoh algebraically (and rather less concisely) [4].

To review and recapitulate this methodology, we consider yet another model of indels that takes potential reading frames into account. Our goal will be a discontinuous distance function, defined for gap length k as $\gamma_k = \alpha_{k \bmod 3} + \beta \cdot k$, where $\alpha_0 \ll \alpha_1 = \alpha_2 = \alpha$. That is, indels that would maintain a reading frame will be penalized less than those that would disrupt it. We begin with a mechanism of deletion conforming to this notion, as shown in Figure 6a. We create successive states for each individual base deletion, in a loop of length three, and from each deletion state allow the deletion to terminate with penalty α, unless the deletion is a multiple of three in size (we use $\alpha_0 = 0$). We can incorporate this mechanism into an editor according to Definition 2.4 as shown in Figure 6b, along with a simple substitution mechanism; insertion is not shown but is analogous to deletion.

As before, we can merge start and final states to produce the equivalent editor of Figure 6c. By examination, it is obvious that state d_0 can also be merged with the start/final state: its only outgoing transitions are a free move to that state, and a deletion move to d_1 which s already possesses. Thus we arrive at Figure 6d, and again can mechanically and non-algebraically produce a recurrence:

$$s[a,b] = \min \begin{cases} s[a-1,b-1] & \text{if } x_a = y_b \\ s[a-1,b-1] + \sigma & \text{if } x_a \neq y_b \\ d_1[a-1,b] + \beta & \\ i_1[a,b-1] + \beta & \end{cases} \qquad d_1[a,b] = \min \begin{cases} s[a,b] + \alpha \\ d_2[a-1,b] + \beta \end{cases}$$

$$s[0,0] = 0 \qquad d_2[a,b] = \min \begin{cases} s[a,b] + \alpha \\ s[a-1,b] + \beta \end{cases} \qquad \text{etc.}$$

Thus, great flexibility in designing gap weights can be achieved through the use of additional states. Of course, this comes at the cost of adding new matrices, and because there are finite states it is not possible to model forms such as concave gap functions [7] without additional features, e.g. a pushdown automaton would be required to model inversions. However, we *can* use this framework to incorporate certain "meta" models of alignment in a very intuitive manner. For example, practical alignment algorithms used for purposes of detecting sequence similarities differ from those shown above in several respects. First, they are constructed so as to maximize similarity rather than minimize distance (see [12] for a discussion), and this is easily accomplished by reconsidering the weights on arcs and by substituting *max* functions in Definition 2.7. More importantly, the best-fitting *substrings* of the inputs are desired, for a so-called *local alignment*. To achieve such variations the model is formally augmented with new types of zero-weighted scanning moves that act on either but not both input strings and produce empty alignment output, and which may be inserted into existing models in a variety of series and parallel configurations for the desired effect. Space does not permit a full development of this technique, but as an example Figure 7a shows an equivalent aligner for the simple series-scanned case, that produces local alignment.

