

MIT OpenCourseWare  
<http://ocw.mit.edu>

6.047 / 6.878 Computational Biology: Genomes, Networks, Evolution  
Fall 2008

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.

# 6.047/6.878 Lecture 6: HMMs I

September 24, 2008

## 1 Introduction

To this point, the class has dealt mainly with known, characterized gene sequences. We have learned several methods of comparing genes in hopes of quantifying divergence, highlighting similar gene segments which have been preserved among species, and finding optimal alignments of sequences such that we are better able to understand the "evolution" of the sequence. We have used BLAST, hashing, projections and neighborhood search to find sequences in a database. Having learned these methods, we now can begin to address the question:

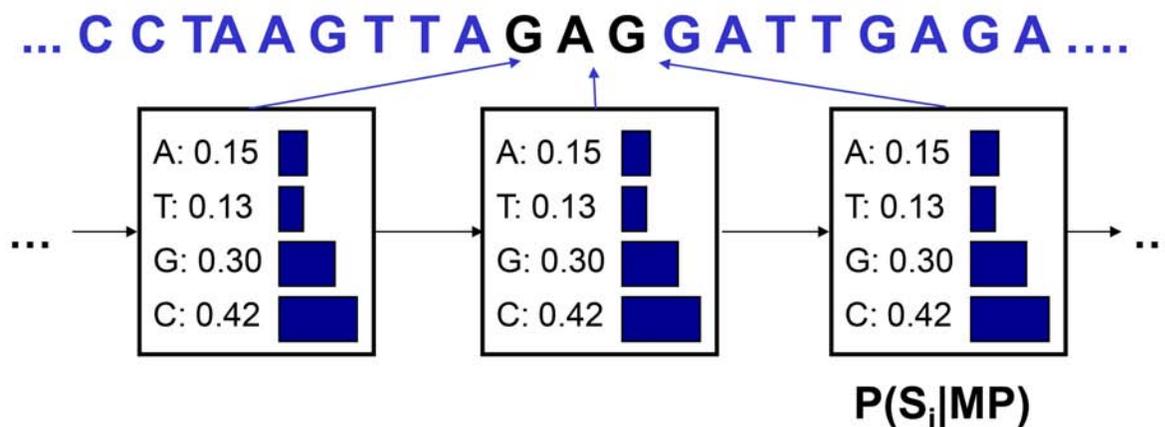
### 1.1 What do we do with a new found piece of DNA?

The first approach is usually to compare the DNA to known sequences, for which the methods mentioned work well. But many times the new sequence cannot be matched to genes within a database. Then, one could compare several unknown sequences and try to group them for later use. All these technics build on comparisons of DNA sequences and on the perseverance of homologous parts. But why not take an orthogonal approach and try to just understand the given piece itself better. This could involve analyzing the sequence for unusual properties like k-mer frequencies and motifs or recurring patterns. Here we want to explore the more constructive approach of modeling properties in a *generative model*.



## 2 The one state model

To get some feeling for the concept of modeling and for the purpose of learning (some of) the probabilistic methods and tricks needed we start with the simplest model possible namely the one state model. For this model we assume that a sequence is generated over time, while at each point of time every base has a certain globally fixed probability to occur (which is therefore independent from time, history, ...).



A simple example would be to apply this model to GC-rich regions by assuming that in such a sequence nucleotides are independent and occur with frequencies  $P(A) = 0.15, P(T) = 0.13, P(G) = 0.30$  and  $P(C) = 0.42$ . If we have now a GC-rich region sequence  $S = AAATGCCGCAATTCGAA$ , then we can infer  $P(S|MP)$ , the probability of getting this exact sequence  $S$  given our model  $MP$  by first using independence:

$$P(S|MP) = \prod_{i=1}^{|S|} P(S_i|MP)$$

This product can always be rearranged with respect to the nucleotides gives a probability only dependent on the nucleotide count. In our example this is:

$$P(S|MP) = P(A)^6 * P(T)^4 * P(G)^3 * P(C)^2 = (0.15)^6 * (0.13)^4 * (0.30)^3 * (0.42)^2 = 1.55E - 11$$

Remark:

Note that this is the probability of getting one sequence with 6As, 4Ts, 3Gs, and 2Cs. This probability needs to be multiplied by the number of permutations of this sequence in order to find the probability of getting any sequence with the given nucleotide counts. When

using the model, we assume that for any position in the generated sequence the nucleotide probabilities hold. We do not account for any dependencies among nucleotides which may come into play.

