Using Markov Models and Hidden Markov Models to Find Repetitive Extragenic Palindromic Sequences in *Escherichia coli*

Kevin Karplus

UCSC-CRL-94-24
26 July 1994

Board of Studies in Computer Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
karplus@cse.ucsc.edu

ABSTRACT

*This paper presents a technique for using simple Markov models and hidden Markov models (*HMMs*) to search for interesting sequences in a database of DNA sequences. The models are used to create a* cost map *for each sequence in the database. These cost maps can be searched rapidly for subsequences that have significantly lower costs than a null model. Milosavljević's algorithmic significance test is used to determine when a subsequence is significantly found. The sequences reported are trimmed to maximize the signal-to-noise ratio (cost savings / $\sqrt{length}$).*

Methods are given for automatically constructing simple Markov models and hidden Markov models from small training sets.

The techniques are illustrated by searching a database of E. coli genomic DNA, EcoSeq6, for clusters of Repetitive Extragenic Palindromic sequences (REPs). Of the known REPs, 91% are found with simple Markov models starting with a single REP cluster as a seed, and 95% are found by a hidden Markov model built from the results of the simple Markov model search. There are no false positives from the simple Markov models, and the few extra sequences found by the HMMs *may be genuinely related sequences.*

This paper describes a technique for using models suitable for data compression as a tool for finding interesting sequences in a database. The technique does not require that the structure or consensus sequence for the target sequences be known in advance—only that an example sequence be provided.

The paper presents an efficient algorithm for the search, methods for constructing the models automatically, and results of finding clusters of Repetitive Extragenic Palindromic sequences (REPs) in the *E. coli* genome database, EcoSeq6 [12]. It compares the set of REPs found with previous lists [7, 2, 13]. The new search techniques do as well as the best of previous techniques (self-BLAST), finding about 95% of the known REPs. Furthermore, the hidden Markov model produced for the search provides a good, human-readable description of the structure of the REP family, clearly distinguishing the two main REP sequences and the REPv variant.

Although the technique could be used for sequences over any alphabet, the software has only been implemented for DNA or RNA sequences so far.

1 Using compression models to find significant sequences

The problem of finding interesting sequences can be broken into several subproblems: defining what sequences are interesting, devising an algorithm to find them efficiently, and determining whether the sequences found are statistically significant or just chance variations.

The approach used here is to define the interesting sequences by using a model m that assigns probabilities to sequences ($P_m(s)$). The probabilities are assigned so that all the probabilities for sequences of a given length sum to 1—the length of the sequence is not predicted by the model. A sequence is interesting if the probability assigned by the model is significantly higher than the probability of the sequence using a null model. Normally we use the negative log probability of the sequence as a *cost* measure, since the probabilities become extremely small for longer sequences. Using base-2 logarithms gives us the encoding cost in bits of sequence s using model m:

$$cost_m(s) = -\log_2 P_m(s) \ .$$

Section 2 will talk about two ways that models can be constructed automatically, but first let's discuss how they are used for searching efficiently.

1.1 Efficient search using a cost map

The simplest algorithm for finding sequences using a model would be to enumerate all possible sequences and compute the cost of each. Unfortunately, this crude algorithm is too expensive. In a database of d characters, up to d^2 different sequences may be found. If computing the cost of a sequence takes time proportional to the length of the sequence, then the entire search algorithm would be order d^3. Furthermore, if several overlapping sequences have a low cost, we would like the algorithm to pick out the best of them—deferring for the moment exactly what we mean by "best".

Therefore, we need a search technique whose execution time is linear in d and which returns only disjoint sequences. First, let's assume that the model used to compute the probabilities provides not just an overall cost for a sequence, but assigns a cost to each position of the sequence. Second, let's assume that the cost of any given sequence is the sum of the costs for each of the characters of the sequence. Section 2.4 discusses the ways in which the models actually used violate these assumptions and why the violations are not serious in this context.

With the above assumptions, we can compute the costs for each character of the database once and save the costs in an array, called a *cost map*. From the cost map we can compute the cost for any subsequence of the database by taking the sum of costs in the array starting at the beginning of the subsequence and stopping at the end. Hence, given a cost map, finding interesting sequences becomes independent of the model used to create the cost map. This separation of the model from the search technique is one of the main advantages of the techniques described in this paper.

Our problem then is to find non-overlapping subsequences of a cost map whose cost is significantly lower than what we would expect from random strings of characters. Furthermore, we would like to do this in time proportional to the number of positions in the cost map.

1.2 When is a subsequence significant?

A sequence is considered interesting when the model gives it a significantly lower cost than we would expect. Formally, we look for sequences where the cost of the sequence using the model ($cost_m$) is significantly less than the cost with a null model ($cost_0$).

There are many ways to determine when the cost is significantly lower. The current program uses Milosavljević's Algorithmic Significance Method [8], which has a simple threshold test to determine if a sequence s is significant:

$$cost_0(s) - cost_m(s) > T \ ,$$

where the threshold T is computed from N, the number of sequences checked for significance:

$$T = log_2(N/\text{significance level}) \ .$$

If we want a probability of 0.01 or less of getting a sequence returned due to chance matching, then the threshold is computed as $T = \log_2(N/.01) = 6.64 + \log_2 N$. In the searching technique described in Section 1.3, one sequence is checked for each position in the database, so N is the length of the database d. For EcoSeq6, with 1,875,932 bases, finding a sequence requires that its cost under the model be 27.483 bits less than the null model cost.

This simple threshold technique is easy to implement, but the database is not really drawn randomly from the distribution implied by the null model. So even if we can refute the null model with high confidence, we have not necessarily found a good match to our model. For example, a hidden Markov model may assign cost slightly less than 2 bits/base to G and C in the *junk loop* that matches the uninteresting parts of the database. If the null model assigns 2 bits/base, then almost any sufficiently large GC-rich region would have significant savings.

On the other hand, a short REP cluster with only 24 bases would need to be encodable in only $48 - 27.479 = 20.521$ bits (0.855 bits/base) in order to be found. This stringent condition is difficult to meet with the simple models described in Section 2. Luckily, the regions around the REP clusters are fairly similar, and so the model can usually find a significant, slightly larger region around a REP.

In general, using this threshold yields too many long sequences and not enough short ones. The current program has some methods for suppressing the bogus long sequences, but does not find short sequences unless they compress very well. Since the standard deviation of the cost of a sequence of length n grows with $\sqrt{n}$, the threshold should probably have the form $T = a + b\sqrt{n}$, rather than being a simple constant, but it is unclear how to set a and b to get a given significance level for a database.

Suppressing the incorrectly found long sequences is done by adding the extra condition that the model must save at least 0.1 bits/base and by artificially replacing the 2 bits/base of the null model by a smaller estimate of the true entropy of the database (say, 1.99 bits/base). A better technique would be to use a better null model than the very crude 2 bit/base model—perhaps a second- or third-order Markov model. Note that although using a better null model or estimating the entropy at less than 2 bits/base helps to eliminate the incorrectly found long sequences, it becomes more difficult to find the correct short ones.

1.3 Scanning algorithm for finding the best subsequences

Given the definition of significantly found sequences in Section 1.2, it is easy to scan a cost map to find significant subsequences. We simply start at the left end of the map and add up the savings in each position ($\sum_i cost_0(i) - cost_m(i)$), keeping track of the greatest cumulative savings encountered. When the cumulative savings becomes negative, we have just scanned an uninteresting sequence, and so we reset the savings and greatest savings to zero, and continue the scan with the left endpoint in the current position.

A significant sequence is present when the greatest savings seen is greater than T, but the scan is continued until either we reach the end of the cost map or the savings per position drops to less than half the savings per position at the point of greatest savings. The significant sequence runs from the start of the scan to the location of the greatest savings.

This method finds sequences that are significant, but the endpoints may not be optimally chosen. If we just picked the maximal segment (as in [4]), we would have extraneous characters on the ends of the segments. If the model for the interesting sequences provides estimates for uninteresting positions that are as good as the null models, then the savings for a junk character is zero or slightly positive (violating the requirements of a scoring system used with a maximal-segment method). In fact, early versions of the program reported the maximal segments and picked up long strings of junk.

Maximizing the savings for a sequence is too greedy, reporting uninteresting sequences. One way to correct the problem would be to trim the significant sequences to maximize the savings per position. This approach, however, is too conservative, throwing away parts of the sequence that remain interesting.

Instead, the program tries to maximize the signal-to-noise ratio (SNR). The noise (the standard deviation of the savings for an uninteresting sequence) grows with $\sqrt{n}$ for junk sequences of length n, so maximizing

$$\text{SNR} = \frac{(cost_0(s) - cost_m(s))}{\sqrt{n}}$$

should give us the best signal-to-noise ratio.

We can move the endpoints inward one position at a time, keeping track of the position that gives the maximum SNR. To make sure we do not lose any significant sequences, we stop the trimming before the remaining savings would be less than $\max(T, \text{greatest savings} - T/2)$.

After finding a sequence, we can restart the scan at the right endpoint of the reported sequence. Restarting here will scan some bases repeatedly, which may seem like a violation of the assumption used for determining the significance threshold, but the only way to find a sequence containing one of these rescanned positions is for the position to be part of a sequence that saves at least twice the threshold. This savings is great enough that it would be significant even if all $\binom{d}{2}$ subsequences had been examined.

2 Markov Models as compression models of interesting sequences

2.1 Simple Markov models

A simple Markov model of order k estimates the probabilities for letters in a given position based only on the characters in the preceding k positions. The model is trained by giving it a seed (a set of sequences that are interesting) and counting the number of times each word of length $k+1$ occurs in the seed. The words that start with the same k characters constitute a context, and their counts can be converted to estimates of the probability of the characters in the final position of the word.

The strengths of a simple Markov model are that it allows very fast searching (the cost of each position can be computed in constant time), it has fairly small memory demands (4^{k+1} words for an alphabet of four letters), and the model can be built quickly from a fairly small seed.

The weaknesses are that the models are not human-readable and are not directly useful for aligning sequences. Furthermore, any extraneous junk that is in the set of seed sequences is also searched for—not just the interesting part of the seed.

