Optimal Alignments in Linear Space using Automaton-derived Cost Functions (Extended Abstract)

Submitted to CPM'96

Kevin P. Murphy*

9 December 1995

Abstract

In a previous paper [SM95], we showed how finite automata could be used to define objective functions for assessing the quality of an alignment of two (or more) sequences. In this paper, we show some results of using such cost functions. We also show how to extend Hischberg's linear space algorithm [Hir75] to this setting, thus generalizing a result of Myers and Miller [MM88b].

1 Introduction

An *alignment* of two sequences $x = x_1x_2 \dots x_m$ and $y = y_1y_2 \dots y_n$ is a way of writing them down, one on top of the other, such that in each column, there are either two similar characters, or a character and a space (denoted '-', representing the empty string). An *optimal* alignment is one which minimizes a cost function, or equivalently, which maximizes a similarity function.[1] The cost of an alignment is usually defined as the sum of the costs of each column in the alignment. We shall call this the Needleman-Wunsch [NW70] cost function. The crucial thing to notice is that the cost of a column is independent of the context in which it occurs. It is commonly recognised that this is an unrealistic assumption for applications in molecular biology. For example, the cost of an indel (insertion or deletion) depends on whether we have just performed an indel. We can model context-sensitive cost functions by introducing the notion of state. For example, the first time we perform a deletion, we enter a "delete" state; subsequent delete operations performed in this state are "cheaper" than normal. We make a transition between states every time we perform a mutation (edit) operation.

A very convenient way to represent context-sensitive cost functions is as a finite transducer [SM95], that is, as a finite automaton whose transition arcs are labelled by the operations they represent. We have developed a Graphical User Interface that allows the user to interactively create an automaton, which implicitely defines an objective function f. A stand-alone program is then automatically generated which can compute alignments of two strings x, y which are optimal with respect to f, using dynamic programming (DP). (This is called "compiling" the automaton.) The naive DP method takes quadratic time and space. In this paper, we show how to extend Hirschberg's algorithm [Hir75] to do this in linear space. This generalizes a result of Myers and Miller [MM88b], who show how to do this in the case where f allows for affine gap costs. We also include results which illustrate the biological utility of context-sensitive cost functions.

*murphyk@cs.ucdavis.edu

[1] In this paper, we shall only consider global alignments and will speak consistently of minimizing costs. However, all of the results apply to the maximization case, which is more appropriate for local alignment. See [SM95].

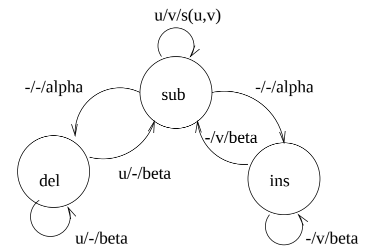

Figure 1: The affine gap machine. The start and final state is "sub". The label u/-/beta denotes that any single character u can be replaced by a space ('-') at a cost of β. Similarly, the label $u/v/s(u,v)$ denotes that u can be replaced with v at a cost of $s(u,v)$.

1.1 Notation

An example of a finite automaton is given in Figure 1. This encodes the well-known affine gap cost function, which specifies that the cost of a gap (a contiguous run of indels) of length k is $\gamma(k) = \alpha + \beta k$, where α is the cost of "opening-up" a gap, and β is the cost of continuing it. The cost of a substitution is $s(u,v)$. We will now illustrate how an automaton defines a cost function in general.

We associate a 2-dimensional array q_k with the k'th state, with the interpretation that $q_k(i,j)$ is the least cost of aligning $x_1 \ldots x_i$ and $y_1 \ldots y_j$, starting in the start state, and ending in state k. We can compute this by looking at the cost of aligning some initial prefix of $x_1 \ldots x_i$ and $y_1 \ldots y_j$ and ending up in a neighboring state, say k', plus the cost of the transition from k' to k. We use $k' \xrightarrow[c]{u/v} k$ to denote a transition from state k' to state k which costs c and has label u/v; this label means that u must match the unaligned portion of $x_1 \ldots x_i$ and v must match the unaligned portion of $y_1 \ldots y_j$. (It is useful to imagine the automaton consuming characters from the input tapes x and y as it traverses arcs.) Hence we can compute $q_k(i,j)$ by examining the cost of all matching incoming arcs, plus the cost of getting to the states these arcs emanate from:

$$q_k(i,j) = \min_{k'} \left\{ q_{k'}(i - |u|, j - |v|) + c : \ k' \xrightarrow[c]{u/v} k, \quad u = x_{i-|u|+1 \ldots i}, \quad v = y_{j-|v|+1 \ldots j} \right\}$$