Knowing  $P(S|MP)$  alone is not a great information especially since there is hardly a good intuition as to whether or not the computed small probabilities are relatively high or not. Thus we create an alternative background model with uniform probabilities  $P(A) = P(T) = P(G) = P(C) = 0.25$  and get that the probability that the background generated our sequence  $S$  is with  $P(S|B) = 9.31E - 10$  actually nearly ten times higher. Since

$$P(S|B)/P(S|MP) > 1$$

we have gained some confidence that  $S$  is more likely not a GC-rich region but a background sequence.

Instead of using this rule we can take (the monotone) logarithm on both sides.

$$Score = \log(P(S|B)/P(S|MP)) = \log(P(S|B)) - \log(P(S|MP)) > 0$$

Using again independence leads to the so called log-likelihood ratios:

$$Score = \sum_i \log(P(S_i|MP)/P(S_i|B))$$

Computing the total score is therefore just adding up of the log-likelihood ratios for each nucleotide in the sequence. In our example these ratios are  $A : -0.73$ ,  $T : -0.94$ ,  $G : 0.26$ ,  $C : 0.74$ .

## 2.1 Use and drawback of the one state model

As we have seen we can use the one state model to decide whether a sequence is more likely a background or a GC-rich region. However, we need some extension to use this approach for finding the GC-rich parts in a whole sequence. One idea would be to take windows (e.g. of fixed size) in the sequence and decide for these windows whether they are likely to be GC-rich. Besides computational problems this does not give a satisfactory model/technic since in reality regions of one type are not of a specific length. Therefore in the next section we extend our model such that it can itself "decide" in which type of subsequence it is and freely transition between the types.

## 3 Markov Chains and Hidden Markov Models

A Markov chain  $M = (Q, A, p)$  consists of a (typically finite) set of states  $Q$ , transition state probabilities  $A$  - where  $A = (a_{i,j})$  is a stochastic matrix with probabilities  $a_{i,j}$  that  $M$

changes from being in state  $q_i$  to state  $q_j$  in one step - and the distribution  $p$  over  $Q$  giving the probabilities to start in a certain state.

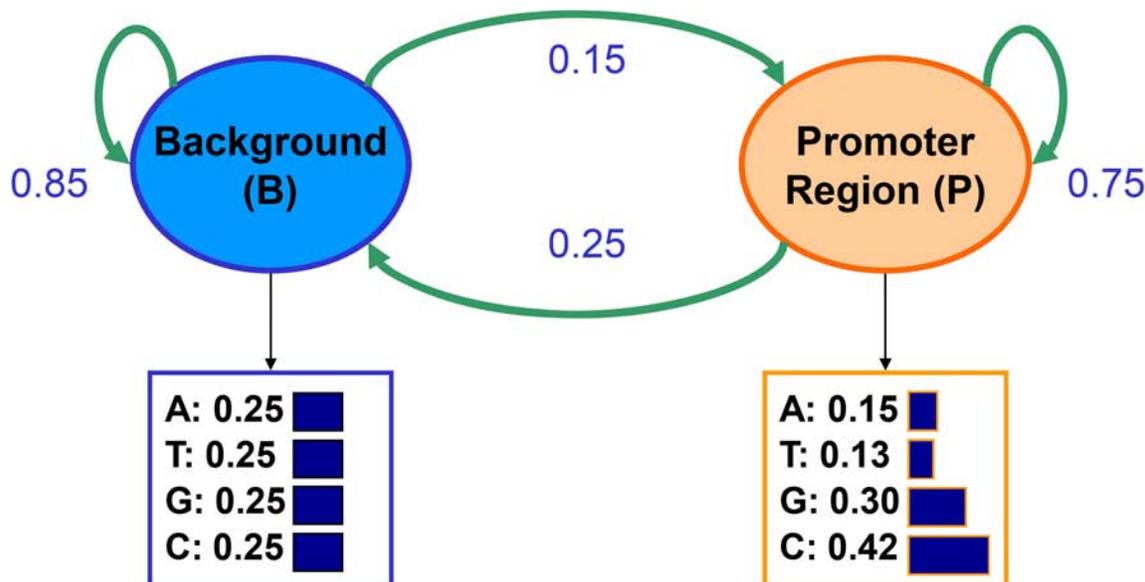
A Markov chain starts in one state (according to  $p$ ) and transitions each time step to the same or an other state (according to  $A$ ). The state trajectory of such a random walk is the generated sequence.