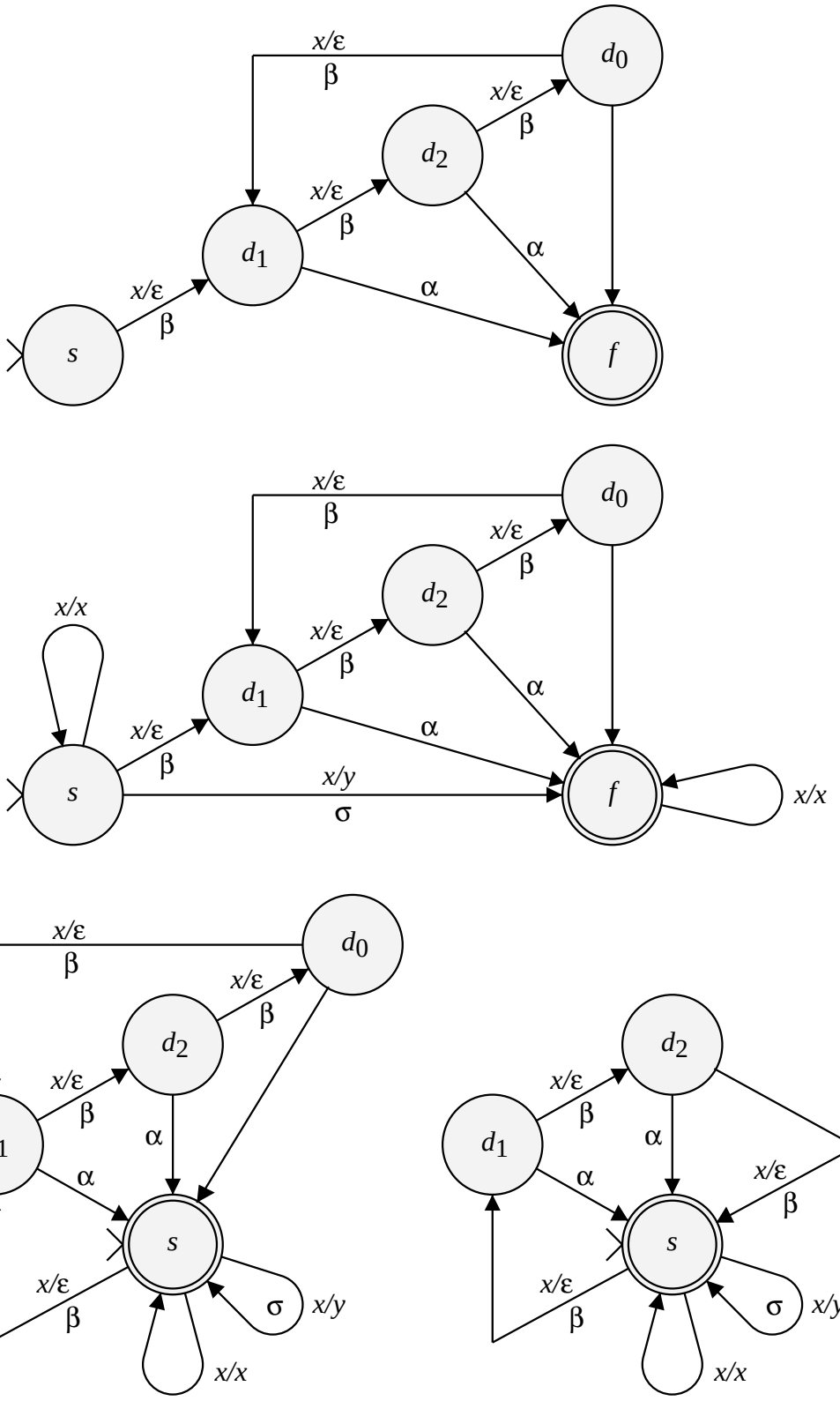

Figure 6: (a) Weighted mechanism for frame-sensitive deletion (top), (b) an editor with substitutions as well (middle), (c) an equivalent editor with merged start and final states (bottom left), and (d) a simplified editor (bottom right). Insertions are omitted for economy, but would simply mirror the deletion apparatus.

