Applications of genetic algorithms – motif finding – Hidden markov models- CpG- Islands Decoding Algorithms-similarity search tools –Fragment Assembly-Gene finding.

Hidden markov models

Introduction

Very efficient programs for searching a text for a combination of words are avail-able on many computers. The same methods can be used for searching for patterns in biological sequences, but often they fail. This is because biological `spelling' is much more sloppy than English spelling: proteins with the same function from two different organisms are almost certainly spelled differently, that is, the two amino acid sequences differ. It is not rare that two such homologous sequences have less than 30% identical amino acids. Similarly in DNA many interesting sig-nals vary greatly even within the same genome. Some well-known examples are ribosome binding sites and splice sites, but the list is long. Fortunately there are usually still some subtle similarities between two such sequences, and the ques-tion is how to detect these similarities.

The variation in a family of sequences can be described statistically, and this is the basis for most methods used in biological sequence analysis, see [1] for a presentation of some of these statistical approaches. For pairwise alignments, for instance, the probability that a certain residue mutates to another residue is used in a substitution matrix, such as one of the PAM matrices. For finding patterns in DNA, *e.g.* splice sites, some sort of weight matrix is very often used, which is simply a position specific score calculated from the frequencies of the four nucleotides at all the positions in some known examples. Similarly, methods for finding genes use, almost without exception, the statistics of codons or dicodons in some form or other.

A hidden Markov model (HMM) is a statistical model, which is very well suited for many tasks in molecular biology, although they have been mostly de-veloped for speech recognition since the early 1970s, see [2] for historical details. The most popular use of the HMM in molecular biology is as a `probabilistic pro-file' of a protein family, which is called a profile HMM. From a family of proteins (or DNA) a profile HMM can be made for searching a database for other mem-bers of the family. These profile HMMs resemble the profile [3] and weight matrix methods [4, 5], and probably the main contribution is that the profile HMM treats gaps in a systematic way.

The HMM can be applied to other types of problems. It is particularly well suited for problems with a simple `grammatical structure,' such as gene finding. In gene finding several signals must be recognized and combined into a prediction of exons and introns, and the prediction must conform to

various rules to make it a reasonable gene prediction. An HMM can combine recognition of the signals, and it can be made such that the predictions always follow the rules of a gene.

Since much of the literature on HMMs is a little hard to read for many biol-ogists, I will attempt in this chapter to give a non-mathematical introduction to HMMs. Whereas the little biological background needed is taken for granted, I have tried to explain HMMs at a level that almost anyone can follow. First HMMs are introduced by an example and then profile HMMs are described. Then an HMM for finding eukaryotic genes is sketched, and finally pointers to the litera-ture are given

From regular expressions to HMMs

Most readers have no doubt come across regular expressions at some point, and many probably use them quite a lot. Regular expressions are used in many pro-grams, in particular on Unix computers. In programs like awk, grep, sed, and perl, regular expressions can be used for searching text files for a pattern. With grep for instance, you can search a file for all lines containing `C. elegans' or `Caenorhabditis elegans' with the regular expression 'C[\.a-z]*elegans' '.Thiswillmatch any line containing a C followed by any number of lower-case letters or `.', then a space and then elegans. Regular expressions can also be used to characterize protein families, which is the basis for the PROSITE database [6].

Using regular expressions is a very elegant and efficient way to search for some protein families, but difficult for other. As already mentioned in the in-troduction, the difficulties arise because protein spelling is much more free than English spelling. Therefore the regular expressions sometimes need to be very broad and complex. Imagine a DNA motif like this:

```
A C A - - - A T G
T C A A C T A T C
A C A C - - A G C
A G A - - - A T C
A C C G - - A T C
```

(I use DNA only because of the smaller number of letters than for amino acids). A regular expression for this is

[AT] [CG] [AC] [ACGT]* A [TG] [GC] ,

Meaning that the first position is A or T, the second C or G, and so forth. The term `[ACGT]*' means that any of the four letters can occur any number of times.

T G C T - - A G G

The problem with the above regular expression is that it does not in any way distinguish between the highly implausible sequences

A C A C - - A T C

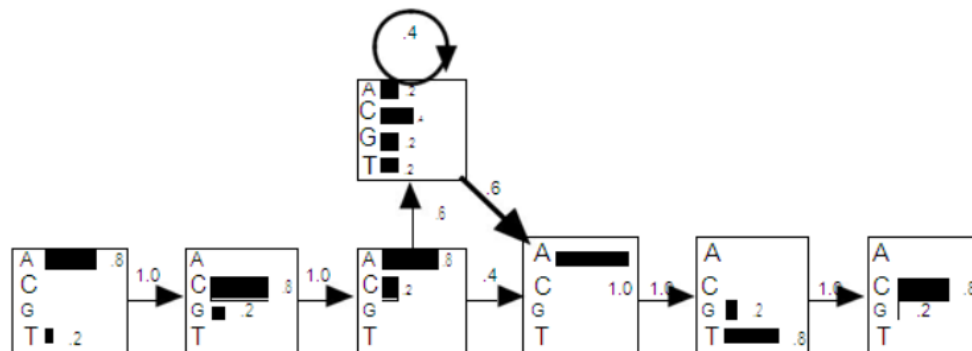


Figure : A hidden Markov model derived from the alignment discussed in the text. The transitions are shown with arrows whose thickness indicates their probability. In each state the histogram shows the probabilities of the four nucleotides.