A useful generalization of this model is the hidden Markov model which instead of directly outputting the state sequence, couples every state with a probability distribution over an output alphabet  $\Sigma$ . In addition to transitioning every time an output is drawn from the distribution of the current state. Since it is possible that one character has non-zero probability to be generated in two (or more) different states, one can not identify the state trajectory from the output. The states are *hidden* for the observer.

Note the important property (and assumption) that the next state in each of both model only depends on the current state. Thus the model is *memoryless*.

### 3.1 Example: GC-rich regions

Having HMM we can easily make a model for a whole sequence containing both background and promoter regions:

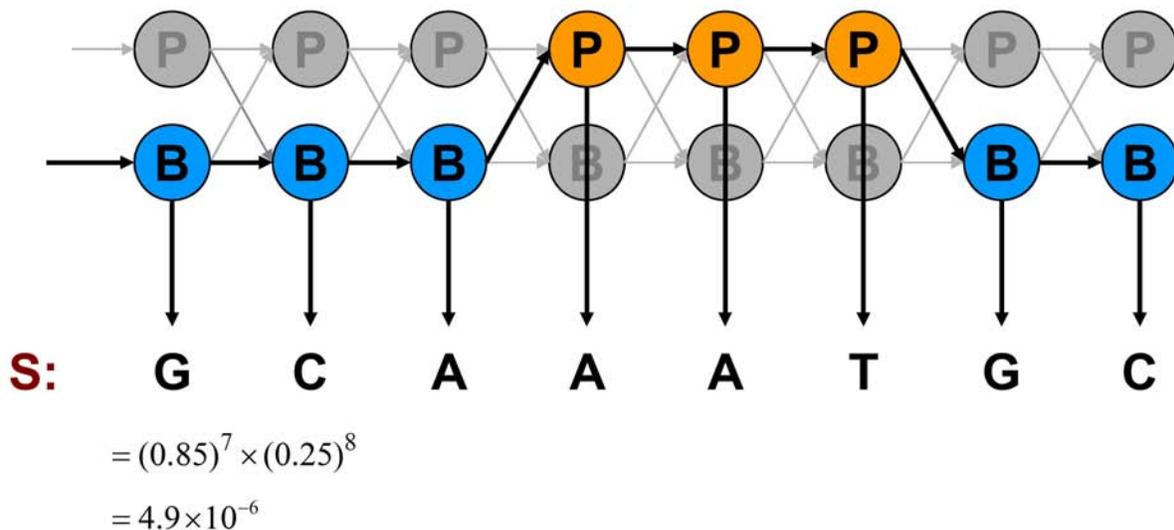


Remark:

The above diagram shows a HMM with a 85% chance that when in the background state, the next item in the sequence will also be part of the background DNA. Similarly there is a 15% chance that there will be a state transition from background to promoter . . . . Note that the probabilities of all transition choices from a given state form a distribution (their sum is 100%).

### 3.2 Using MC and HMM as a Generative Model

These models can be easily used as a generative model. Instead of getting a sequence by drawing nucleotides from one distribution as in the simple model, we now first initialize the model using the start probabilities of  $p$ , remember the current state where we are in, draw a nucleotide from the corresponding distribution and perform a state transition according to the probabilities in  $A$ . In such a way a sequence corresponds to a path in the state space.