For the affine gap machine, we get the following familiar set of simultaneous equations (disregarding boundary conditions):

$$\begin{aligned} q_s(i,j) &= \min\{q_s(i-1,j-1) + s(x_i,y_j), \ \ q_i(i,j-1) + \beta, \ \ q_d(i-1,j) + \beta\} \\ q_i(i,j) &= \min\{q_i(i,j-1) + \beta, \ \ q_s(i,j) + \alpha\} \\ q_d(i,j) &= \min\{q_d(i-1,j) + \beta, \ \ q_s(i,j) + \alpha\} \end{aligned}$$

If we let the start state have index s and the final state have index f, then the initial condition is $q_s(0,0) \stackrel{\text{def}}{=} 0$, and the goal is to compute $q_f(m,n)$.

We allow the labels on arcs to be arbitrary combinations of string constants and variables. (We will see the utility of this later.) If the same variable is repeated, it must be bound (instantiated) in the same way when it is matched. For example, the label "XYX/XZX" will match the substrings "aba/aca", since 'X' is bound consistently to 'a', but will not match "abd/acd", "aba/acd", etc. (where we have used upper case letters for variables, and lower case for constants.) These are called *parameterized patterns* [Bak93]. When compiling an automaton, it is simple to generate side conditions which specify when an arc can be traversed;

if it cannot, we assign the arc an infinite cost. In the case of the affine gap machine, the labels are either the empty string, which always matches, or a single variable, which matches provided there is at least one character left in the string, i.e., $i > 1$ or $j > 1$ as appropriate. For example, we can rewrite the above equations to take into account boundary conditions as follows:

$$q_s(i,j) = \min\{q_s(i-1,j-1) + (i > 1 \wedge j > 1 \ ? \ s(x_i, y_j) \ : \ \infty), \ \ q_i(i,j-1) + \beta, \ \ q_d(i-1,j) + \beta\}$$

etc., where we have used the conditional operator syntax "predicate ? result-if-true : result-if-false" from C. When the labels contain string constants, the side conditions can become very messy, further illustrating the benefit of using an automaton to specify the cost function.

1.2 Semantics

Karp and Held [KH67] refer to the kind of automaton we have (informally) introduced above as an *additive sequential decision process*. This is a special kind of *discrete decision process*, in which the aim is to choose a sequence of actions (a policy) which minimizes some cost function, for a given set of parameters, e.g., input strings and arc weights (costs). Unfortunately, it is not clear what this implies about the kind of string alignment cost functions which can be represented by automata. What is clear is that, with $2k+1$ states, it is possible to represent any k-piece-wise linear cost function as a natural extension of the affine gap model (see e.g, [AWY92]). However, it does not seem to be possible to represent convex gap costs, for example, for which the best known algorithm runs in $O(nm \log n)$ time [MM88a].

Another important question is: what do the arc weights mean, and where do they come from? The most natural interpretation is that they represent the negative log probability of the operation on the arc they are attached to, scaled up to be integer-valued [BT86]. Allison, Wallace and Yee [AWY92] use the EM algorithm to learn the transition probabilities for the affine gap machine.[2] They start with arbitary initial values, and iterate until the message length of the resulting alignment is (locally) minimal. The message length of an alignment, under the r-hypothesis that the two strings are related, is the number of bits needed to transmit one string, plus all the parameters of the automaton, which can be used to generate the other string. This is compared with the message length of an alignment under the null hypothesis, which assumes the strings are not related; this requires $2(m+n)$ bits to encode the two strings (assuming an alphabet of size 4), plus a few more to encode their lengths. If the r-hypothesis leads to a significantly shorter encoding, the strings are assumed to be related by the specified automaton. Interestingly, Allison et al. point out that Hirschberg's linear space algorithm is not applicable in their case, since they are not interested in a single optimal alignment, but a probability distribution over all alignments, which takes quadratic space to represent.