with the most plausible character in each position (the dashes are just for aligning these sequences with the previous ones). What is meant by a `plausible' sequence can of course be debated, although most would probably agree that the first sequence is not likely to be the same motif as the 5 sequences above. It is possible to make the regular expression more discriminative by splitting it into several different ones, but it easily becomes messy. The alternative is to score sequences by how well they fit the alignment.

To score a sequence, we say that there is a probability of 4 5 =0.8 for an A in the first position and 1/5= 0.2 for a T, because we observe that out of 5 letters 4 are As and one is a T. Similarly in the second position the probability of C is 4/ 5 and of G 1/5, and so forth. After the third position in the alignment, 3 out of 5 sequences have `insertions' of varying lengths, so we say the probability of making an insertion is 3/5 and thus 2/5 for not making one. To keep track of these numbers a diagram can be drawn with probabilities as in Fig. 4.1.

This is a hidden Markov model. A box in the drawing is called a state, and there is a state for each term in the regular expression. All the probabilities are found simply by counting in the multiple

alignment how many times each event occur, just as described above. The only part that might seem tricky is the `insertion,' which is represented by the state above the other states. The probability of each letter is found by counting all occurrences of the four nucleotides in this region of the alignment. The total counts are one A, two Cs, one G, and one T, yielding probabilities 1/5, 2/5, 1/5, and 1/5 respectively. After sequences 2, 3 and 5 have made one insertion each, there are two more insertions (from sequence 2) and the total number of transitions back to the main line of states is 3 (all three sequences with insertions have to finish). Therefore there are 5 transitions in total from the insert state, and the probability of making a transition to itself is 2/5 and the probability of making one to the next state is 3/5.

	Sequence	Probability $\times$ 100	Log odds
Consensus	A C A C - - A T C	4.7	6.7
Original	A C A - - - A T G	3.3	4.9
sequences	T C A A C T A T C	0.0075	3.0
	A C A C - - A G C	1.2	5.3
	A G A - - - A T C	3.3	4.9
	A C C G - - A T C	0.59	4.6
Exceptional	T G C T - - A G G	0.0023	-0.97

Table: Probabilities and log-odds scores for the 5 sequences in the alignment and for the consensus sequence and the `exceptional' sequence.

It is now easy to score the consensus sequence ACACATC. The probability of the first A is 4 5. This is multiplied by the probability of the transition from the first state to the second, which is 1. Continuing this, the total probability of the consensus is

$$\begin{aligned} P(\text{ACACATC}) &= 0.8 \times 1 \times 0.8 \times 1 \times 0.8 \times 0.6 \times \\ &\quad 0.4 \times 0.6 \times 1 \times 1 \times 0.8 \times 1 \times 0.8 \\ &\simeq 4.7 \times 10^{-2}. \end{aligned}$$

Making the same calculation for the exceptional sequence yields only 000238* . 10^{-2} , which is roughly 2000 times smaller than for the consensus. This way we achieved the goal of getting a score for each sequence, a measure of how well a sequence fits the motif.

The same probability can be calculated for the four original sequences in the alignment in exactly the same way, and the result is shown in Table . The probability depends very strongly on the length of the sequence. Therefore the probability itself is not the most convenient number to use as a score, and the

log-odds score shown in the last column of the table is usually better. It is the logarithm of the probability of the sequence divided by the probability according to a null model. The null model is one that treats the sequences as random strings of nucleotides, so the probability of a sequence of length L is 0.25^L. Then the log-odds score is

$$\text{log-odds for sequence } S = \log \frac{P(S)}{0.25^L} = \log P(S) - L \log 0.25.$$

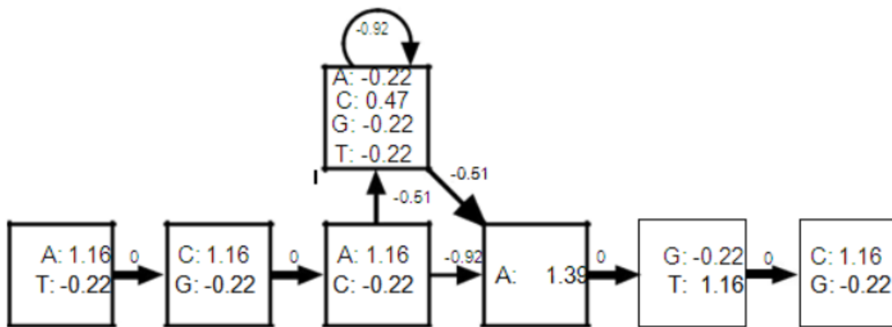


Figure : The probabilities of the model in Fig. 4.1 have been turned into log odds by taking the logarithm of each nucleotide probability and subtracting log 0. 25. The transition probabilities have been converted to simple logs