Some previous work with Markov models has concentrated on using them to predict the frequencies of short words [10, 1, 14]. These studies have generally found that fairly low-order models ($k = 2$ to $k = 4$) trained on the entire database work best for that application. In contrast, for searching we use fairly high-order Markov models ($k = 6$ to $k = 10$) and small seeds (as low as

30 characters). Even with a large seed of 12,000 characters (essentially all the REPs), the average value of a count for the 4^9 words of an order-8 model is only 0.046. With such small seeds and large models, almost all contexts have zero counts for all four characters, making the way zero counts are handled particularly important.

Zero-offset

The simplest way to handle zero counts is to add a *zero-offset* z to all counts, so that the estimated probability of character b in a given context C is $(z + \mathrm{count}_C(b))/(4z + \sum_x \mathrm{count}_C(x))$. If all the counts in a context are zero (the usual case in uninteresting sequences), this method assigns a probability of 0.25 (a cost of 2 bits) to all four possible characters, independent of the choice of z. The second most common situation is to have a count of one for one character and zero for the other characters in the context. This method assigns a probability of $(1 + z)/(1 + 4z)$ to the character that was seen, and $z/(1 + 4z)$ to the other characters.

If the seed is large enough to contain multiple instances of interesting sequences, we can also choose z to minimize the encoding cost of doing adaptive compression on the seed using the model. The minimization is currently done by using Newton's method to set the derivative of the encoding cost to zero, but almost any standard optimization technique should work.

Neighbor blurring

When doing searches, we are usually interested not only in sequences identical to the seed, but in sequences that have only a few characters different from the seed. Unfortunately, even a single mutation can change a k-long context into one that has zero counts of all four words beginning with that context. To compensate somewhat for this limitation of high-order models, we can *blur* the word counts by adding a weighted sum of all neighboring word counts:

$$\mathrm{newcount}(w) = \mathrm{count}(w) + n \sum_{x \in N(w)} \mathrm{count}(x) \ ,$$

where $N(w)$ is the set of all words that differ from w in exactly one position, and n is the blurring weight for neighbor blurring.

The program uses a slightly more sophisticated model, with three blurring parameters n_1, n_2, and n_3. The n_1 weight is used for words whose difference is (A$\leftrightarrow$G) or (C$\leftrightarrow$T). The n_2 weight is used for words whose difference is (A$\leftrightarrow$C) or (G$\leftrightarrow$T), and n_3 is used for (A$\leftrightarrow$T) or (G$\leftrightarrow$C).

The neighbor weights are best chosen by optimizing the adaptive compression of a related set of sequences, using a simple optimization technique, such as gradient descent, on the four parameters z, n_1, n_2, and n_3. The optimal value of z is much smaller when neighbor blurring is used, since the neighbor blurring eliminates many of the zero counts, and provides a more accurate estimate of the probabilities than the simple zero-offset.

One problem with neighbor blurring is that we do not treat the predicted position of the word differently from the part that establishes the context. When the counts get large in a given context, blurring them may produce a worse estimate of the probabilities than the raw counts. Since we use the blurring only for models with extremely small counts, where blurring in the predicted position generally produces better predictions, this problem has been ignored.

Complement blurring

In many cases, we want to look for sequences on either strand of the DNA. We can achieve this by *complement blurring* of the word counts:

$$\mathrm{newcount}(w) = \mathrm{count}(w) + c\ \mathrm{count}(w') \ ,$$

where w' is the dyadic complement of w and c is the complement blurring weight.

order	n_1	n_2	n_3	z	c	bits	bits/base
2	-0.00001	0.00001	0.00002	3.02665	1	22335.607	1.8494
3	0.00117	0.00002	0.00000	1.51871	1	20017.514	1.6575
4	0.00635	0.00014	0.00156	0.80799	1	18410.264	1.5244
5	0.02657	0.00716	0.00341	0.44887	1	16486.609	1.3651
6	0.03536	0.00894	0.00725	0.19349	1	15384.983	1.2739
7	0.03964	0.01233	0.01248	0.06928	1	14445.732	1.1961
8	0.03546	0.01435	0.01448	0.02557	1	13604.666	1.1265
9	0.03742	0.01477	0.01463	0.01345	1	13629.369	1.1285
10	0.03753	0.01664	0.01566	0.00777	1	13899.144	1.1509

Table 1: Optimal blurring parameters for adaptive compression of the 109 sequences of REP99-gxn (12077 bases) with the complement parameter fixed at $c = 1$.

Generally, c is set to either 0 or 1, depending on the application. Choosing c by optimizing adaptive compression does not seem to work well, as gradient descent methods do not converge when optimizing with c, z, and the neighbor blurring parameters simultaneously.

Table 1 gives a table of the parameter setting for the best adaptive compression of the 109 sequences (12077 bases) of REP99-gxn (with $c = 1$). The number of counts for an order-k model is $12077 - 109k$, since the first k bases of a sequence do not generate counts.

Note that the order-2 and order-3 models use the zero-offset in preference to neighbor blurring, but as the order gets larger (and the counts per context smaller), the neighbor blurring becomes much more important. If we look at the ratio of z to the expected count $(12077 - 109k)/4^{k+1})$, we see it increasing from .0163 for order-2 models to 2.97 for order-10, even though the value of z itself is decreasing rapidly.

If we assume that all contexts contain a single count of 1 and three counts of zero, we can approximate the single-point mutation frequencies for each of the three types of substitution as

$$m_i = \frac{n_i + z}{1 + n_1 + n_2 + n_3 + 4z} \; .$$

We can improve this estimate somewhat by scaling down the z value in the formula by the expected count for non-zero contexts. We can estimate the number of non-zero contexts as roughly $4^k(1-(1-4^{-k})^x)$, where x is the total number of counts made. The expected count for a non-zero context is thus

$$\frac{x}{4^k(1-(1-4^{-k})^x)} \; .$$

The estimated mutation frequencies using this method are given in Table 2—since the assumptions of the method are more reasonable for higher-order models, the mutation rates estimates are probably most accurate for the highest order model. The very high predicted rate of substitution for $k = 5$ and $k = 6$ probably results from merging together contexts from the two parts of the REP, which are similar but not identical. Higher-order models can identify the separate parts of the REP more reliably, and so the predicted substitution rates are more likely to be reasonable.

Unfortunately, I have no way to check these predicted substitution rates against other methods for estimating substitutions rates (such as character counts in a multiple alignment), as I do not have a multiple alignment for all the REP sequences.

2.2 Hidden Markov Models

A hidden Markov model[1] (HMM) consists of a set of states connected by directed edges. Each state assigns probabilities to the characters of the alphabet used in the sequence and to the edges

[1]Rabiner has written a nice tutorial on HMMs for those who want a more detailed treatment than this paper can provide [11].

order	m_1	m_2	m_3	m
2	0.0040	0.0040	0.0040	0.0121
3	0.0091	0.0080	0.0080	0.0251
4	0.0224	0.0166	0.0179	0.0569
5	0.0555	0.0393	0.0362	0.1310
6	0.0766	0.0564	0.0552	0.1882
7	0.0709	0.0493	0.0494	0.1697
8	0.0509	0.0327	0.0328	0.1164
9	0.0452	0.0250	0.0248	0.0950
10	0.0411	0.0221	0.0212	0.0845

Table 2: Estimated substitution frequencies for the three types of substitutions (and combined mutation frequency) in REP99-gxn based on the optimal neighbor blurring parameters from Table 1.

leaving the state. There is usually a designated start state and a designated stop state (though some HMMs allow starting or stopping in any state).

A *path* in an HMM is a sequence of states such that there is an edge from each state in the path to the next state in the path. The length of a path is the number of states in the sequence, and the probability of a path is the product of the probabilities of the edges traversed.

Each path through the HMM gives a probability distribution for each position in a string of the same length, based on the probabilities for the characters in the corresponding states. The probability of the sequence given a particular path is the product of the probabilities of the characters.

The probability of any sequence of characters is the sum, over all paths whose length is the same as the sequence, of the probability of the path times the probability of the sequence given the path:

$$P_{\text{hmm}}(w) = \sum_{path\ x} P(x)P(w|x)\ .$$

If the HMM has designated start and stop states, then the sum is limited to those paths that start and end in the correct states. The above equation has omitted the renormalization needed to get the "probabilities" to sum to 1.

For computational reasons, it is often better to look not at all paths, but only at the path that maximizes the probability of the given sequence. It is also convenient to switch from path probabilities to *encoding cost*, by taking the negative log likelihood ($cost(x) = -\log_2 P(x)$). The Viterbi cost of a sequence, given an HMM, is the cost of the minimum-cost path through through the HMM.

$$\begin{aligned} \text{cost}_{\text{hmm}}(w) &= \min_{path\ x} \text{cost}(x) + \text{cost}(w|x) \\ &= \min_{path\ x} \left(\sum_{\text{edges } e\ \in\ x} \text{cost}(e) \right. \\ &\qquad \left. + \sum_{\text{states } s_i\ \in\ x} \text{cost}_{s_i}(w_i) \right)\ . \end{aligned}$$

This encoding cost for the best path (which is easily found with the Viterbi algorithm) can be assigned to individual positions in a cost map by assigning the cost of the edge into a state of the path and the cost of the character in that state to the corresponding position in the cost map.

2.3 Strengths and weaknesses of hidden Markov models

Hidden Markov models offer many advantages over simple Markov models for modeling biological sequences:

- A well-tuned HMM generally provides better compression than a simple Markov model, allowing more sequences to be significantly found.
- The models are fairly readable (at least when drawn rather than just listed). A high-quality model for REPs (compressing previously unseen REPs to about 1.25 bits/base) may have around 200 states and 300 edges, rather than the 4^9 counts of the order-8 simple Markov model. The low ratio of edges to states means that large parts of the model are simple straight-line sequences, which are easy to draw and to understand.
- The HMMs can be used for generating alignments, with each state of the machine corresponding to one column in the alignment. The best path found by the Viterbi algorithm identifies a state for each position, and that in turn can specify the column. HMMs are a bit more powerful than alignments, since the same state can be used repeatedly in a path, but each column can only be used once in an alignment. This results in ambiguous alignments if a column alignment model is used, but can be quite convenient for describing phenomena like random numbers of repeats of a short subsequence.