2 Optimal alignments as a shortest paths problem

It is well known that computing an optimal alignment with respect to the standard Needleman-Wunsch cost function is equivalent to finding the shortest path through the edit graph. Using an automaton to define the cost function simply extends this to three dimensions, as shown in Figure 2. Now we have to find the shortest path from the point $(s, 0, 0)$ to (f, m, n), where the path can proceed along any of the "vectors" defined by the transitions of the machine, e.g., the transition $k \xrightarrow[c]{u/v} k'$ defines the (set of) vectors joining (k, i, j) to $(k', i - |u|, j - |v|)$ for all $i \geq |u|, j \geq |v|$. For the Needleman-Wunsch one-state machine, all the

[2] The main problem in learning the transition probabilities is in getting enough data to avoid over-fitting. It would be interesting to investigate using priors for this cf. [BHK+93].

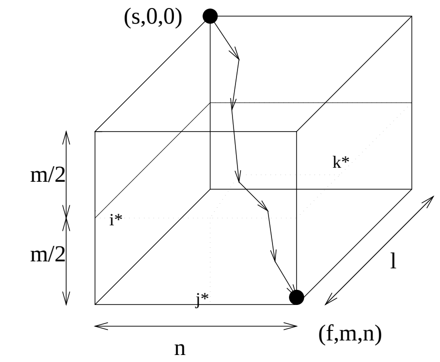

Figure 2: Computing an optimal alignment is equivalent to finding the shortest path from $(s, 0, 0)$ to (f, m, n). In the linear space algorithm, we compute the midpoint of this optimal path, (k^*, i^*, j^*), and then recursively compute the optimal alignments of each half. The vertical axis is labelled by x, the horizontal axis by y, and the axis into the page denotes the l different states in the machine.

transitions are labelled as $u/-$, $-/v$ or u/v, corresponding to vertical, horizontal and diagonal unit-length edges respectively. The affine gap machine adds edges between the planes which represent the different states. In general, the set of legal paths through the graph may be quite complex, since some transitions may be labelled by string constants, and hence some edges may only be present opposite the appropriate substrings in x and y. Also, some of the edges may have non-unit length.

There are two main ways to solve problems of this kind: in a top-down or "push you" sense using memoization, or a bottom-up, "pull me" sense using dynamic programming. The top-down method is more efficient when the cost matrices are sparse, since values are computed only when needed: to compute $q_f(m, n)$, we might only need to know the values of a small number of interior cells, so their values (and only their values) are (recursively) computed. Some examples of automata which give rise to sparse matrices are given later. The bottom-up method is more efficient when the matrices are dense (as they usually are), since there is no overhead from the recursive calls. More importantly, it is possible to compute an optimal alignment in linear space by using the bottom-up method in a divide-and-conquer fashion, as we now show.

3 The linear space method

In the case of the affine gap machine, it is clearly possible to compute the *cost* of an optimal alignment in $O(n)$ space, since to compute the cost of an entry, we only need to know the values of its immediate predecessors, and hence it is sufficient to keep two row vectors per state, one for the current row and one for the previous. In general, we define the *depth* of a state, $d(q)$, as the number of rows it needs to keep. This can be computed by looking at the longest x-pattern of every incoming arc:

$$d(q) = 1 + \max_{q'} \{|u| : q' \xrightarrow[c]{u/v} q\}$$

For example, in the affine gap machine (see Figure 1), $d(ins) = 1$, $d(del) = 2$, and $d(sub) = 2$, so the total storage space is $5(n + 1)$ cells. (A more careful analysis, in [MM88b], reduces this to $2(n + 1) + 3$; however, here we want to concentrate on general techniques, which can be applied to all automata.) For simplicity of implementation (to keep all the arrays the same "height"), we define $d = \max_q d(q)$, and allocate a $d \times (n+1)$ array for each state. In this context, the notation $q_k(i, j)$ is shorthand for $q_k(i \bmod d, j)$; in other words, we can think of each state as having a depth d circular buffer (of width $n+1$).