When a sequence fits the motif very well the log-odds is high. When it fits the null model better, the log-odds score is negative. Although the raw probability of the second sequence (the one with three inserts) is almost as low as that of the exceptional sequence, notice that the log-odds score is much higher than for the exceptional sequence, and the discrimination is very good. Unfortunately, one cannot always assume that anything with a positive log-odds score is `a hit,' because there are random hits if one is searching a large database.

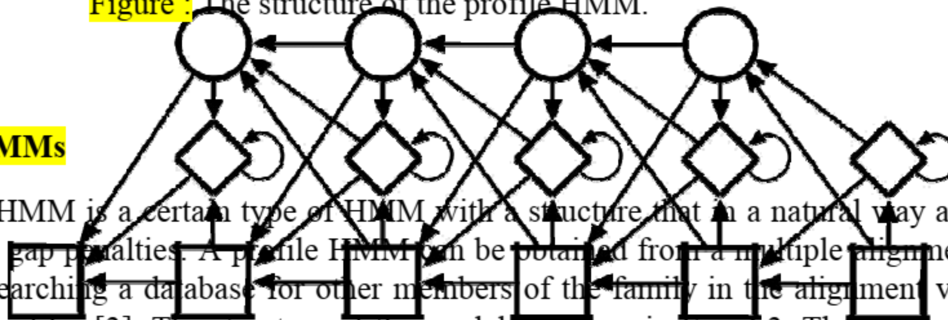
Instead of working with probabilities one might convert everything to log-odds. If each nucleotide probability is divided by the probability according to the null model (0.25 in this case) and the logarithm is applied, we would get the numbers shown in Fig.The transition probabilities are also turned into logarithms. Now the log-odds score can be calculated directly by adding up these numbers instead of multiplying the probabilities. For instance, the calculation of the log-odds of the consensus sequence is

$$\begin{aligned} \text{log-odds}(\texttt{ACACATC}) &= 1.16 + 0 + 1.16 + 0 + 1.16 - 0.51 + \\ &\quad 0.47 - 0.51 + 1.39 + 01.16 + 0 + 1.16 \\ &= 6.64. \end{aligned}$$

(The finite precision causes the little difference between this number and the one in Table 4.1.)

If the alignment had no gaps or insertions we would get rid of the insert state, and then all the probabilities associated with the arrows (the transition probabili-ties) would be 1 and might as well be ignored completely. Then the HMM works exactly as a weight matrix of log-odds scores, which is commonly used.

Figure : The structure of the profile HMM.



Profile HMMs

A profile HMM is a certain type of HMM with a structure that in a natural way allows position dependent gap penalties. A profile HMM can be obtained from a multiple alignment and can be used for searching a database for other members of the family in the alignment very much like standard profiles [3]. The structure of the model is shown in Fig. 4.3. The bottom line of states are called the main states, because they model the columns of the alignment. In these states the probability distribution is just the frequency of the amino acids or nucleotides as in the above model of the DNA motif. The second row of diamond shaped states are called insert states and are used to model highly variable regions in the alignment. They function exactly like the top state in Fig. 4.1, although one might choose to use a fixed distribution of residues, *e.g.* the overall distribution of amino acids, instead of calculating the distribution as in the example above. The top line of circular states are called delete states. These are a different type of state, called a silent or null state. They do not match any residues, and they are there merely to make it possible to jump over one or more columns in the alignment, *i.e.*, to model the situation when just a few of the sequences have a `–' in the multiple alignment at a position. Let us turn to an example.

Suppose you have a multiple alignment as the one shown in Fig.A region of this alignment has been chosen to be an `insertion,' because an alignment of this region is highly uncertain. The rest of the alignment (shaded in the figure) are the columns that will correspond to main states in the