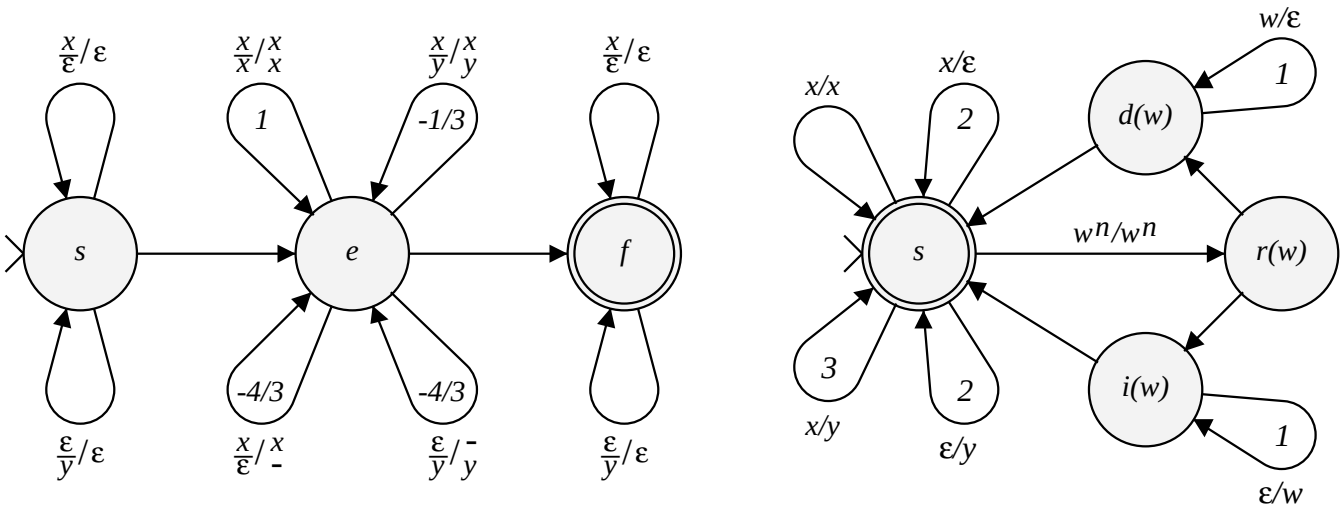

Figure 7: (a) Equivalent aligner for local alignment (left), and (b) an editor for variable numbers of tandem repeats, where $w \in \Sigma^*$, $|w| \leq 4$, and $n \approx 20$ (right).

We emphasize that this aligner has empty output on the reflexive transitions on s and f, so that only the local alignment would be displayed, in the customary fashion. Again, we can produce recurrences by direct examination of the automaton:

$$s[a,b] = \max \begin{cases} s[a-1,b] \\ s[a,b-1] \\ e[a,b] \end{cases} = \max_{\substack{0 \leq i \leq a \\ 0 \leq j \leq b}} e[i,j]$$

$$f[a,b] = \max \begin{cases} f[a-1,b] \\ f[a,b-1] \end{cases} = 0$$

$$e[a,b] = \max \begin{cases} e[a-1,b-1] + 1 & \text{if } x_a = y_b \\ e[a-1,b-1] - 1/3 & \text{if } x_a \neq y_b \\ e[a-1,b] - 4/3 \\ e[a,b-1] - 4/3 \\ f[a,b] \ (= 0) \end{cases}$$

The simplifications given for the recurrences of s and f also follow by reasoning from the automaton. We know that $f[a,b] = 0$ for any a, b because all out-transitions from f have zero weight, and any inputs can be emptied to achieve the conditions for termination. Similarly, s permits any prefixes of the inputs to be consumed with zero weight, so that the maximum weight from any position on the inputs is simply the maximum of zero and the result of the free transition from there to e (i.e. the maximum value in the matrix of e) which we have seen is at least zero. These are the same equations derived by Smith and Waterman [11]. Other uses of meta-alignment scanning nodes include best-fit alignments that specify containment or overlap as required by fragment assembly algorithms.

As a final example of more sophisticated gap models we present one in which the gap penalty is context-dependent — dependent, that is, on what lies across from the gap. The editor in Figure 7b is based on the observation that regions of the genome with multiple short tandem repeats are observed to exhibit great variability in the number of those repeats [8]. In comparing these highly polymorphic segments, an alignment algorithm would be more effective if it penalized gaps within such VNTRs much less than it did gaps in other sequence, since such gaps are known to arise more frequently than elsewhere. The editor shown implements this idea *via* the move that recognizes such matching repeats w over a certain threshold number n, and then allows additional such repeats to be inserted or deleted at much reduced cost (this being a *min* machine). The repeat w is shown as a parameter on the VNTR mechanism nodes so that the identity of a repeat may be preserved through subsequent transitions, but again this is just shorthand for a more complex (though still finite) automaton with instantiated inputs. Note that, for both the machines in Figure 7, affine gaps or indeed any other models of gaps could be substituted for the ordinary gap models illustrated, and in fact this technique makes it easy to combine aspects of various models.