We can compute an optimal alignment (and not just its cost) in linear space using the following divide and conquer technique. Suppose $m > 1$ and let $i^* = \lfloor m/2 \rfloor$, so that row i^* is the middle row of the cost matrices.[3] We compute the cost of an optimal path from $(s, 0, 0)$ to (k, i^*, j), and from (k, i^*, j) to (f, m, n), for all states k and all columns j (where s is the start state and f is the final state). We claim that the optimal path from $(s, 0, 0)$ to (f, m, n) must pass through (one of) the point(s) (k^*, i^*, j^*), where

$$j^*, k^* = \arg\min_{j,k} \{q_k(i^*, j) + q_k^R(m - i^*, n - j)\}$$

and, as before, $q_k(i,j)$ is the least cost of getting from $(s, 0, 0)$ to (k, i, j), and $q_k^R(m-i, n-j)$ is the least cost of getting from (k, i, j) to (f, m, n). (In the case of ties, we can take the leftmost or rightmost j^* in order to bound the range of optimal alignments.) We now describe how to compute these reverse costs q^R.

Let $\mathcal{M}_{sf}(x, y)$ denote the cost of the optimal alignment between x and y using the automaton $\mathcal{M}$, starting in state s and finishing in state f. Then $q_k(i,j) = \mathcal{M}_{sk}(x_1 \ldots x_i, y_1 \ldots y_j)$. Now let $\mathcal{M}^R$ denote the same machine except that the directions of all the arcs (and their labels) are reversed. (For example, the reversed affine gap machine interchanges the role of the silent and α-charging arcs between "sub" and "del/ins", so that now α is better thought of as the cost of "closing" a gap.) Analogously, we have $q_k^R(m-i, n-j) = \mathcal{M}_{fk}^R(x_m \ldots x_i, y_n \ldots y_j)$. The intuitive reason for this is that the optimal path from (k, i, j) to (f, m, n) must proceed along a series of vectors defined by $\mathcal{M}$; the cost of this path is the same as the cost of the path in which we start at (f, m, n) and traverse the same vectors backwards.

Having identified one of the points on the optimal path, we recursively solve the subproblems of finding an optimal path from $(s, 0, 0)$ to (k^*, i^*, j^*), and from (k^*, i^*, j^*) to (f, m, n). It is clear that, in general, we will need to be able to compute optimal paths between any two arbitrary points. This can be easily accomplished by redefining which states are the start and final states, as necessary. This generalises what Myers and Miller call "type 1" and "type 2" midpoints, corresponding to the cases where k^* is the `sub` state and the `del` state of the affine gap machine, respectively. They handle the case where the midpoint is in the middle of a gap by subtracting off the gap-start cost α from the cost of the path from the midpoint to the end. We don't need to worry about this explicitely, since in the reversed affine gap machine, the arc from the final state ("sub") to the delete state is not charged α for a silent transition (one with label $-/-/c$ for some c).

The base case is when $m \leq 1$, when we perform the usual bottom-up algorithm, storing the optimal moves at each point, so that we can traceback to produce the alignment. Note that, even if $m = 0$, the optimal alignment may traverse a complicated path in the $i = 0$ plane to get from $(s, 0, 0)$ to $(f, 0, n)$ (corresponding to following a series of $-/v$ arcs), so we need to store pointers explicitely in an auxiliary array (of $O(n)$ size, of course). This part of the traceback is then printed out so we can subsequently re-use the space.

If it takes $T = cmn$ time to compute the cost of the optimal alignment, it takes $2T$ time to compute the optimal alignment itself in linear space, since we are halving the amount of work we do at each step: $t = cmn + 1/2(cmn) + 1/4(cmn) + \cdots \leq 2cmn$. The total amount of space taken is $O(n + \log m)$, since there are $O(\log_2 m)$ recursive subcalls.