  HMMs also allow variant structures to be modeled directly, not just as inserts and deletes to a consensus sequence. For example, the REPv variant of the REP sequence, often found next to IHF binding sites [9], is modeled very clearly by REP99-gxn.hmm400m—in fact the IHF binding site itself occurs frequently enough next to REPs to have been included in the model.
- Separate HMMs built for recognizing particular structures can be merged to create HMMs that recognize sequences of structures [5]. Unfortunately, doing this cleanly requires a slightly different version of HMMs which allows *null states*—states that don't match any characters in the input sequence. The current version of my HMM code cannot handle HMMs with null states, but the extension is planned and should be straightforward.

HMMs do have some weaknesses:

- The Viterbi algorithm is expensive, both in terms of memory and compute time. For a sequence of length n, the dynamic programming for finding the best path through a model with s states and e edges takes memory proportional to sn and time proportional to en. For the REP searches, doing a search with a hidden Markov model is about 10 times slower than using a simple Markov model—for larger HMMs (needed for longer target sequences) the penalty would grow.

  Other algorithms for hidden Markov models, such as the forward-backward algorithm, are even more expensive.
- The HMM needs to be trained on a set of seed sequences and generally requires a larger seed than the simple Markov models. The training involves repeated iterations of the Viterbi algorithm (see Section 2.7), which can be quite slow.
- For a given set of seed sequences, there are many possible HMMs, and choosing one can be difficult. Smaller models are easier to understand, but larger models can fit the data better. Figure 1 shows the compression efficiency (in bits per base) for a number of different HMMs compressing the same set of REP sequences. Note that the compression continues to improve with larger models, and so deciding which model to use is somewhat arbitrary. Section 2.7 describes how models were chosen for this paper.

2.4 Violations of assumption needed for cost maps

In Section 1.1, I mentioned the main assumption that the costmap-based search relied on: the cost of any subsequence can be computed as the sum of costs of individual positions. Although the Viterbi algorithm gives us a way to assign costs to individual positions, the true cost of a subsequence may not be the same as the sum of the costs on the best path for the whole sequence.

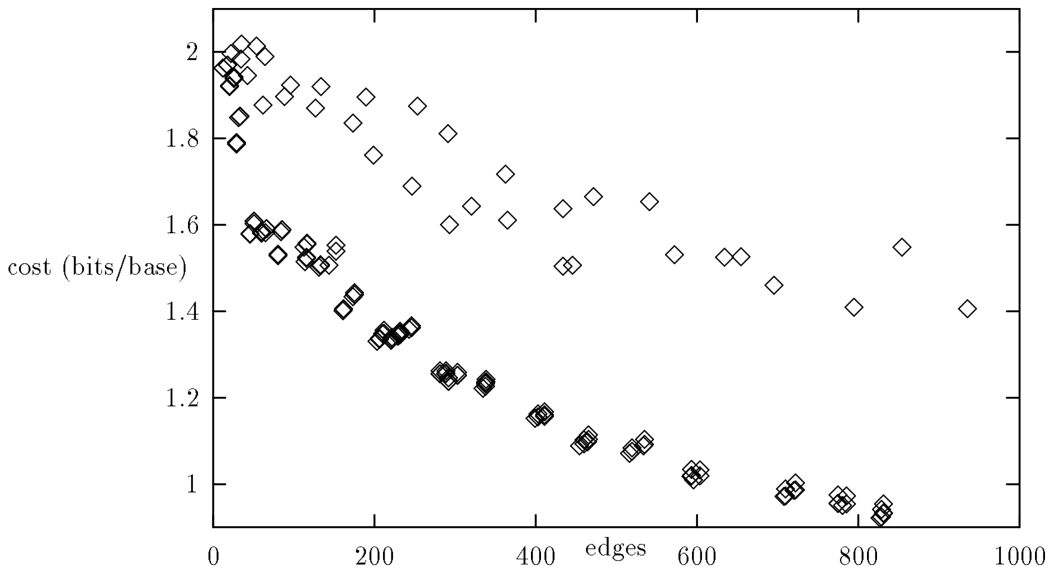

Figure 1: Information content of a set of seed sequences is plotted against the size of the hidden Markov model used for encoding them (expressed as the number of edges in the model). The points in the upper curve are from models that have not been trained—in the lower curve, from models that have been trained and had useless edges and states removed.

There is also a difference between the true compression cost of a subsequence and the cost from the map for simple Markov models, but in those models, the only difference in cost is in the first k positions, since there is no compression for the first k positions of a sequence. In the simple Markov model, we can compensate for the startup error by extending the beginnings of the subsequences found by k positions.

For hidden Markov models, a change anywhere in the sequence can change which path is the best one, and so change the cost for positions arbitrarily far away. It is not difficult to construct pathological examples in which this would completely invalidate the use of cost maps.

By limiting the types of HMMs considered, I have managed to use them quite successfully with costmap search. First, all my models contain a *junk loop*, which models the uninteresting sequences that occur between the interesting ones. The paths in the HMM for the interesting sequences begin and end at the junk loop. If there are multiple interesting sequences within a contiguous part of the database, then the best path should use the junk loop for the uninteresting parts and paths through the rest of the HMM for the interesting parts. The costs in the junk loop are very close to the null-model costs, and the costs in the rest of the model are very much lower, so the basic assumption of the cost map—that interesting positions, and only interesting positions, have much lower costs than the null model costs—is still satisfied.

2.5 Building a hidden Markov model from a simple Markov model

Although hand-crafted hidden Markov models may be useful for searches where a consensus sequence is already known, I am primarily interested in HMMs that are generated automatically from a set of seed sequences.

We need a way to control the size of the HMM, and ensure that the most useful, shared information in the seed sequences is reflected in the HMM. We already have a mechanism for pooling information from a set of seed sequences: the counts of simple Markov models. The construction process used for the results in Section 3.1 starts by building an order-k simple Markov model for a set of seed sequences, constructing an HMM from the counts of $k+1$-words in a simple Markov model of the seed, then training the model on the sequences, using the cross-training technique described in Section 2.7.

The construction technique creates states and maps words to them. First, a junk-loop state is created, and all words that aren't explicitly mapped to another state will map to the junk state. Then words are examined in decreasing order of frequency, creating a state for each one that isn't already mapped. The state will be used to match the character in the middle position of the word, so all words that differ only in that position will map to the same state. The count for each character in a state is initialized to the count for the word that has that character in the middle position.

There is a natural *shift* relationship between two words of the same length. This relationship is exploited to create the edges of the HMM, and to ensure some connection between the various states.

Definition 1: *A $(k+1)$-word w_1 is a* left-shift *of $(k+1)$-word w_2 if the last k characters of w_1 are identical to the first k letters of w_2 (w_2 is also called a* right-shift *of w_1).*

Note that the middle character position for the right-shift of w corresponds to the position one to the right of the middle in w.

We will provide an edge from state s to state t if some right-shift of one of the words mapping to s maps to t or if some left-shift of one of the words mapping to t maps to s. The count for the edge is the minimum of the counts for the corresponding words.

To ensure some connectivity in the Markov model, we don't just map the high-frequency words to states, but map their most probable predecessors and successors as well. When a new state is created for a word w, we examine the four right-shifts of the word and choose the one with the largest count $w_{\max}$. If this is above some threshold, then we create a state for $w_{\max}$ and repeat until either the maximum count drops below the threshold or all four right-shifts of the word already map to states. We do the same with the four left-shifts of w.

For example, if two seeds aaaaaatggggggg and aaaaaacgggggg were used with an order-3 Markov model, the words that map to states would be aAa, aAt, aAc, aCg and aTg, tGg, cGg, and gGg. The HMM will have only one state for aCg and aTg (perhaps best named aYg, to indicate what letters it has low costs for), but the right-shifted words (tGg and cGg) map to different states, even though they match the same character (G). Similarly the left-shifted words aAt and aAc map to different states, though both match A.

As a slight added complexity, we also create the states corresponding to the dyadic complements of the words, and tie the complementary states together with a variable-weight tie. The tie has no effect on the model when we are determining the probability of a sequence, but when the probability estimates for a state are updated, a weighted multiple of the counts for the complementary characters in the tied state is added to the counts. This trick helps the model learn palindromic sequences and sequences that can occur on either strand.

2.6 Direct construction of HMM from a sequence

HMMs can be constructed directly from a single sequence, by adding one state for each character in the sequence and connecting them in a single path. The model can be made to match sequences of any length by adding a self-loop edge to the first and last states. Such a model is not very useful in itself, since it will only match sequences which are nearly identical, but it can serve as a starting point for the model modification techniques described in Section 2.8.

From a seed with multiple sequences one could also construct parallel paths from the start state to the stop state, one per sequence. Unfortunately, such a model would offer no advantages over far more efficient string search techniques, and would be much too large for seeds of reasonable size.

2.7 Training a hidden Markov model

Tuning and pruning

The basic mechanism for training an HMM to a set of sequences is to repeatedly run the following *tuning* algorithm. First, run the Viterbi algorithm to determine the best path through the model for each sequence, and count the number of times each edge is used and the number of times each character occurs in each state. More sophisticated training algorithms, such as the Baum-Welch method [11], could be used to get expected counts, but this simple method works fairly well.

After all the sequences in the training set have been counted, the counts can be converted to probabilities for each set of characters in a state and for each set of edges out of a state. As with the counts in the simple Markov model, some care has to be taken with zero counts. Since the counts in the HMM tend to be much larger than in the simple Markov models, the handling does not have to be as careful, and the program just adds the old probability estimate plus a small fixed zero-offset to all the counts.

After the probabilities have been recomputed and converted to costs, the whole tuning step is repeated, either for a fixed number of iterations or until the change in cost per character is less than a user-specified threshold. Generally four iterations are enough to get convergence to better than 0.01 bits/base.

After tuning an HMM, some edges or states may never have been used. A simple pruning step after tuning removes these unused states and edges from the HMM.