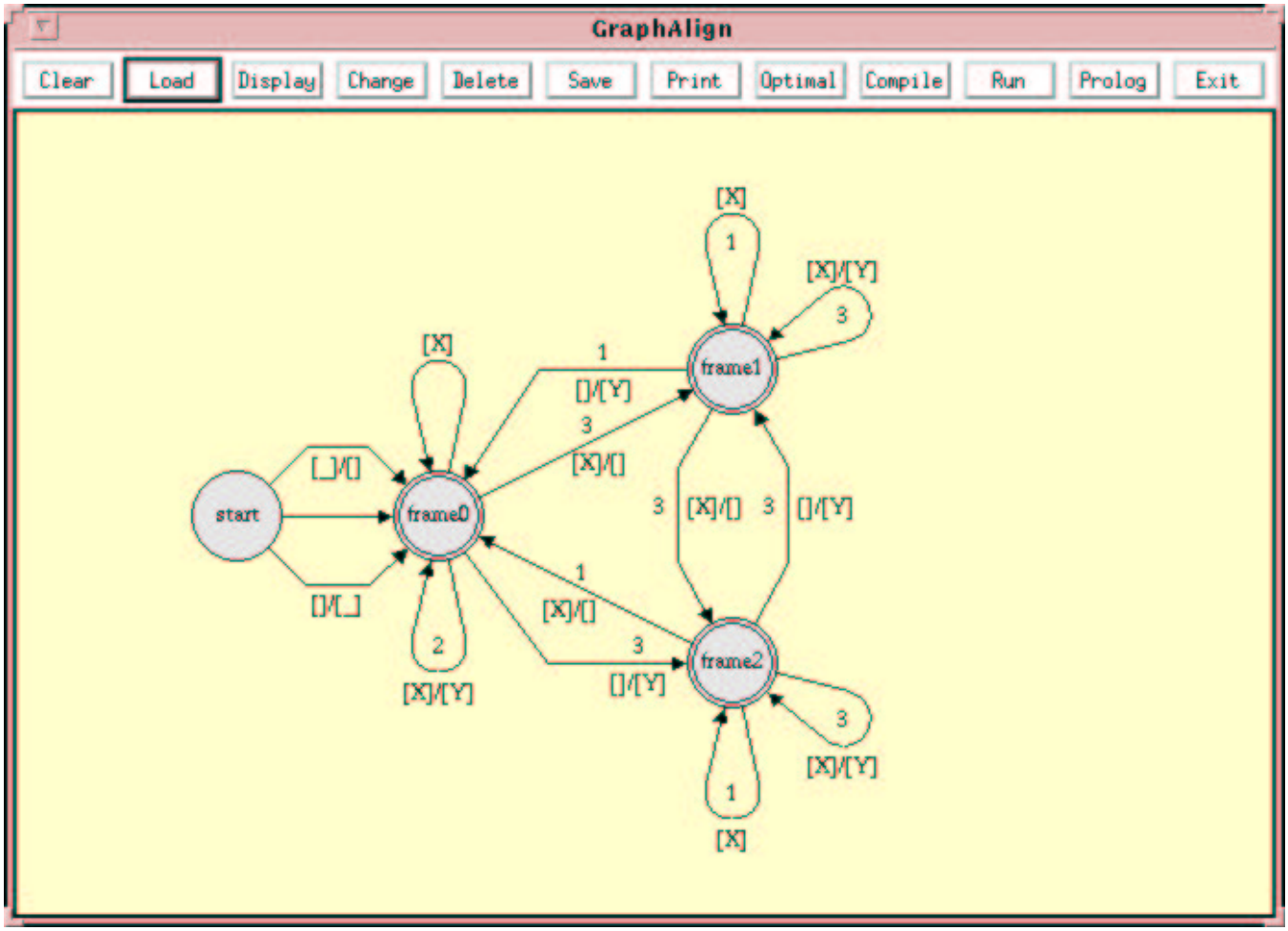

Figure 8: Frame-maintenance aligner, as entered into the graphical user interface.

3 Visual Programming

We have recently begun to implement a system making practical use of this methodology. A Prolog program was written that takes as input a specification of an aligner, given as a database of appropriately-labelled and indexed nodes and transitions. The program translates this to Prolog code, essentially using the constructions given above to (1) create a matrix for each node to store not only aggregated weights but also traceback information, and (2) generate declarative code to implement the appropriate recurrence. The Prolog code generated uses foreign function calls to dynamically-allocated 'C' arrays for efficiency, and we observe true quadratic-time behavior, although the recursive overhead in the current version limits speeds to the range of 1–10 msec per matrix cell.