## 4 (Algorithmic) questions related to HMMs

Having the correspondence between paths in a HMM and a sequence present several problems of interest. The simplest one is to compute the probability of a given path or parse sequence. Having made this connection between sequences, paths and probabilities we can score paths according to their probability and might want to find the single highest score path that could have generated a given sequence. This is done by the *Viterbi Algorithm*. We will also be interested in computing the *total probability* of a given sequence being generated by a particular HMM over all possible state paths that could have generated it; the method we present is yet another application of dynamic programming, known as the *forward algorithm*. One motivation for computing this probability is the desire to measure the accuracy of a model. Being able to compute the total probability of a sequence allows us to compare alternate models by asking the question: “Given a portion of a genome, how likely is it that each HMM produced this sequence?” Finally, even though we now know the Viterbi decoding algorithm for finding the single optimal path, we want to find the most likely state at any position of a sequence (given the knowledge that our HMM produced the entire sequence). This question is named *posterior decoding*. The posterior decoding - which is not given here - applies both the forward algorithm and the closely related *backward algorithm*.

	<b>One path</b>	<b>All paths</b>
<b>Scoring</b>	1. Scoring $x$ , one path $P(x, \pi)$ Prob of a path, emissions	2. Scoring $x$ , all paths $P(x) = \sum_{\pi} P(x, \pi)$ Prob of emissions, over all paths
<b>Decoding</b>	3. Viterbi decoding $\pi^* = \operatorname{argmax}_{\pi} P(x, \pi)$ Most likely path	4. Posterior decoding $\pi^{\wedge} = \{\pi_i \mid \pi_i = \operatorname{argmax}_k \sum_{\pi} P(\pi_i = k   x)\}$ Path containing the most likely state at any time point.

Notice that all questions given here want information related to all possible paths. This is critical since there is an exponential number of such possible sequences/paths. Indeed, in an

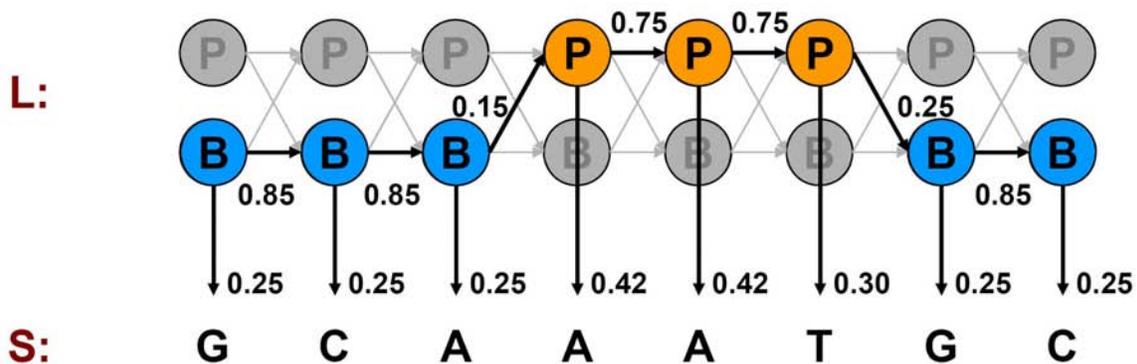
HMM with  $k$  states, at each position we can be in any of  $k$  states; hence, for a sequence of length  $n$ , there are  $k^n$  possible parses. This looks familiar to the problem we faced for computing optimal alignments. The similarities go further: As discussed in Lecture, matching states of an HMM to a nucleotide sequence is somewhat similar to the problem of alignment in that we wish to map each character to the hidden state that generated it. The only difference is that we can reuse states, i.e., return to the same state. It is therefore no surprise that the very general dynamic programming technique we used for computing alignments also helps here.