model. For each non-insert column we make a main state and set the probabilities equal to the amino acid frequencies. To estimate the transition probabilities we count how many sequences use the various transitions, just like the transition probabilities were calculated in the first example. The model is shown in Fig. There are two transitions from a main state to a delete state shown with dashed lines in the figure, that from main state 12 to delete state

```
G G W W R G d y . g g k k q L W F P S N Y V I G W L N G y n e t
t g e r G D F P G T Y V P N W W E G q l . . n n r r G I F P S N
Y V D E W W Q A r r . . d e q i G I V P S K - - G E W W K A q s
. . t g q e G F I P F N F V G D W W L A r s . . s g q t G Y I P
S N Y V G D W W D A e l . . k g r r G K V P S N Y L - D W W E A
r s l s s g h r G Y V P S N Y V G D W W Y A r s l i t n s e G Y
I P S T Y V G E W W K A r s l a t r k e G Y I P S N Y V G D W W
L A r s l v t g r e G Y V P S N F V G E W W K A k s l s s k r e
G F I P S N Y V G E W C E A q t . k n g q . G W V P S N Y I S D
W W R V v n l t t r q e G L I P L N F V L P W W R A r d . k n g
q e G Y I P S N Y I R D W W E F r s k t v y t p G Y Y E S G Y V
E H W W K V k d . a l g n v G Y I P S N Y V I H W W R V q d . r
n g h e G Y V P S S Y L K D W W K V e v . . n d r q G F V P A A
Y V V G W M P G l n e r t r q r G D F P G T Y V P D W W E G e l
. . n g q r G V F P A S Y V E N W W N G e i . . g n r k G I F P
A T Y V E E W L E G e c . . k g k v G I F P K V F V G G W W K G
d y . g t r i q Q Y F P S N Y V D G W W R G s y . . n g q v G W
F P S N Y V Q G W W R G e i . . y g r v G W F P A N Y V G R W W
K A r r . a n g e t G I I P S N Y V G G W T Q G e l . k s g q k
G W A P T N Y L G D W W E A r s n . t g e n G Y I P S N Y V N D
W W T G r t . . n g k e G I F P A N Y V
```

Figure :An alignment of 30 short amino acid sequences chopped out of a alignment of the SH3 domain. The shaded areas are the most conserved and were chosen to be represented by the main states in the HMM. The unshaded area with lower-case letters was chosen to be represented by an insert state.

Pseudocounts

It is dangerous to estimate a probability distribution from just a few observed amino acids. If for instance you have just two sequences with leucine at a certain position, the probability for leucine would be 1 and the probability would be zero for all other amino acids at this position, although it is well known that one often sees for example valine substituted for leucine. In such a case the probability of a whole sequence may easily become zero if a single leucine is substituted by a valine, or equivalently, the log-odds is minus infinity.

Therefore it is important to have some way of avoiding this sort of over-fitting, where strong conclusions are drawn from very little evidence. The most com-mon method is to use pseudocounts, which means that one pretends to have more counts of amino acids than those from the data. The simplest is to add 1 to all the counts. With the leucine example it would mean

that the probability of leucine would be estimated as 3 23 and for the 19 other amino acids it would become 1 23. In Fig. 4.6 a model is shown, which was obtained from the alignment in Fig. 4.6 using a pseudocount of 1.

Adding one to all the counts can be interpreted as assuming a priori that all the amino acids are equally likely. However, there are significant differences in the occurrence of the 20 amino acids in known protein sequences. Therefore, the next step is to use pseudocounts proportional to the observed frequencies of the amino acids instead. This is the minimum level of pseudocounts to be used in any real application of HMMs.

Because a column in the alignment may contain information about the preferred type of amino acids, it is also possible to use more sophisticated pseudo-count strategies. If a column consists predominantly of leucine (as above), one would expect substitutions to other hydrophobic amino acids to be more probable than substitutions to hydrophilic amino acids. One can *e.g.* derive pseudocounts for a given column from substitution matrices.

Searching a database

Above we saw how to calculate the probability of a sequence in the alignment by multiplying all the probabilities (or adding the log-odds scores) in the model along the *path* followed by that particular sequence. However, this path is usually not known for other sequences which are not part of the original alignment, and the next problem is how to score such a sequence. Obviously, if we can find a path through the model where the new sequence fits well in some sense, then we can score the sequence as before. We need to `align' the sequence to the model. It resembles very much the pairwise alignment problem, where two sequences are aligned so that they are most similar, and indeed the same type of dynamic programming algorithm can be used.

For a particular sequence, an alignment to the model (or a path) is an assign-ment of states to each residue in the sequence. There are many such alignments for a given sequence. For instance an alignment might be as follows. Let us label the amino acids in a protein as A_1, A_2, A_3, *etc.* Similarly we can label the HMM states as M_1, M_2, M_3, *etc.* for match states, I_1, I_2, I_3 for insert states, and so on. Then an alignment could have A_1 match state M_1, A_2 and A_3 match I_1, A_4 match M_2, A_5 match M_6 (after passing through three delete states), and so on. For each such path we can calculate the probability of the sequence or the log-odds score, and thus we can find the *best* alignment, *i.e.*, the one with the largest probability. Although there are an enormous number of possible alignments it can be done efficiently by the above mentioned dynamic programming algorithm, which is called the Viterbi algorithm. The algorithm also gives the probability of the sequence for that alignment, and thus a score is obtained.