Useless states and edges occur fairly commonly on the automatically constructed HMMs because parallel sequences are often constructed. For example, in Section 2.5 the states for tGg and cGg matched exactly the same characters. In training, one of the two states will get a higher count (hence lower cost) and the other will become unused.

The smaller models in Table 3 on page 16 (hmm125, small, and cross) were constructed from simple Markov models by the methods of Section 2.5 and tuned on the seed sequences (REP99-gxn). Unused edges and states were removed, and the models were made slightly more general by *flattening* the probabilities—re-estimating the probabilities from the counts using a larger zero-offset than used in the tuning. (These models were flattened with a zero-offset of 0.1.)

Cross-training

Because both the computational cost of a model and the amount of compression obtainable from an HMM vary with the size, determining the best size to use for a model can be difficult. Usually, we want the best compression we can get for the interesting sequences that aren't already in the training set.

To estimate this, we use a *cross-training* procedure. In cross-training, the initial set of seed sequences is split into two parts: the training set and the cross-training set. We build and tune models based on just the training set, then check them on the cross-training set, choosing the model that does best on the cross-training set. Note that this differs from cross-validation, where a check is made after all decisions have been made. If cross-validation is desired in addition, then the set of initial seeds must be divided into three sets.

Figure 2 shows a typical plot of cross-training cost versus the size of the model. Note that increasing the size of the model, which nearly always decreases the cost for the training set, eventually starts overtraining and modeling those aspects of the training set that are not shared with the cross-training set. Since we want the smallest model that will get nearly optimal compression, we are usually interested in a model near the knee of the curve—say the one that maximizes the savings (in bits per base) relative to the null model for the cross-training set divided by the $\log_2$ of the number of edges (Figure 3).

After choosing the model using cross-training, it can be improved slightly by retuning on the entire initial set of seeds and flattening the probabilities. This preserves the structure of the model, but includes all the data in the tuning.

2.8 Modifying an HMM for better matches to a seed

The methods of Section 2.7 discuss ways to change the parameters of an HMM without changing the structure of the model (other than deleting edges or states that are not used). In this section, I'll discuss a few techniques for modifying the model structure, either to improve the match between the model and the seed sequences or to decrease the size of the model.

After any operation that changes the structure of the model, the probability distributions for the states and edges involved in the change need to be recomputed, so that they continue to sum to 1.

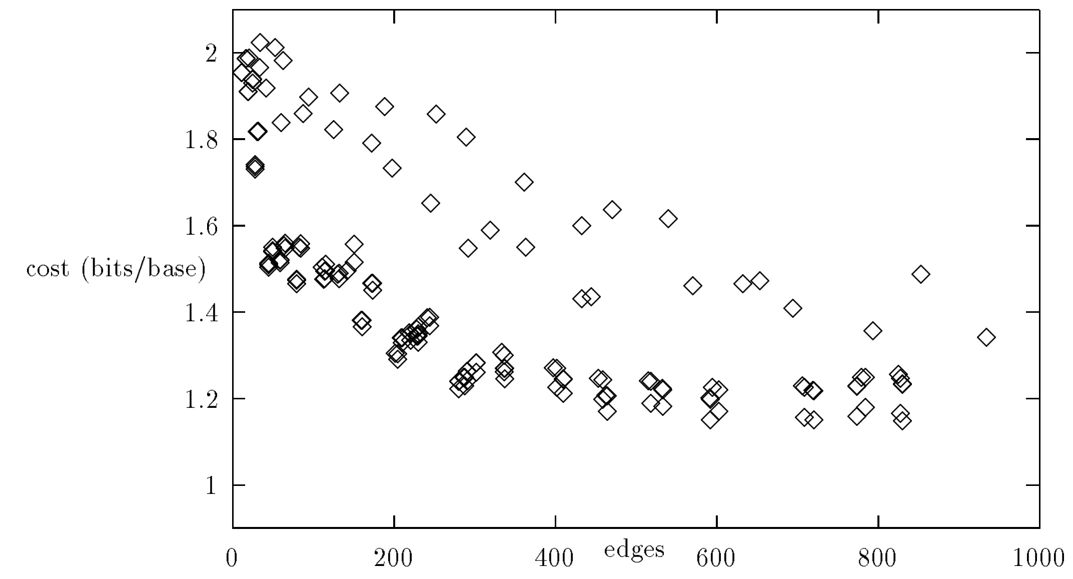

Figure 2: Scatter diagram of information content (in bits/base) of cross-training set (54 sequences) versus model size, for HMMs built from the 109 sequences of REP99-gxn (Section 3.1).

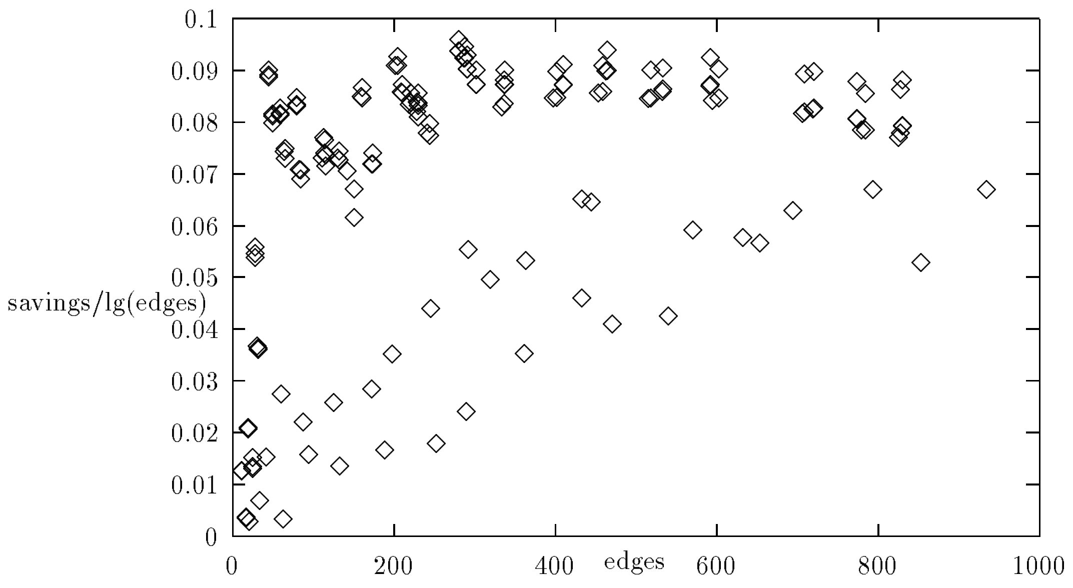

Figure 3: Scatter diagram of the savings relative to a 2-bit/base null model (in bits/base) of cross-training set (54 sequences) versus model size, for HMMs built from the 109 sequences of REP99-gxn (Section 3.1).

Two operations reduce the size of the model: *RemoveUseless* and *MergeStates*. Four operations increase the size of the model: *AddUseful*, *AddSkipEdges*, *UnrollSelfLoops*, and *SplitVagueStates*.

RemoveUseless

The simplest version of the *RemoveUseless* operation was described in Section 2.7—any edges or states whose counts are zero after training are removed. Models can be pruned more aggressively by removing any states or edges whose counts are less than some constant. Pruning off edges that have only been used once results in fairly mild pruning that usually does not change the average cost for the seed sequences much, especially if retraining is done to tune the new path through the model now used by the sequence that previously used the deleted edge. More aggressive pruning can make much larger changes in costs.

MergeStates

When constructing an HMM from a high-order simple Markov model, there are often several states created that really represent the same position in the family of sequences, but which look different in the simple Markov model because of some slight variation in a nearby position. In some cases, the tuning of the model will favor one of the states sufficiently that simple pruning with *RemoveUseless* will remove the other states corresponding to the same position, but in other cases parallel paths with slight differences will remain in the model.

To reduce the size of the model, and increase its generality, it is helpful to try to identify such parallel paths and merge them together. The *MergeStates* operation looks for two states that have a common neighbor on the same side (that is, either both have out-edges to the neighbor, or both have in-edges from the neighbor). If the two states predict similar enough distributions for the characters, then merging the states into a single state will not increase the cost of sequences by much.

The current version of the program only considers merges between states whose connection to the neighbor is a major one (the edge count a high enough percentage of the edge counts on this side of the state), and only accepts merges for which the estimated increase in total cost for the seed sequences is less than a specified threshold (generally a few bits).

All states are checked to see what they can merge with, and merges that are sufficiently good are done. After a merge, the neighbors of the merged state are rechecked, so that nearly parallel paths can be "zipped" together from a common endpoint.

AddUseful

The *AddUseful* operations adds edges and states to the HMM by considering one sequence at a time, and adding one useful edge and one useful state (with a pair of edges) to the HMM for that sequence.

Dynamic programming is used to find the cost of the lowest cost path from the start state to each state for each position in a sequence (let's call the cost $F_s(i)$ for the path to state s in position i). A similar dynamic programming algorithm finds the lowest-cost path from each state to the stop state (let's call it $B_s(i)$). Note that the lowest cost path from the start state to the stop state for the sequence w as a whole is $\min_s(F_s(i) + B_s(i))$, independent of i.

When looking for a new edge to add, consider the state a_i that minimizes $F_s(i)$ and the state b_{i+1} that minimizes $B_s(i+1) + \text{cost}_s(w_{i+1})$. If we added an edge from a_i to b_{i+1} with cost c, then there would be a path through the model with total cost

$$F_{a_i}(i) + c + \text{cost}_{b_{i+1}}(w_{i+1}) + B_{b_{i+1}}(i+1) \ .$$

If this cost is lower than the current best path, then adding the edge would lower the cost. Unfortunately, we can't freely add edges with arbitrary costs, since the cost is a negative log likelihood, and adding an edge must "steal" probability from the other out-edges of state a_i.

To minimize the error in estimating how much a new edge will save, and to make the most useful change to the model, the program looks for the position i that minimizes

$$F_{a_i}(i) + B_{b_{i+1}}(i+1) + \mathrm{cost}_{b_{i+1}}(w_{i+1}) \ ,$$

for which there isn't already an edge from a_i to b_{i+1}. An edge is added with a cost that is as large as possible while still offering some savings over the current best path. More accurately, an edge is added only if this maximum cost exceeds some user-supplied threshold, generally around 5 or 10 bits.