In summary, the overall algorithm is shown in Figure 3. The procedure `compute-cost` obviously depends on the specific machine, but in Figure 4 we give the pseudo-code for the affine gap machine. The basic idea is to associate a function with each state which is responsible for computing $q_k(i,j)$ given its predecessors in

[3] If the automaton has patterns which are more than one character long, some of its vectors will be non-unit vectors, and hence some parts of the matrix will not be accessible. This is a problem if i^* bisects one of these vectors. However, we can always convert such a machine to a form in which each pattern is a single character, by adding new states and transitions if necessary.

```
proc align(x, y, start, final)
{
 if |x| ≤ 1
  then compute − cost (M, x, y, start)  storing optimal moves
      print traceback
 else
  i* := ⌊|x|/2⌋
  compute − cost (M, x_1 ... x_{i*}, y_1 ... y_n, start)  //computes the q array
  compute − cost (M^R, x_m ... x_{i*}, y_n ... y_1, final)  //computes the q^R array
  j*, k* := arg min_{j,k} {q_k(i*, j) + q^R_k(|x| − i*, |y| − j)}
  align(x_1 ... x_{i*}, y_1 ... y_{j*}, start, k*)
  align(x_{i*} ... x_m, y_{j*} ... y_n, k*, final)
}
```

Figure 3: The main procedure of the linear space algorithm.

```
proc compute − cost (M, x, y, start)
{
 //(0,0)
 q[start, 0, 0] := 0
 if (start ≠ sub) then sub(0,0);  if (start ≠ ins) then ins(0,0);  if (start ≠ del) then del(0,0)
 // Rest of row 0
 for j := 1 to |y|
  sub(0, j);  ins(0, j);  del(0, j)
 // Rest of array
 for i := 1 to |x|
  for j := 0 to |y|
   sub(i, j);  ins(i, j);  del(i, j)
}
```

Figure 4: The procedure for computing the cost matrices.

the matrix, and then to call these functions in the appropriate order; we will discuss what the "appropriate order" is below. In practice, these functions would be inlined. Finally, we give an example of how to compute the cost function associated with a specific state in Figure 5. In practice, we would avoid the bounds checking on i and j when we know that this function is being called from an interior cell.

```
proc sub(i, j)
{
 c[0] := (j ≥ 1) ? q[ins, i mod depth, j − 1] + α  :  ∞
 c[1] := (i ≥ 1) ? q[del, (i − 1) mod depth, j] + α  :  ∞
 c[2] := ((i ≥ 1)  and  (j ≥ 1)) ? q[sub, i − 1, j − 1] + S(x(i), y(j))  :  ∞
 q[sub, i, j] := min_{0≤k≤2} c[k]
}
```

Figure 5: The procedure for computing a single entry in the cost matrix associated with the "sub" state.

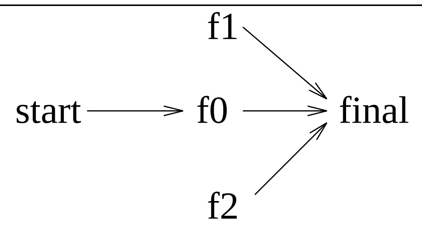

Figure 6: The dependency graph for the frameshift machine in Figure 7. The graph indicates that start must be computed before f0, and f0, f1 and f2 must all be computed before final.

3.1 Computing the states in the appropriate order

As we have seen, the heart of the cost computation (where the program spends most of its time) involves computing the values of the cells in the interior of the matrix. This loop has the following form.

for $i := 0$ to m
  for $j := 0$ to n
    for $k := 0$ to $l - 1$
      $q(k, i, j) := \ldots$

What order should we compute the states in? For the affine gap machine, it is clear that the `ins` and `del` states "depend on" the `sub` state, and hence `sub` should be computed first, since the equations for state `del` and `ins` assume that the corresponding point (i, j) has already been computed in state `sub`. In general, we say state t depends on s if there is a silent arc from s to t. To compute the appropriate order, we construct a dependency graph G, with vertices equal to the states, and an (s, t) edge if there is a silent arc from s to t (indicating that s must be computed before t.) Then a topological sort of G will give us the required ordering. (This can be proved by induction.) See Figure 6 for an example. If G has a cycle, no linear ordering can be found, indicating that the machine has a silent cycle (a cycle consisting of all silent arcs). Such a machine is not allowed, lest it traverse the silent cycle infinitely often, resulting in a non-terminating computation.