The log-odds score found in this manner can be used to search databases for members of the same family. A typical distribution of scores from such a search is shown in Fig. 4.7. As is also the case with other types of searches, there is no clear-cut separation of true and false positives,

and one needs to investigate some of the sequences around a log-odds of zero, and possibly include some of them in the alignment and try searching again.

An alternative way of scoring sequences is to *sum* the probabilities of all pos-sible alignments of the sequence to the model. This probability can be found by a similar algorithm called the forward algorithm. This type of scoring is not very common in biological sequence comparison, but it is more natural from a

probabilistic point of view. However, it usually gives very similar results.

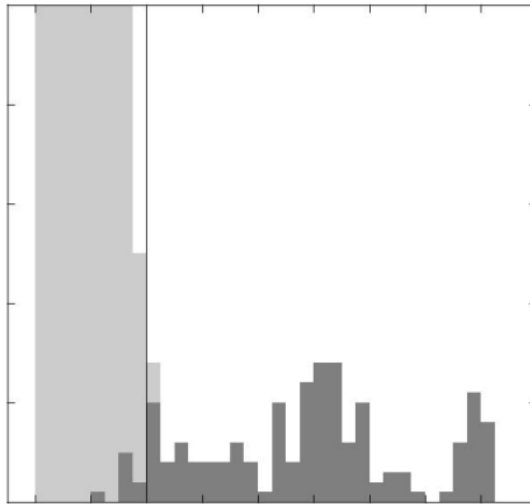


Figure : The distribution of log-odds scores from a search of Swissprot with a profile HMM of the SH3 domain. The dark area of the histogram represents the sequences with an annotated SH3 domain, and the light those that are not annotated as having one. This is for illustrative purposes only, and the sequences with log-odds around zero were not investigated further.

Model estimation

As presented so far, one may view the profile HMMs as a generalization of weight matrices to incorporate insertions and deletions in a natural way. There is however one interesting feature of HMMs, which has not been addressed yet. It is possible to estimate the model, *i.e.* determine all the probability parameters of it, from *unaligned* sequences. Furthermore, a multiple alignment of the sequences is produced in the process. Like many other multiple alignment methods this is done in an iterative manner. One starts out with a model with more or less random probabilities, or if a reasonable alignment of some of the sequences are available, a model is constructed from this alignment. Then, when all the sequences are aligned to the model, we can use the alignment to improve the probabilities in the model. These new probabilities may then lead to a slightly different alignment. If they do, we then repeat the process and improve the probabilities again.

The process is repeated until the alignment does not change. The alignment of the sequences to the final model yields a multiple alignment[1].

[1]Another slightly different method for model estimation sums over all alignments instead of using the most probable alignment of a sequence to the model. This method uses the forward algorithm instead of Viterbi, and it is called the Baum–Welch algorithm or the forward–backward algorithm.

Although this estimation process sounds easy, there are many problems toconsider to actually make it work well. One problem is choosing the appropriate model length, which determines the number of inserts in the final alignment.Another severe problem is that the iterative procedure can converge to suboptimal solutions. It is not guaranteed that it finds the optimal multiple alignment,

i.e. the most probable one.

HMMs for gene finding

One ability of HMMs, which is not really utilized in profile HMMs, is the ability to model *grammar*. Many problems in biological sequence analysis have a gram-matical structure, and eukaryotic gene structure, which I will use as an example, is one of them. If you consider exons and introns as the `words' in a language, the sentences are of the form exon-intron-exon-intron...intron-exon. The `sentences' can never end with an intron, at least if the genes are complete, and an exon can never follow an exon without an intron in between. Obviously this grammar is greatly simplified, because there are several other constraints on gene structure, such as the constraint that the exons have to fit together to give a valid coding region after splicing. In Fig. 4.8 the structure of a gene is shown with some of the known signals marked.

Formal language theory applied to biological problems is not a new invention. In particular David Searls [7] has promoted this idea and used it for gene finding [8], but many other gene finders use it implicitly. Formally the HMM can only represent the simplest of grammars, which is called a regular grammar [7, 1], but that turns out to be good enough for the gene finding problem, and many other problems. One of the problems that has a more complicated grammar than the HMM can handle is the RNA folding problem, which is one step up the ladder of grammars, because base pairing introduces correlations between bases far from each other in the RNA sequence.

I will here briefly outline my own approach to gene finding with the weight on the principles rather than on the details.