We have hidden the logic-based code generator within an easy-to-use graphical interface, shown in Figure 8 — in effect, a domain-specific visual programming system. The interface is designed as a specialized drawing tool, with which the modeller may use the mouse to deposit or adjust the labelled nodes and arcs (including reflexive arcs) of a finite transducer. The Motif-based drawing tool has all the features found in similar programs, such as "snapping" of objects to a grid to ensure a neat appearance, and the ability to add and adjust articulation points to linear arcs. However, the drawing tool is connected to the underlying logic that will, at the press of a button, generate and execute a dynamic programming algorithm specified by the automaton on the screen. For example, with this model the "Run" button produces the following behavior in a dialog window:

```
Enter 1st sequence: ttaggcttatgcgattcgttatgcggtatgcttagctttaggcgttatgcgggatc
Enter 2nd sequence: tattcgggcttatgtcggcggattctgagtcggtactttacttattcggatctatg
Enter start state: start
Running alignment...
Alignment completed in 6.313 ms per cell, total value 68:

     t-t--aggcttatgcgattcgttatgcggtatgcttagctttaggcgttatgcggg---atc
     |-+-- ||||||||      | |     ||  |--++-|||||--+-||||  ||| ---||
     tattcgggcttatgtcggcggattctgagtcgg--ta-cttta--c-ttattcggatctatg
```

We note several things about the implementation: First, the labels on transitions are given as uppercase logic variables within square-bracketed lists, conforming to Prolog notation. Where a single input is indicated (e.g. `[X]`) it is understood that both inputs are the same, and otherwise different. Although the machine is portrayed as an editor, it is actually an aligner, with the output entered separately and ordinarily not shown. In specifying the aligner outputs, the user may indicate both an upper and lower output (generally single characters) as well as a character to appear between the aligned elements. In the alignment shown, a blank between the rows of characters indicates a mismatch, a hyphen indicates a gap, a vertical bar indicates a match that is "in frame", and a plus indicates a match that is "out of frame", in the following sense: While the frame-sensitive machine derived in Figure 6 implemented a gap penalty model in which frame-preserving gaps were penalized less than others, it did not allow for cases where a gap that is not a multiple of three nevertheless *restores* the proper reading frame. Moreover, that model does not penalize the length of out-of-frame sequence downstream from a frame-disrupting gap. The model of Figure 8 maintains the notion of frame, however, so that both matches and mismatches in the putatively correct `frame0` are penalized less than in other frames, and it is possible for the correct frame to be restored by subsequent frameshifts.[2] Note also that the user may specify any node as the start state; the node named *start* is specifically designed to allow for any initial frame orientation in the input strings.

Because Prolog still provides top-down control for the algorithms thus generated, they are properly classified as a form of dynamic programming termed *memoization* [3]. We have investigated whether greater speedups could be obtained by generating true bottom-up, iterative code at the back end of the system. Indeed, an implementation in which the algorithm specification was translated instead to C++ resulted in two orders of magnitude speedup; however, a memoized version in C++ was only a factor of two slower than this, indicating that it is the recursive overhead in Prolog that is the major bottleneck in the initial prototype.

Acknowledgements

This work was supported by the Department of Energy Office of Energy Research, under grant number DE-FG02-92ER61371 to D.B.S. The authors thank Chris Overton and Kyle Hart for helpful discussions and suggestions, and Mike Yasayko for implementing a C++ prototype.

References

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing.* Printice-Hall, Englewood Cliffs, NJ, 1972.

[2] L. Allison, C.S. Wallace, and C.N. Yee. Finite-state models in the alignment of macromolecules. *J. Mol. Evol.*, 35(1):77–89, 1992.

[2]The derivation of the novel and elegant recurrence associated with this model is left as an easy exercise, given the methodology presented in the previous section.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, Cambridge, MA, 1989.

[4] O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162:705–708, 1982.

[5] R.M. Karp and M. Held. Finite-state processes and dynamic programming. *SIAM J. Appl. Math.*, 13(3):693–718, 1967.

[6] A. Krogh, M. Brown, I.S. Mian, K. Sjolander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *J. Mol. Biol.*, 235(5):1501–1531, 1994.

[7] W. Miller and E.W. Myers. Sequence comparison with concave weighting functions. *Bull. Math. Biol.*, 50:97–120, 1988.

[8] Y. Nakamura, M. Leppert, P. O'Connell, R. Wolff, T. Holm, M. Culver, C. Martin, E. Fujimoto, M. Hoff, E. Kumlin, and R. White. Variable number of tandem repeat (VNTR) markers for human gene mapping. *Science*, 235:1616–1622, 1987.

[9] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48:443–453, 1970.

[10] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26(4):787–793, 1974.

[11] T.F. Smith and M.S. Waterman. Identification of common molecular sequences. *J. Mol. Biol.*, 147:195–197, 1981.

[12] T.F. Smith, M.S. Waterman, and W.M. Fitch. Comparative biosequence metrics. *J. Mol. Evol.*, 18:38–46, 1981.