3.2 Saving time by reducing the computational volume

One way of reducing the "computational volume" (the number of cells which must be evaluated, when computing the cost of an alignment) is to compute a lower bound on the cost of any path; if this exceeds some known upper bound on the cost of the optimal alignment, this path need not be explored. In this way, we can avoid computing regions of the matrices which lie too far from the main diagonal. This approach was pioneered in [Ukk85, Fic84], and generalised in [Spo89, Spo91], but only in the case of affine gap scoring functions. It is not clear how to automatically compute a lower bound on the cost of a path using an arbitrary scoring function, so this technique has not yet been implemented.

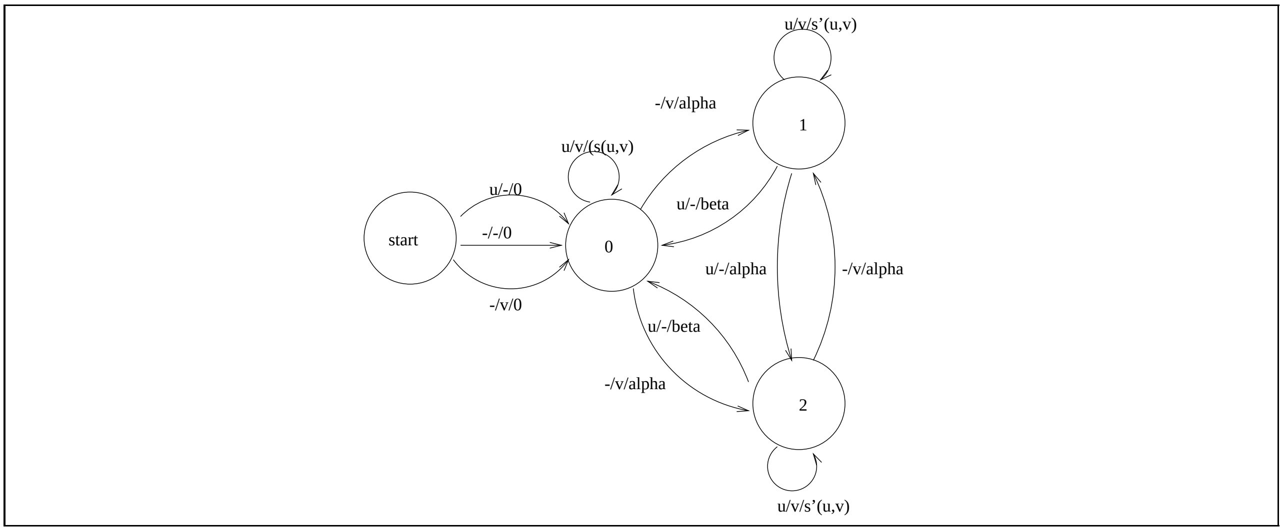

Figure 7: The frameshift machine. The state labelled "start" allows the reading frame to shift one position in either the x or y string before starting an alignment. 0 is assumed to represent the correct reading frame. There is a unique final state (not shown for simplicity), with silent transitions to it from states 0, 1 and 2, since the alignment may end in any reading frame. In-frame substitutions are charged less than out-of-frame substitutions ($s < s'$). Indels which disrupt the reading frame are charged more than indels that restore it ($\beta < \alpha$). If the "start" state is deleted, states 0, 1 and 2 have a density of $\frac{1}{3}$.

4 Experimental results

4.1 The frameshift machine

In Figure 7 we illustrate an automaton which takes into account reading frame when performing alignments. Essentially this automates the common practice of translating in all three reading frames, and aligning against all of them separately, and then jumping between frames to follow the strongest main diagonal so as to minimize the cost. This can be useful when aligning translated cDNA sequences with proteins. This is similar to the work of Guan and Uberbacher [GU95]; in the future, we will attempt to replicate their results.

4.2 The gene-finding machine

In Figure 8 we illustrate an automaton which can splice out introns "for free" when aligning coding sequences. This is similar to, but much simpler than, the Hidden Markov Model used to find genes in *E. Coli* [KMH93]. In the future we will investigate the biological utility of this.

4.3 The VNTR machine

In [SM95] we presented an automaton which can charges indels inside a VNTR (Variable Number of Tandem Repeats) region less than normal. In the future we will investigate the biological utility of this.