States are added in a similar way, by considering a_i and b_{i+2}, and choosing the position that minimizes

$$F_{a_i}(i) + B_{b_{i+2}}(i+2) + \mathrm{cost}_{b_{i+2}}(w_{i+2}) \ .$$

If there is no state with an edge from a_i and an edge to b_{i+2}, then we can add a new state s with $\mathrm{cost}_s(w_{i+1}) = 0$, an edge from a_i with cost c and an edge to b_{i+2} with cost 0. States are added only if the max cost we can use for c and still have a lower cost path is above a user-specified threshold, generally around 10 bits.

The *AddUseful* operation is particularly valuable when modifying models constructed from a single sequence, but often adds edges or states which are idiosyncratic (useful only for a single sequence). The scripts that use *AddUseful* generally follow it by retuning the model and pruning out edges and states that are used infrequently.

AddSkipEdges

The HMM implementation used for these experiments does not include null states, which have been used by other researchers for modeling deletions. To approximate the effect of null states, we can add *skip edges* around each state. That is, for each path of three states ($a \rightarrow b \rightarrow c$) in the HMM, we can add a new edge around the middle state ($a \rightarrow c$).

In some models, one state will have many in- and out-edges, and the number of skip edges around that state would be enormous. Since such nodes are usually junk loops, skip edges around them generally will not save many bits. To avoid the potential quadratic expansion of the number of edges, skip edges around a state are not added for states which would produce too many skip edges. The threshold is currently set at about one fifth the number of states.

The skip edges need to be given a fairly high cost, to avoid stealing too much probability from the other out-edges of a. For example, the `hmm700b` script described in Section 3.1 adds skip edges with a cost of 6 bits (probability approximately 0.016) at the end of the script.

UnrollSelfLoops

The automatically constructed models often contain self-loops (edges for a state back to itself). These self-loops represent a subsequence of one or more characters drawn from the distribution in the state—the length of the subsequence being modeled as an exponential distribution. In many cases, the subsequence can be better modeled by using two states. One of the two loop-unrolling transformations shown in Figure 4 is applied to all self-loops (except the start and stop states). These transformations do not change the cost for any sequences, but retraining the HMM can capture more detailed information about the first or last character of the subsequence or match the length distribution for the subsequence a little better.

SplitVagueStates

The HMMs occasionally merge paths that should be separate. One hint that this is happening is a state that matches multiple characters well. The *SplitVagueStates* operation replicates such states, duplicating the in- and out-edges, and modifies the probabilities in the states so that each only matches one character well.

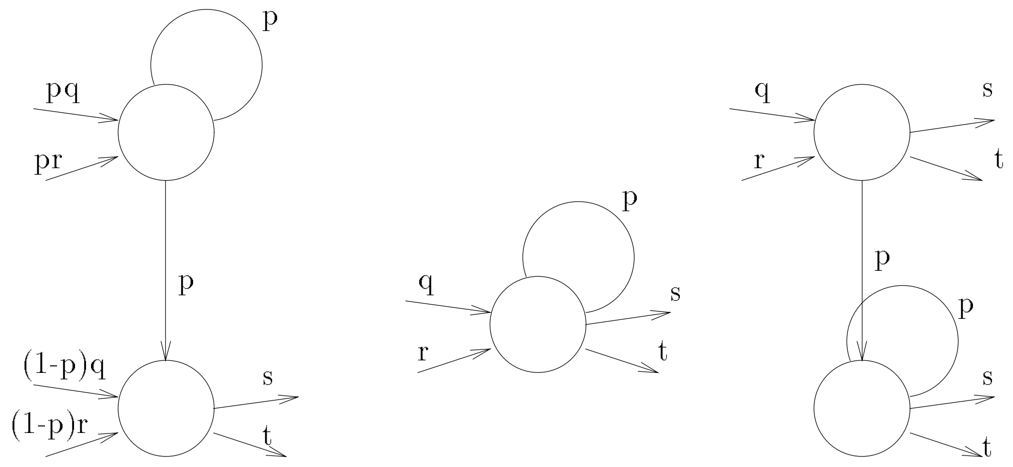

Figure 4: Two possible transformations for unrolling self-loops. Transforming the self-loop in the middle to the pair of states on the left allows the last character of the subsequence modeled by the self-loop to be better modeled, and transforming to the pair on the right allows the first character to be better modeled.

A related operation (*SplitStates*) attempts to separate two paths, splitting any state that has two high-probability out-edges (or two likely in-edges). This operation is essentially the reverse of what MergeStates does. SplitStates was not used in any of the scripts for the HMMs of Table 4.

3 Looking for REPs in EcoSeq6

As an example of the techniques described in the preceding sections, the cost map method was used to find clusters of repetitive extragenic palindromic sequences (REPs) in the 1875932 bases of the EcoSeq6 database [12]. The sequences found were compared with a list maintained by Ken Rudd [13]. The three search techniques used for building this comparison list were described and referenced in Table I of [2]. The best of the techniques mentioned there (self-BLAST) found 106 of 112 REP clusters in EcoSeq5, or about 95%.

One goal is to do at least as well as the self-BLAST search in finding the already known REPs, and, hopefully, to provide a better characterization of the structure of the REPs than current consensus sequences.

The current consensus sequence for a REP [2] is

`5'GCCKGATG-CGRCGY---RCGYCTTATCMGGCCTAC3'`

where `K` is `G` or `T`, `R` is `G` or `A`, `M` is `C` or `A`, and `Y` is `C` or `T`.

A variant on the REPs, named REPv, has also been identified, and given the following consensus sequence [13]:

`GCCTGATCGCGCTACGCTTATCAGGCCTAC.`

3.1 Looking for REPs using a seed

Using simple Markov models

One of the two largest REP clusters (REP99 or REP106) was chosen as a seed sequence and an order-8 simple Markov model was constructed from it. Since we are looking for the sequences on either strand of the DNA and a large part of what we are looking for is palindromic, the complement blurring parameter was set to 1.

The zero offset and neighbor blurring parameters were chosen arbitrarily, but near values that had given optimal adaptive compression for sets of similar sequences. The zero-offset was set to 0.1, and the neighbor blurring parameters were 0.05 (for A↔G and C↔T) and 0.01 for the other substitutions.

The entire EcoSeq6 database was searched for sequences whose cost using the models was significantly better than 1.99 bits/base. This *expand* step was repeated using the result as a seed for a new model until the number of sequences found no longer increased (5 expansions at 36 seconds each on a SparcStation 10). The final results were called REP99-gxn and REP106-gxn. These both did an excellent job of finding REPs, with 90–91% of the previously identified REP sequences found [13]. All of the sequences overlapped with a known REP sequence, and only 18–20 REP clusters were missed, all of them short clusters with a single REP in them.

A smaller seed was also tried—a single copy of the REPv consensus. Growing by repeated expansion from this seed found essentially the same set of REPs. The statistics for all three searches are listed in the first three lines of Table 3 on page 16.

To check the sensitivity to the model parameters, the repeated expansion from REP99 was done with three more sets of parameters, chosen to give optimal adaptive coding of the REP99-gxn sequences for order-6, order-8, and order-10 Markov models (lines REP99-gx6, REP99-gx8, and REP99-gx10 in Table 3 on page 16). The "extra" sequences found by the order-8 model are just regions of unknown bases in EcoSeq6. Since the program converts these unknowns arbitrarily to sequences of As, once a piece of one of them gets into the seed, the entire region of unknowns will compress very well. The order-10 model does poorly, probably because any base substitution will disrupt the compression for 11 bases, not just 7 or 9 as with the smaller models. The neighbor blurring can compensate somewhat for single errors in the 11-word, but not for two errors.

An experiment was also done using a significance threshold that varied with the square-root of sequence length, rather than being fixed, as described in Section 1.2. The standard deviation of the encoding cost per base was computed for the entire EcoSeq6 database, and the threshold set arbitrarily at five standard deviations plus three bits (the extra three bits was to prevent too many very short sequences from being falsely reported). Line REP99-gxa in Table 3 reports the results of repeatedly growing the set of sequences with this criterion. Over 30 iterations were required before the process converged, as the standard deviation was recomputed for each iteration, and the sequences found varied slightly as the threshold changed. The huge number of false negatives were almost all from the regions of unknown characters, and weeding these out produced a fairly clean set of REPs (REP99-gxawa). Using this set as a seed for the standard repeated expansion search (using a threshold of 27.483 bits) produced REP99-gxawa-gxn, which finds one more REP than REP99-gxn. Of the three "false +" sequences, one is also found by most of the hidden Markov models, and is probably a genuinely related sequence not on Rudd's list (3660932 3660963 uspAeco+846).

Using hidden Markov models

Hidden Markov models were constructed using the REP99-gxn set as a seed. Cross-training was done using a randomly chosen half of the sequences for training and the rest for cross-training. The model that minimized cross-training cost divided by the log of the number of edges was chosen. The HMM chosen, REP99-gxn-cross, has 202 states and 282 edges and compresses the REP99-gxn set to 1.24 bits/base, with almost equal compression of the two halves. A similar script that did not consider larger models came up with the same model, but retuned it on the entire set of sequences, reducing the cost to 1.20 bits/base (REP99-gxn-small). Searching EcoSeq6 with these models takes about 300 seconds on a SparcStation 10, compared to about 32 seconds for searching with a simple Markov model.

A smaller model was also built from the same seed: REP99-gxn-hmm125 built a model with 120 states and 169 edges, getting 1.367 bits/base.