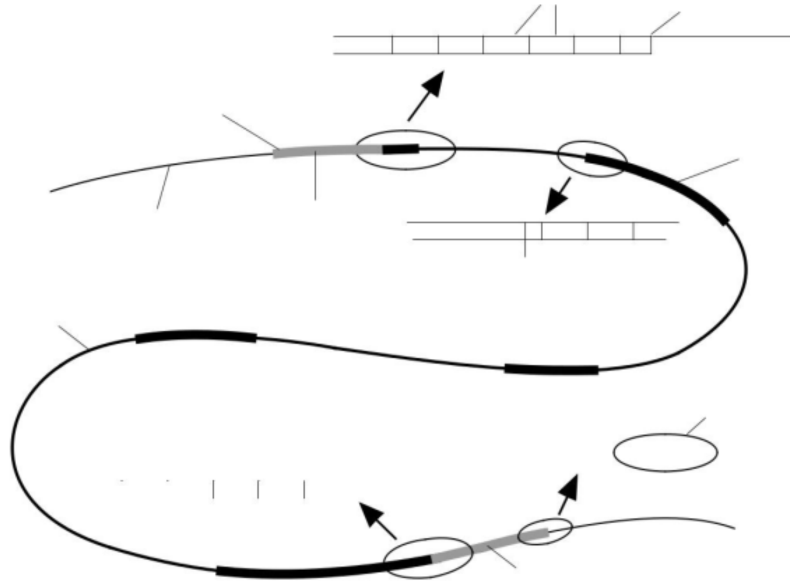


Figure : The structure of a gene with some of the important signals shown.

Signal sensors

One may apply an HMM similar to the ones already described directly to many of the signals in a gene structure. In Fig an alignment is shown of some sequences around acceptor sites from human DNA. It has 19 columns and an HMM with 19 states (no insert or delete states) can be made directly from it. Since the alignment is gap-less, the HMM is equivalent to a weight matrix.

There is one problem: in DNA there are fairly strong dinuclotide preferences. A model like the one described treats the nucleotides as independent, so dinu-cleotide preferences can not be captured. This is easily fixed by having 16 prob-ability parameters in each state instead of 4. In column two we first count all occurrences of the four nucleotides given that there is an A in the first column and normalize these four counts, so they become probabilities. This is the *condi-tional probability* that a certain nucleotide appears in position two, given that the previous one was A. The same is done for all the instances of C in column 1 and similarly for G and T. This gives a total of 16 probabilities to be used in state two of the HMM. Similarly for all the other states. To calculate the probability of a sequence, say ACTGTC , we just multiply the conditional probabilities

$$P(ACTGTC\ldots) = p_1(A) \times p_2(C|A) \times p_3(T|C) \times p_4(G|T) \times p_5(T|G) \times p_6(C|T) \times \ldots$$

```
C T C C C T G T G T C C A C A G G C T
T A T T G T T T T C T T A C A G G G C
G T T C C T T T G T T T C T A G C A C
T G C C T C T C T T T T C A A G G G T
T C C T A T A T G T T G A C A G G G T
T T C T G T T C C G A T G C A G G G C
T T G G G T T T C T T T G C A G A A C
C A C T T T G C T C C C A C A G C G T
C C C A T G T G A C C T G C A G G T A
T A T T T A T T T A A C A T A G G G C
A T G T G C A T C C C C C C A G G A G
T T T T C C T T T T C T A C A G A A T
T C G T G T G T C T C C C C A G C C C
T T C C A T G T C C T G A C A G G T G
A C G A C A T T T T C C A C A G G A G
G T G C C T C T C C C T C C A G A T T
```

Big-O Notation

Computer scientists use the *Big-O* notation to describe concisely the running time of an algorithm. If we say that the running time of an algorithm is quadratic, or $O(n^2)$, it means that the running time of the algorithm on an input of size n is limited by a quadratic function of n. That limit may be $99.7n^2$ or $0.001n^2$ or $5n^2+3.2n+99993$; the main factor that describes the growth rate of the running time is the term that grows the fastest with respect to n, for example n^2 when compared to terms like $3.2n$, or 99993. All functions with a leading term of n^2 have more or less the same rate of growth, so we lump them into one class which we call $O(n^2)$. The difference in behavior between two quadratic functions in that class, say $99.7n^2$ and $5n^2 + 3.2n + 99993$, is negligible when compared to the difference in behavior between two functions in different classes, say $5n^2 + 3.2n$ and $1.2n^3$. Of course, $99.7n^2$ and $5n^2$ are different functions and we would prefer an algorithm that takes $5n^2$ operations to an algorithm that takes $99.7n^2$. However, computer scientists typically ignore the leading constant and pay attention only to the fastest-growing term.

When we write $f(n) = O(n^2)$, we mean that the function $f(n)$ does not grow faster than a function with a leading term of cn^2, for a suitable choice of the constant c. A formal definition of Big-O notation, which is helpful in analyzing an algorithm's running time, is given in figure 2.7.