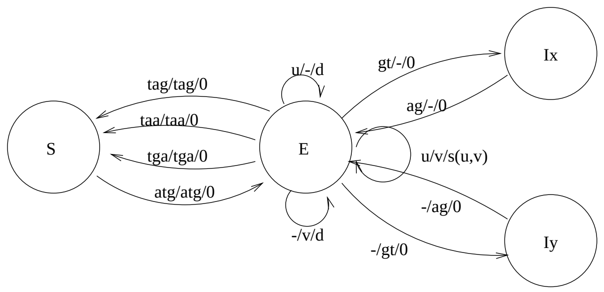

Figure 8: The gene-finding machine. The machine is in state S during non-coding regions, and transitions to state E when it sees a start codon, "atg", indicating the start of an open reading frame. When inside a coding region, it can transition to Ix or Iy when it sees a "gt", indicating (with high probability) the start of an intron. It can return to the exon state when it sees an "ag", indicating (again, with high probability) the end of an intron. Finally, it can return to S when it sees a stop codon (any of "tga", "taa", "tag"), indicating the end of an open reading frame. The resulting machine is appropriate for aligning eukaryotic coding sequences. In the exon state, indels are charged a fixed cost, d; we could obviously extend this to use affine gaps.

5 Acknowledgments

The author would like to thank David Searls and Dan Gusfield for their helpful suggestions. This work was supported in part by DOE grant DE-FG03-90ER60999.

References

[AWY92] L. Allison, C. S. Wallace, and C. N. Yee. Finite-state models in the alignment of macromolecules. *J. of Molecular Evolution*, 35:77–89, 1992.

[Bak93] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications (extended abstract). In *ACM Symposium on the Theory of Computing*, pages 71–80, 1993.

[BHK+93] M. P. Brown, R. Hughey, A. Krogh, I. S. Mian, K. Sjölander, and D. Haussler. Using dirichlet mixtures priors to derive hidden Markov models for protein families. In *International Conf. on Intelligent Systems for Molecular Biology*, pages 47–55, 1993.

[BT86] M. J. Bishop and E. A. Thompson. Maximum likelihood alignment of DNA sequences. *J. of Molecular Biology*, (190):159–165, 1986.

[Fic84] J. W. Fickett. Fast optimal alignment. *Nucleic Acids Research*, 12:175–180, 1984.

[GU95] X. Guan and E. C. Uberbacher. Alignment of dna and protein sequences containing frameshift errors. Technical Report ORNL/TM-12976, Computer Science and Mathematics Division, Oak Ridge National Laboratory, 1995.

[Hir75] D. S. Hirschberg. A linear space algorithm for computing longest common subsequences. *Comm. of the ACM*, 18:341–343, 1975.

[KH67] R. M. Karp and M. Held. Finite-state processes and dynamic programming. *SIAM J. of Appl. Math.*, 13(3):693–718, 1967.

[KMH93] A. Krogh, I. S. Mian, and D. Haussler. A hidden markov model that finds genes in *E. Coli* DNA. Technical Report USCS-CRL-93-33, U. C. Santa Cruz, Dept. Comp. Sci., 1993. Available from `ftp.cse.ucsc.edu`, directory `pub/dna`.

[MM88a] W. Miller. and E. Myers. Sequence comparison with concave weighting functions. *Bull. Math. Bio.*, 1988.

[MM88b] E. W. Myers and W. Miller. Optimal alignments in linear space. *Computer Applications in the Biological Sciences*, 4(1):11–17, 1988.

[NW70] S. B. Needleman and C. D. Wunsch. A general method applicable to the search fror the similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:443–453, 1970.

[SM95] D. B. Searls and K. P. Murphy. Automata-theoretic models of mutation and alignment. In *Proc. ISMB (Intelligent Systems for Molecular Biology)*, 1995.

[Spo89] J. L. Spouge. Speeding up dynamic programming algorithms for finding optimal lattice paths. *SIAM J. Appl. Math*, 49(5):1552–1566, 1989.

[Spo91] J. L. Spouge. Fast optimal alignment. *Computer Applications in the Biological Sciences*, 7(1):1–7, 1991.

[Ukk85] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.