I also tried some more complex scripts that attempted to merge states, remove unneeded states and edges, and do other model manipulation. Table 4 summarizes the sizes and cost/base of all the HMMs tried, and the Appendix lists the scripts used.

search name	reported sequences	reported bases	244 REP sequences false −	244 REP sequences false +	128 REP clusters false −	128 REP clusters false +
simple Markov models:						
REP99-gxn	109	12077	23	0	18	0
REP106-gxn	108	11940	25	0	20	0
REPv-gxn	107	12030	25	0	20	0
REP99-gx6	107	12010	29	0	21	0
REP99-gx8	114	23991	25	6	19	6
REP99-gx10	101	11595	34	0	27	0
REP99-gxa	606	21521	71	493	27	494
REP99-gxawa	121	8097	71	8	27	9
REP99-gxawa-gxn	114	12633	22	3	17	3
no seed:						
EcoSeq6c	371	62827	136	320	82	322
EcoSeq6c-gxn	162	105003	46	61	33	60
hidden Markov models:						
REP99-gxn-hmm125	144	9408	32	2	18	6
REP99-gxn-small	155	10099	20	7	13	10
REP99-gxn-cross	153	10037	21	6	14	9
REP99-gxn-hmm400m	123	11668	18	4	14	4
REP99-gxn-hmm400p	127	11921	13	5	9	6
REP99-gxn-hmm700b	124	12290	14	5	10	5
REP99-gxn-fromseq14	138	20893	34	8	21	10
REP99-gxn-fromseq15	125	22621	25	8	17	7
REP99-gxn-fromseq16	150	21332	26	8	16	10
HMM + expand:						
REP99-gxn-hmm125xn	120	11911	20	0	16	0
REP99-gxn-smallxn	117	12072	21	1	16	1
REP99-gxn-crossxn	116	11980	22	1	17	1
REP99-gxn-hmm400mxn	117	12205	19	2	15	2
REP99-gxn-hmm400pxn	118	12274	17	2	13	2
REP99-gxn-hmm700bxn	120	12804	14	3	10	3
REP99-gxn-fromseq14xn	120	23233	26	6	19	6
REP99-gxn-fromseq15xn	118	23694	23	6	18	6
REP99-gxn-fromseq16xn	122	23505	21	6	16	6

Table 3: Summary of using various models to search for REP clusters in EcoSeq6. The results of the search were compared with a list of 244 known sequences grouped into 128 clusters [13]. The table reports the number of sequences (or clusters) on the list that do not overlap with any sequence found (false −), and the number of sequences found that do not overlap with any on the list (false +).

The largest model (REP99-gxn-hmm700b) has over half its edges as "skip edges" to allow for skipping a base in the sequence. These extra edges increase the cost for the seed set from 0.94 bits/base to 0.96 bits/base, but even this HMM cannot find REP60, which skips three normally crucial bases in the consensus sequence. An HMM that allowed null states (as Krogh's do [5]) might be able to recognize REP60, but Krogh's simple left-to-right models cannot be used for searching for repeated occurrences of a REP in a sequence, and my code has not been rewritten yet to handle null states.

The "fromseq" scripts use direct construction of an HMM from the first sequence in the seed (see Section 2.6), then add useful states and edges for the remaining sequences. These models are somewhat smaller than the models constructed from simple Markov models, but do not do quite as well on the searches.

The results of searching with all these models (looking for sequences that had 27.483 bits better than 1.97 bits/base) were expanded once with a simple Markov model (27.483 bits better than 1.99

name	states	edges	cost (bits)	cost (bits/base)
REP99-gxn-hmm125	120	169	16506	1.36673
REP99-gxn-small	202	282	14587	1.20784
REP99-gxn-cross	202	282	14973	1.23983
REP99-gxn-hmm400m	227	293	13297	1.10098
REP99-gxn-hmm400p	272	422	12540	1.03831
REP99-gxn-hmm700b	437	1358	11553	0.95660
REP99-gxn-fromseq14	134	190	15378	1.27331
REP99-gxn-fromseq15	133	249	14461	1.19742
REP99-gxn-fromseq16	103	146	15251	1.26282

Table 4: Sizes and encoding cost of REP99-gxn for the hidden Markov models built from the sequences of REP99-gxn.

model REP99-	start	stop	where in EcoSeq6
gxn-small,gxn-cross	367257	367298	codBecoM+6609
gxn-hmm125	367259	367298	codBecoM+6611
gxn-hmm400p, gxn-hmm700b	872107	872123	ECOCHLEN-c+119
gxn-hmm700b	2313595	2313636	ECOECOA-c+13
gxn-hmm400p	2313601	2313630	ECOECOA-c+19
gxn-small	2313614	2313632	ECOECOA-c+32
gxn-hmm700bxn	3660917	3660985	uspAeco+831
gxawa-gxn, gxn-crossxn, gxn-hmm400mxn	3660932	3660963	uspAeco+846
gxn-smallxn	3660932	3660965	uspAeco+846
gxn-hmm700b	3660932	3660985	uspAeco+846
gxawa	3660934	3660963	uspAeco+848
gxn-hmm400m	3660940	3660963	uspAeco+854
gxn-small,gxn-cross	3660941	3660963	uspAeco+855
gxn-hmm400p	3660943	3660963	uspAeco+857
gxn-hmm400mxn, gxn-hmm400pxn	3909571	3909601	gyrBecoM+10390
gxn-hmm400p	3909572	3909600	gyrBecoM+10391
gxn-hmm400m	3909573	3909601	gyrBecoM+10392
gxn-small	3909574	3909601	gyrBecoM+10393

Table 5: Table of sequences found by distinctly different searches, but not on the list of known REPs [13]. The codBecoM sequence is adjacent to REPv9, and the ECOECOA-c sequence is adjacent to REP117. But the ECOCHLEN-c, uspAeco, and gyrBecoM sequences do not seem to be near any of the known REPs.

start	stop	sequence
872107	872123	cgcgtcttatcaggcct
3660917	3660985	tggcgcgccttgttacctgatcagcgtaaacaccttatctggcctacggtctgcgtacgcaatcaaaat
3909571	3909601	ttttcgtagggcggataagcaccgcgcatcc

Table 6: The new possible REP sequences or fragments reported in Table 5 are listed here, using the earliest start and latest stop position for any of the searches.

bits/base). Table 3 summarizes the results for the HMMs directly and for the expanded sets (with an “xn” at the end of the name).

Some of the “false positives” may represent previously unrecognized REP sequences, and others may be conserved regions adjacent to REPs. Table 5 lists the sequences that were found repeatedly by distinctly different searches—all of these look like they are closely related to REP or REPv sequences. The three sequences that are not adjacent to an already known sequence are shown in Table 6.

Let’s look at alignments of the three new potential REP sequences to the REP consensus. The first one is clearly the second half of a REP sequence.

```
cgcgtcttatcaggcct
 ****************
-RCGYCTTATCMGGCCTAC3'
```

Looking back a little bit extends the fragment to almost a REP, though the gap in the middle is longer than usual:

```
aaattg-ctgatg--acgtggcggagtgccgcgtcttatcaggcctggagg
     * ****** ****            ****************
   5'GCCKGATGCGRCGY-----------RCGYCTTATCMGGCCTAC3'
```

The second seems to contain a REP in the middle:

```
tggcgcgccttgttacctgat-cagcgtaaacaccttatctggcctacggtctgcgtacgcaatcaaaat
               ****** * ****  ** ***************
            5'GCCKGATGCGRCGY--RCGYCTTATCMGGCCTAC3'
```

The third seems to contain a somewhat corrupt REPv-:

```
ttttcgtagggcggataagcaccgcgc-atcc
     ***** * *******   **** ***
     GTAGGCCTGATAAGCGTAGCGCGATCAGGC
```

My HMMs seem to indicate a different consensus sequence for REPv: `YGCCKGATGCGCTACGCTTATCAGGCCTACR` without the C after the second T. The found sequence is an even better match to the complement of this sequence:

```
ttttcgtagggcggataagcaccgcgcatcccgacac
    ****** * *******   ******** * **
    YGTAGGCCTGATAAGCGTAGCGCATCM-GGCR
```

3.2 Looking for repeated elements without a seed

Since REPs are so common (REP clusters make up about 0.6% of the *E. coli* genome), it should be possible to find them without using a seed—just from looking at the database itself and trying to find repeated patterns.

An attempt was made to *concentrate* the EcoSeq6 database, by building a simple order-8 Markov model (with zero-offset and complement blurring both set to 1 and no neighbor blurring) from the entire database, then using the model to search the database for sequences that compressed significantly better than average. (See the EcoSeq6c line in Table 3.)

The resulting model found about 45% of the REP sequences, but only about 5.6% of the bases found were in a REP cluster. Growing the set of sequences by repeated expansion increased the number of REP clusters found to about 79% (EcoSeq6c-gxn in Table 3), but still only about 7.4% of the bases were in REP clusters. The problem is that there are some much larger repeated sequences, particularly the numerous IS sequences, and the repeated expansion process is looking simultaneously for REPs and IS sequences.

If each of the sequences in the "concentrated" file EcoSeq6c is individually used as a seed that is grown by repeated expansion, we get many different sets of sequences. Most of the sets of sequences are clearly identifiable (REPs, IS2, IS5, ...). If we look just at the sets in which one or more REPs are found, we find very similar coverage of the REPs (21–27 REPs missed), no matter which seed is used (see Table 7). Although REP99 and REP106 were originally chosen as seeds for Table 3 because they were the biggest known REP clusters, it does not seem necessary to start with them—almost any REP cluster found by concentrating the database works about as well.