The relationship $f(n) = O(n^2)$ tells us that $f(n)$ does not grow faster than some quadratic function, but it does not tell us whether $f(n)$ grows slower than any quadratic function. In other words, $2n = O(n^2)$, but this valid statement is not as informative as it could be; $2n = O(n)$ is more precise. We say that the Big-O relationship establishes an *upper bound* on the growth of a function: if $f(n) = O(g(n))$, then the function f grows no faster than the function g. A similar concept exists for *lower bounds*, and we use the notation $f(n) = \Omega(g(n))$ to indicate that f grows no slower than g. If, for some function g, an algorithm's time grows no faster than g *and* no slower than g, then we say that g is a *tight bound* for the algorithm's running time. For example, if an algorithm requires $2n \log n$ time, then technically, it is an $O(n^2)$ algorithm even though this is a misleadingly loose bound. A tight bound on the algorithm's running time is actually $O(n \log n)$. Unfortunately, it is often easier to prove a loose bound than a tight one.

In keeping with the healthy dose of pessimism toward an algorithm's correctness, we measure an algorithm's efficiency as its *worst case* efficiency, which is the largest amount of time an algorithm can take given the worst

A function $f(x)$ is "Big-O of $g(x)$", or $O(g(x))$, when $f(x)$ is less than or equal to $g(x)$ to within some constant multiple c. If there are a few points x such that $f(x)$ is not less than $c \cdot g(x)$, this does not affect our overall understanding of f's growth. Mathematically speaking, the Big-O notation deals with *asymptotic* behavior of a function as its input grows arbitrarily large, beyond some (arbitrary) value x_0.

Definition 2.1 *A function $f(x)$ is $O(g(x))$ if there are positive real constants c and x_0 such that $f(x) \leq cg(x)$ for all values of $x \geq x_0$.*

For example, the function $3x = O(.2x^2)$, but at $x = 1, 3x > .2x^2$. However, for all $x > 15$, $.2x^2 > 3x$. Here, $x_0 = 15$ represents the point at which $3x$ is bounded above by $.2x^2$. Notice that this definition blurs the advantage gained by mere constants: $5x^2 = O(x^2)$, even though it would be wrong to say that $5x^2 \leq x^2$.
Like Big-O notation, which governs an upper bound on the growth of a function, we can define a relationship that reflects a lower bound on the growth of a function.

Definition 2.2 *A function $f(x)$ is $\Omega(g(x))$ if there are positive real constants c and x_0 such that $f(x) \geq cg(x)$ for all values of $x \geq x_0$.*

If $f(x) = \Omega(g(x))$, then f is said to grow "faster" than g.
Now, if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$ then we know very precisely how $f(x)$ grows with respect to $g(x)$. We call this the Θ relationship.

Definition 2.3 *A function $f(x)$ is $\Theta(g(x))$ if $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.*

Figure 2.7 Definitions of the Big-O, Ω, and Θ notations.

possible input of a given size. The advantage to considering the worst case efficiency of an algorithm is that we are guaranteed that our algorithm will never behave worse than our worst case estimate, so we are never surprised or disappointed. Thus, when we derive a Big-O bound, it is a bound on the worst case efficiency.

We illustrate the above notion of efficiency by analyzing the two sorting algorithms, SELECTIONSORT and RECURSIVESELECTIONSORT. The parameter that describes the input size is n, the number of integers in the input list, so we wish to determine the efficiency of the algorithms as a function of n.

The SELECTIONSORT algorithm makes $n - 1$ iterations in the **for** loop and analyzes $n - i + 1$ elements $a_i, \ldots, a_n$ in iteration i. In the first iteration, it analyzes all n elements, at the next one it analyzes $n - 1$ elements, and so on. Therefore, the approximate number of operations performed in SELECTIONSORT is: $n+(n-1)+(n-2)+\cdots+2+1=1+2+\cdots+n = n(n+1)/2$.[15] At each iteration, the same swapping of array elements occurs, so SELECTIONSORT requires roughly $n(n+1)/2 + 3n$ operations, which is $O(n^2)$ operations.[16] Again, because we can safely ignore multiplicative constants and terms that are smaller than the fastest-growing term, our calculations are somewhat imprecise but yield an overall picture of the function's growth.

We will now consider RECURSIVESELECTIONSORT. Let $T(n)$ denote the amount of time that RECURSIVESELECTIONSORT takes on an n-element array. Calling RECURSIVESELECTIONSORT on an n-element array involves finding the smallest element (roughly n operations), followed by a recursive call on a list with $n - 1$ elements, which performs $T(n - 1)$ operations. Calling RECURSIVESELECTIONSORT on a 1-element list requires 1 operation (one for the **if** statement), so the following equations hold.

$$\begin{aligned} T(n) &= n + T(n-1) \\ T(1) &= 1 \end{aligned}$$

Therefore,

$$
\begin{aligned}
T(n) &= n + T(n-1) \\
&= n + (n-1) + T(n-2) \\
&= n + (n-1) + (n-2) + \cdots + 3 + 2 + T(1) \\
&= O(n^2).
\end{aligned}
$$

Thus, calling RECURSIVESELECTIONSORT on an n element array will require roughly the same $O(n^2)$ time as calling SELECTIONSORT. Since RECURSIVESELECTIONSORT always performs the same operations on a list of size n, we can be certain that this is a tight analysis of the running time of the algorithm. This is why using SELECTIONSORT to sort a 5000-element array takes $1,000,000$ times longer than it does to sort a 5-element array: $5,000^2 = 1,000,000 \cdot 5^2$.