3.3 Hard-to-find REPs

Some of the known REPs did not come up in any of the searches, and others rarely appeared. Table 8 on page 20 lists the REP sequences that were rarely found. Most of the missed sequences are fairly far from the consensus sequence, but three of the misses are rather surprising: REP39- (which is found only by REP99-gx6), REP60 (which is found only by REP99-gxn-fromseq14, REP99-gxn-fromseq15, and REP99-gxn-fromseq16), and REP83 (which is found by most of the HMMs, but not by the simple Markov models).

	reported		244 REP sequences		128 REP clusters	
search name	sequences	bases	false −	false +	false −	false +
EcoSeq6c-359-gxn	109	12374	21	0	18	0
EcoSeq6c-367-gxn	109	12016	22	0	18	0
EcoSeq6c-292-gxn	110	12055	22	0	18	0
EcoSeq6c-366-gxn	109	12171	22	0	18	0
EcoSeq6c-287-gxn	109	12213	22	0	18	0
EcoSeq6c-110-gxn	109	12216	22	0	18	0
EcoSeq6c-272-gxn	109	12232	22	0	18	0
EcoSeq6c-3-gxn	109	12255	22	0	18	0
EcoSeq6c-2-gxn	116	23671	22	6	18	6
EcoSeq6c-275-gxn	115	24390	22	6	18	6
EcoSeq6c-283-gxn	108	12202	22	0	19	0
EcoSeq6c-370-gxn	109	12010	23	0	18	0
EcoSeq6c-290-gxn	109	12188	23	0	18	0
EcoSeq6c-132-gxn	109	12235	23	0	18	0
EcoSeq6c-281-gxn	109	12240	23	0	18	0
EcoSeq6c-1-gxn	109	12270	23	0	18	0
EcoSeq6c-293-gxn	109	12427	23	0	18	0
EcoSeq6c-6-gxn	114	12868	23	5	18	5
EcoSeq6c-371-gxn	116	24726	23	6	18	6
EcoSeq6c-125-gxn	108	12072	23	0	19	0
EcoSeq6c-278-gxn	108	12204	23	0	19	0
EcoSeq6c-126-gxn	108	12289	23	0	19	0
EcoSeq6c-284-gxn	108	12559	23	0	19	0
EcoSeq6c-4-gxn	109	12049	24	0	19	0
EcoSeq6c-271-gxn	109	12110	24	0	19	0
EcoSeq6c-270-gxn	108	12132	24	0	19	0
EcoSeq6c-279-gxn	108	12201	24	0	19	0
EcoSeq6c-294-gxn	108	12236	24	0	19	0
EcoSeq6c-124-gxn	108	12651	24	0	19	0
EcoSeq6c-368-gxn	108	12135	24	0	20	0
EcoSeq6c-131-gxn	107	12624	24	0	20	0
EcoSeq6c-273-gxn	114	25173	24	6	20	6
EcoSeq6c-108-gxn	109	12018	25	0	19	0
EcoSeq6c-18-gxn	108	12058	25	0	19	0
EcoSeq6c-289-gxn	114	24525	25	6	19	6
EcoSeq6c-107-gxn	108	11629	25	0	20	0
EcoSeq6c-16-gxn	109	11649	25	0	20	0
EcoSeq6c-7-gxn	108	11673	25	0	20	0
EcoSeq6c-129-gxn	108	11731	25	0	20	0
EcoSeq6c-15-gxn	107	11967	25	0	20	0
EcoSeq6c-364-gxn	107	12007	25	0	20	0
EcoSeq6c-282-gxn	108	12234	25	0	20	0
EcoSeq6c-5-gxn	108	12271	25	0	20	0
EcoSeq6c-19-gxn	107	12319	25	0	20	0
EcoSeq6c-365-gxn	107	12572	25	0	20	0
EcoSeq6c-285-gxn	108	12128	26	0	20	0
EcoSeq6c-276-gxn	107	12300	26	0	20	0
EcoSeq6c-17-gxn	107	11707	26	0	21	0
EcoSeq6c-274-gxn	107	11771	26	0	21	0
EcoSeq6c-291-gxn	106	11790	27	0	22	0
EcoSeq6c-149-gxn	106	12112	27	0	22	0

Table 7: Search results starting with each sequence from the "concentrated" file EcoSeq6c as a seed for repeated expansion. Only those searches that found at least on REP are reported here. The results are sorted by the number of known REPs that they missed.

missed	start	stop	name
81	4160148	4160172	REP95a (> 6)
81	4071267	4071294	REP123 (6)
81	3575796	3575821	REP72 (> 6)
81	2966594	2966627	REP61 (> 6)
81	2727203	2727228	REP55 (> 6)
81	2014124	2014151	REP40a (> 6)
80	698308	698337	REP115- (5)
80	4294808	4294836	REP102b (> 6)
80	1977449	1977480	REP39- (2)
79	2538240	2538265	REP50 (> 6)
78	3803080	3803108	REP80- (6)
78	2948389	2948418	REP60 (3)
77	4618233	4618264	REP110 (> 6)
77	2457214	2457245	REP119 (5)
77	1177270	1177299	REP30 (5)
73	990645	990675	REP26b (> 6)
73	4160213	4160242	REP95b- (4)
73	3936184	3936214	REP83 (3)
72	4143645	4143674	REP93 (5)
72	2362130	2362161	REP118 (4)
64	368000	368027	REP10- (5)
58	4084547	4084578	REP90-

Table 8: REP sequences that were not found by most searches. The first column lists the number of searches (out of 81) that failed to find the REP. The name is from the list compiled by Rudd [13].

4 Conclusions and future work

I have shown how cost maps can be used effectively to search for interesting DNA sequences using two different types of models: simple Markov models and hidden Markov models. The HMMs provide a more sensitive search technique, but both types of model are quite effective at finding REPs in the *E. coli* genome—as effective as the best previously known techniques [2, Table I].

Several improvements are planned including better techniques for building HMMs (perhaps using simple Markov models to define states, but actual sequences to get the connectivity between states), better handling of ambiguity characters in the database and in the seed sequences, interfacing the HMMs to a multiple-alignment program, using a non-constant null model, including null states in the HMMs, allowing the user to specify that only sequences that use particular states of the HMM are interesting, chaning the definition of blurring to blur only in the "context" positions and not the predicted position, and modifying the code to handle protein sequences.

There are interesting repeated sequences that are not found by concentrating EcoSeq6. For example, the sequences found by Kunisawa and Nakamura [6] are not in the concentrated file nor in the sets of sequences found by the REP models. Growing their set of five examples finds a total of eight examples in EcoSeq6. Perhaps the threshold for significance could be changed to find repeated elements that are either not quite as common or not as long as the REP and IS sequences.

I'm also interested in studying the HMMs that are produced and using them to characterize and classify the REP sequences. One previous study identified some interesting REP clusters as containing binding sites for integration host factor (IHF), calling them repetitive IHF-binding palindromic elements (RIPs) [9]. As a preliminary step, I examined the HMMs to see if they modeled the IHF binding site of the RIPs. The site has also been referred to as sequence L of a BIME [3]).

In REP99-gxn.hmm400m, there is a sequence of states matching CAATATATTG (Figure 5, upper-left side), which matches 48 times, while Oppenheim *et al.* reported only 33 possible binding sites in EcoSeq5, 28 as part of "RIP" elements and 5 as parts of "near-RIPs" [9]. The HMM missed one of the RIP sites (in REP102—only part of REP102 was found by the HMM) and one of the

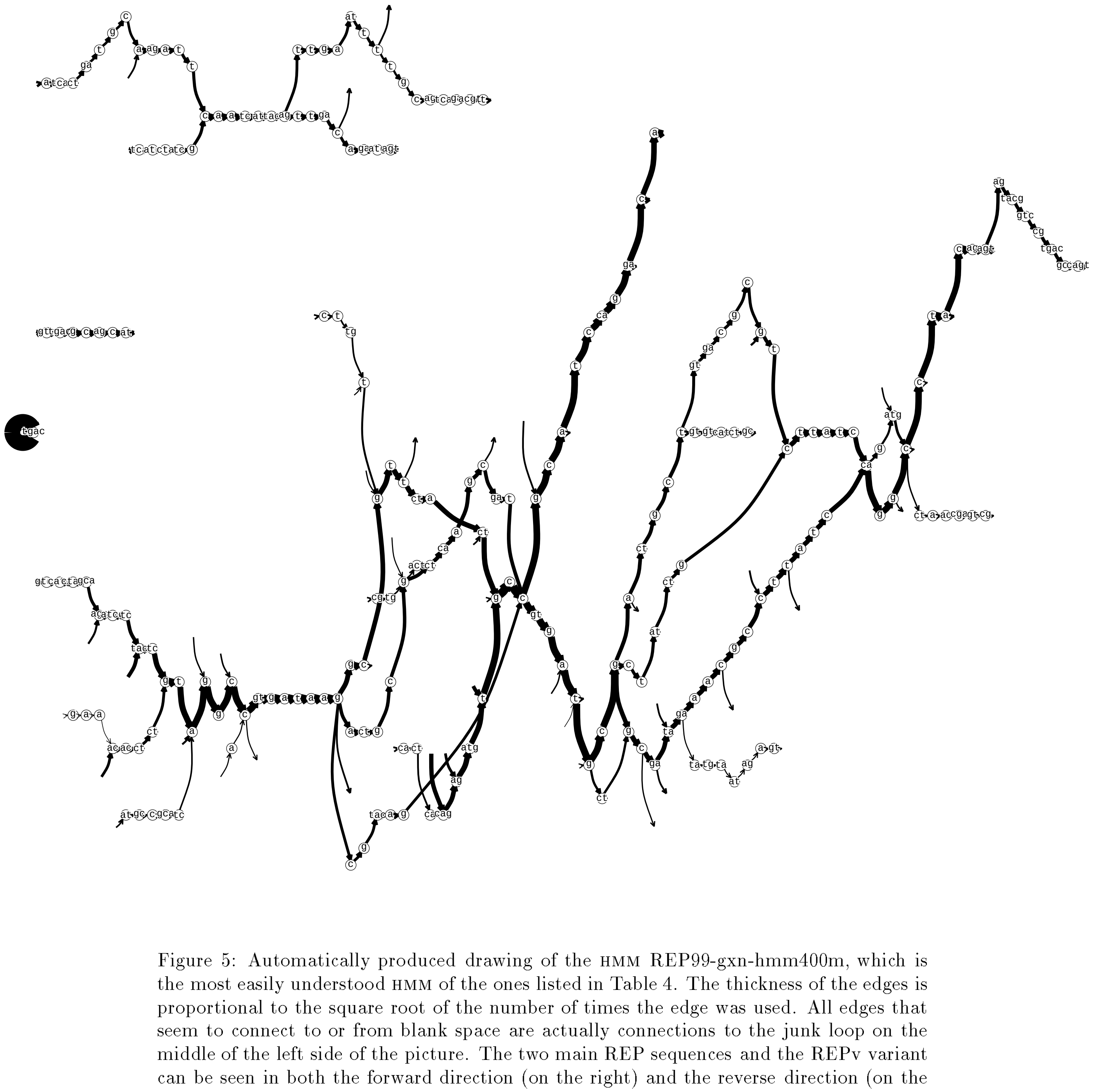

Figure 5: Automatically produced drawing of the HMM REP99-gxn-hmm400m, which is the most easily understood HMM of the ones listed in Table 4. The thickness of the edges is proportional to the square root of the number of times the edge was used. All edges that seem to connect to or from blank space are actually connections to the junk loop on the middle of the left side of the picture. The two main REP sequences and the REPv variant can be seen in both the forward direction (on the right) and the reverse direction (on the left).

near-RIPs (REP95 was not found at all). The seventeen locations for possible IHF binding sites newly found by the HMM are in REPs 18 (twice), 36, 44, 45, 46, 64 (twice), 89, 107, 112, 113 (twice), 121, 126, 127. Also, there are two locations in REP34, only one of which is listed as a RIP by Oppenheimer *et al.* [9, Fig. 4].

Since the IHF binding sequence is palindromic for the middle 10 bases, the 9-words of the simple Markov model can't determine the direction in the middle, and so the paths for the two directions share the middle two states when built directly from the simple Markov models. State merging results in blurring the two directions still more.

References

[1] J. Arnold, A. J. Cuticchia, D. A. Newsome, W. W. Jenning III, and R. Ivarie. Mono- through hexanucleotide composition of the sense strand of yeast DNA: a Markov chain analysis. *Nucleic Acids Research*, 16(14):7145–7158, 1988.

[2] G. P. Dimri, K. E. Rudd, M. K. Morgan, H. Bayat, and G. F.-L. Ames. Physical mapping of repetitive extragenic palindromic sequences in *Escherichia coli* and phylogenetic distribution among *Escherichia coli* strains and other enteric bacteria. *Journal of Bacteriology*, 174(14):4583–4593, July 1992.

[3] E. Gilson, W. Saurin, D. Perrin, S. Bacheullier, and M. Hofnung. Palindromic units are part of a new bacterial interspersed mosaic element (BIME). *Nucleic Acids Research*, 19(7):1375–1383, 1991.

[4] S. Karlin and S. F. Altshul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proceedings of the National Academy of Sciences, USA*, 87:2264–2268, Mar. 1990.

[5] A. Krogh, I. S. Mian, and D. Haussler. A hidden Markov model that finds genes in *E. coli* DNA. Technical Report UCSC-CRL-93-33, University of California at Santa Cruz, Computer Science, UC Santa Cruz, CA 95064, 1993. In preparation.

[6] T. Kunisawa and M. Nakamura. Identification of regulatory building blocks in *Escherichia coli* genome. *Protein Sequences and Data Analysis*, 4:43–47, 1991.

[7] M.-Y. Leung, B. E. Blaisdell, C. Burge, and S. Karlin. An efficient algorithm for identifying matches with errors in multiple long molecular sequences. *Journal of Molecular Biology*, 221:1367–1378, 1991.

[8] A. Milosavljević. Discovering sequence similarity by the algorithmic significance method. In *Proceedings, 1st International Conference on Intelligent Systems for Molecular Biology*, pages 284–291, Menlo Park, 1993.

[9] A. B. Oppenheim, K. E. Rudd, I. Mendelson, and D. Teff. Integration host factor binds to unique class of complex repetitive extragenic DNA sequences in *Escherichia coli*. *Molecular Microbiology*, 10(1):113–122, 1993.

[10] G. J. Phillips, J. Arnold, and R. Ivarie. Mono- through hexanucleotide composition of the *Escherichia coli* genome: a Markov chain analysis. *Nucleic Acids Research*, 15(6):2611–2626, 1987.

[11] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb. 1989.

[12] K. E. Rudd. Maps, genes, sequences, and computers: An *Escherichia coli* case study. *ASM News*, 59:335–341, 1993.

[13] K. E. Rudd, 1994. Personal communication.

[14] E. E. Stückle, C. Emmrich, U. Grob, and P. J. Nielsen. Statistical anlysis of nucleotide sequences. *Nucleic Acids Research*, 18(22):6641–6647, 1990.

A Script for hmm125

```
// build a model with at most 125 states, tune it and flatten it

BuildHMM 125 1 1 999
Tune 5 .001
RemoveUseless 20 .99
Flatten 1. .1
ReorderStates
Print
```

B Script for hmm400m

```
// Build a 400-state hmm, tune, merge and add useful
// Models tried with start=stop=loop

SplitSeq .5 0 50% for cross training

BuildHMM 400 1 1 999
Tune 5 .001
RemoveUseless 20 .99
Flatten 1 .1
Tune 1 .001
MergeStates 1 1.2

Tune 5 .001
RemoveUseless 20 .99
Flatten 1 .1
Tune 1 .001
MergeStates 2 1.4

Tune 5 .001
RemoveUseless 20 .99
Flatten 1 .1
Tune 1 .001
MergeStates 4 1.6

Tune 5 .001
RemoveUseless 20 1.99
Flatten 1 .1
AddUseful 10 15

Tune 5 .001
RemoveUseless 20 .99
Flatten 1 .1
Tune 1 .001
MergeStates 2 1.4

Tune 5 .001
RemoveUseless 20 .99
Flatten 1 .1
Tune 1 .001
MergeStates 4 1.6

Tune 5 .001
RemoveUseless 20 1.99
Flatten 1 .1
```

```
RestoreBest
SplitSeq 0 0 // tune on whole set
Tune 5 .001
RemoveUseless 20 .99
Flatten 1 .1

ReorderStates
Print
```

C Script for hmm400p

```
// Build a 400-state hmm, tune, merge and add useful
// Models tried with start=stop=loop
SplitSeq 0.3 0 30% for cross training
BuildHMM 400 1 1 999
TieMult .4
Tune 5 0.001
RemoveUseless 20 0.99
Flatten 1 0.1
MergeStates 1 1.2
Tune 5 0.001
RemoveUseless 20 0.99
AddUseful 5 10
Tune 5 0.001
RemoveUseless 20 0.99
Flatten 1 0.1
MergeStates 1 1.2
Tune 5 0.001
RemoveUseless 20 0.99
AddUseful 5 10
Tune 5 0.001
RemoveUseless 20 0.99
Flatten 1 0.1
MergeStates 1 1.2
Tune 5 0.001
RemoveUseless 20 0.99
Flatten 1 0.1
MergeStates 2 1.4
Tune 5 .001
RemoveUseless 20 0.99
Flatten 1 0.1

RestoreBest
SplitSeq 0 0 // tune on whole set
Tune 5 0.001
RemoveUseless 20 0.99
Flatten 1 0.1
ReorderStates
Print
```

D Script for hmm700b

```
//build model with single loop state
BuildHMM 700 1 1 999
TieMult 0.0 // turn off complement ties

Tune 3 .001
RemoveUseless 20 .9 // remove useless
```

```
MergeStates 4 1.6 // merge aggressively
Tune 3 .001
RemoveUseless 20 2.99 // prune strongly

// add edges with potential gain>=4, states with gain>=5
AddUsefulIncr 4 5
Tune 5 .001
RemoveUseless 20 1.99 // prune mildly

// add edges with potential gain>=4, states with gain>=5
AddUsefulIncr 4 5
Tune 5 .001
RemoveUseless 20 .99 // remove useless

Flatten 1. .4 // flatten fairly strongly
AddSkipEdges 6.

ReorderStates
Print
```

E Script for fromseq14

```
BuildFromSeq 0 // build a model from the first sequence

AddUsefulIncr 5. 10.
Tune 4 .001
RemoveUseless 20 4.9
Tune 4 .001

SplitVagueStates 1.5
Tune 4 .001
RemoveUseless 20 1.9

AddUsefulIncr 5. 10.
Tune 4 .001
RemoveUseless 20 2.9
Tune 4 .001

SplitVagueStates 1.5
Tune 4 .001
RemoveUseless 20 1.9

AddUsefulIncr 5. 10.
Tune 4 .001
RemoveUseless 20 0.9

RestoreBest
Flatten 1 0.1
ReorderStates
Print
```

F Script for fromseq15

```
BuildFromSeq 0 // build a model from the first sequence

AddUsefulIncr 5. 10.
Tune 4 .001
```

```
RemoveUseless 20 4.9
Tune 1 .001

MergeStates 4 1.6
Tune 3 .001
RemoveUseless 20 1.9

AddUsefulIncr 5. 10.
Tune 4 .001
RemoveUseless 20 2.9
Tune 4 .001

SplitVagueStates 1.5
Tune 4 .001
RemoveUseless 20 1.9

AddUsefulIncr 5. 10.
Tune 4 .001
RemoveUseless 20 0.9

MergeStates 4 1.6
Tune 3 .001
RemoveUseless 20 0.9

RestoreBest
Flatten 1 0.1
ReorderStates
Print
```

G Script for fromseq16

```
BuildFromSeq 0 // build a model from the first sequence

SplitSeq .3 0 // 30% for cross training

AddUsefulIncr 2. 5.
Tune 4 .001
RemoveUseless 20 4.9
Tune 2 .001
Flatten 1 0.5

MergeStates 6 1.8
Tune 3 .001
RemoveUseless 20 1.9

AddUsefulIncr 2. 5.
Tune 4 .001
RemoveUseless 20 3.9
Tune 2 .001
Flatten 1 0.5

UnrollSelfLoops
Tune 4 .001
Flatten 1 0.5

SplitVagueStates 1.5
Tune 4 .001
Flatten 1 0.5
```

```
AddUsefulIncr 2. 5.
Tune 4 .001
RemoveUseless 20 1.9
Flatten 1 0.5

MergeStates 4 1.6
Tune 3 .001
RemoveUseless 20 0.9
Flatten 1 0.5

RestoreBest
SplitSeq 0 0
Tune 4 .001
RemoveUseless 20 1.9
Tune 4 .001
RemoveUseless 20 1.9
Flatten 1 0.5

ReorderStates
Print
```