Of course, this does not show that the Sorting problem requires $O(n^2)$ time to solve. All we have shown so far is that two particular algorithms, RECURSIVESELECTIONSORT and SELECTIONSORT, require $O(n^2)$ time; in fact, we will see a different sorting algorithm in chapter 7 that runs in $O(n \log n)$ time.

We can use the same technique to calculate the running time of HANOITOWERS called on a tower of size n. Let $T(n)$ denote the number of disk moves that HANOITOWERS(n) performs. The following equations hold.

$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + 1 \\ T(1) &= 1 \end{aligned}$$

This recurrence relation produces the following sequence: $1, 3, 7, 15, 31, 63,$ and so on. We can solve it by adding 1 to both sides and noticing

$$T(n) + 1 = 2 \cdot T(n-1) + 1 + 1 = 2(T(n-1) + 1).$$

If we introduce a new variable, $U(n) = T(n) + 1$, then $U(n) = 2 \cdot U(n-1)$. Thus, we have changed the problem to the following recurrence relation.

$$\begin{aligned} U(n) &= 2 \cdot U(n-1) \\ U(1) &= 2 \end{aligned}$$

This gives rise to the sequence $2, 4, 8, 16, 32, 64, \ldots$ and it is easy to see that $U(n) = 2^n$. Since $T(n) = U(n) - 1$, we see that $T(n) = 2^n - 1$. Thus, HANOITOWERS is an exponential algorithm, which we hinted at in section 2.5.

Algorithm Design Techniques

Over the last forty years, computer scientists have discovered that many algorithms share similar ideas, even though they solve very different problems. There appear to be relatively few basic techniques that can be applied when designing an algorithm, and we cover some of them later in this book in varying degrees of detail. For now we will mention the most common algorithm design techniques, so that future examples can be categorized in terms of the algorithm's design methodology.

Exhaustive Search

An *exhaustive search*, or *brute force*, algorithm examines every possible alternative to find one particular solution.

For example, if you used the brute force algorithm to find the ringing telephone, you would ignore the ringing of the phone, as if you could not hear it, and simply walk over every square inch of your home checking to see if the phone was present. You probably would not be able to answer the phone before it stopped ringing, unless you were very lucky, but you would be guaranteed to eventually find the phone no matter where it was.

Dynamic Programming

Some algorithms break a problem into smaller subproblems and use the solutions of the subproblems to construct the solution of the larger one. During this process, the number of subproblems may become very large, and some algorithms solve the same subproblem repeatedly, needlessly increasing the

running time. Dynamic programming organizes computations to avoid recomputing values that you already know, which can often save a great deal of time. The Ringing Telephone problem does not lend itself to a dynamic programming solution, so we consider a different problem to illustrate the technique.

Divide-and-Conquer Algorithms

One big problem may be hard to solve, but two problems that are half the size may be significantly easier. In these cases, *divide-and-conquer* algorithms fare well by doing just that: splitting the problem into smaller subproblems, solving the subproblems independently, and combining the solutions of subproblems into a solution of the original problem. The situation is usually more complicated than this and after splitting one problem into subproblems, a divide-and-conquer algorithm usually splits these subproblems into even smaller sub-subproblems, and so on, until it reaches a point at which it no longer needs to recurse. A critical step in many divide-and-conquer algorithms is the recombining of solutions to subproblems into a solution for a larger problem. Often, this merging step can consume a considerable amount of time.

Machine Learning

Another approach to the phone search problem is to collect statistics over the course of a year about where you leave the phone, learning where the phone tends to end up most of the time. If the phone was left in the bathroom 80% of the time, in the bedroom 15% of the time, and in the kitchen 5% of the time, then a sensible time-saving strategy would be to start the search in the bathroom, continue to the bedroom, and finish in the kitchen. Machine learning algorithms often base their strategies on the computational analysis of previously collected data.

Randomized Algorithms

If you happen to have a coin, then before even starting to search for the phone, you could toss it to decide whether you want to start your search on the first floor if the coin comes up heads, or on the second floor if the coin comes up tails. If you also happen to have a die, then after deciding on the second floor, you could roll it to decide in which of the six rooms on the second floor to start your search.[17] Although tossing coins and rolling dice may be a fun way to search for the phone, it is certainly not the intuitive thing to do, nor is it at all clear whether it gives you any algorithmic advantage over a deterministic algorithm. We will learn how randomized algorithms

help solve practical problems, and why some of them have a competitive advantage over deterministic algorithms.