### 4.1 Likelihood of a path

But before we jump into these complexities we have a look at the evaluation of just one path. Analog to the simple example we can compute the probability of one such generation by multiplying the probabilities that the model makes exactly the choices we assumed. Again the exploitation of the independence allows an easy calculation:

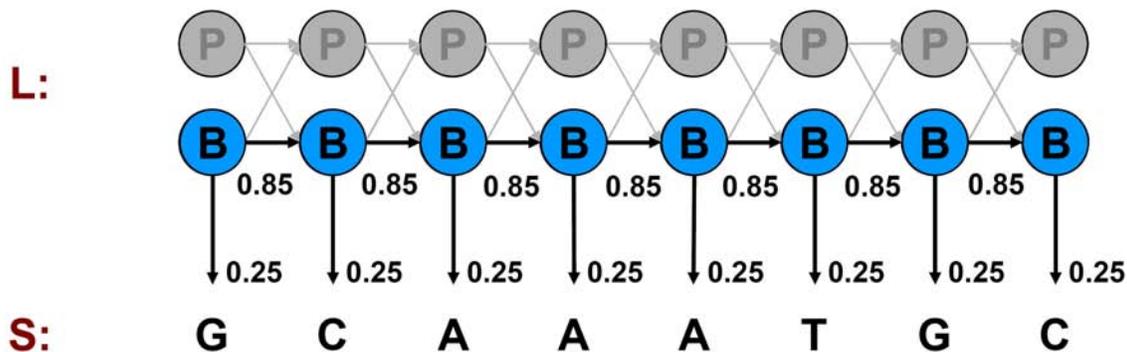
As an example are here two paths and their probabilities:

Path 1:



$$\begin{aligned}
 P &= P(G | B)P(B_1 | B_0)P(C | B)P(B_2 | B_1)P(A | B)P(P_3 | B_2)\dots P(C | B_7) \\
 &= (0.85)^3 \times (0.25)^6 \times (0.75)^2 \times (0.42)^2 \times 0.30 \times 0.15 \\
 &= 6.7 \times 10^{-7}
 \end{aligned}$$

Path 2:



$$\begin{aligned}
 P &= P(G | B)P(B_1 | B_0)P(C | B)P(B_2 | B_1)P(A | B)P(B_3 | B_2)\dots P(C | B_7) \\
 &= (0.85)^7 \times (0.25)^8 \\
 &= 4.9 \times 10^{-6}
 \end{aligned}$$

## 4.2 Viterbi Algorithm and Decoding

As promised we want now find the optimal (most likely) path creating a sequence. The reason why we can use dynamic programming for this lies in the following fact quite typical for questions on paths: Every sub path of an optimal path is optimal and vice versa every optimal path can be constructed by constantly putting together smaller optimal sub path (or prolonging a smaller path).

This leads to an the following recurrence:

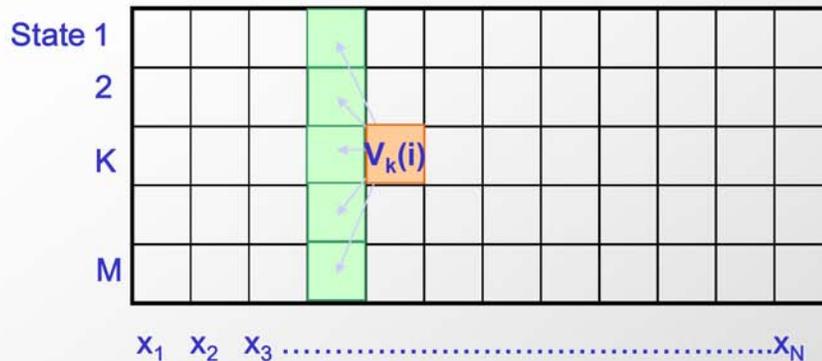
(We assume to have a single start state 0 which transits according to  $p$  to the next state)

$V_k(l)$  = Probability of the most likely path of length  $l$  from state 0 to state  $k$

$$V_k(l + 1) = e_k(x_{l+1}) * \max_j a_{j,k} V_j(l)$$

which can be solved easily:

## The Viterbi Algorithm



**Input:**  $x = x_1 \dots x_N$

**Initialization:**

$$V_0(0)=1, V_k(0) = 0, \text{ for all } k > 0$$

**Iteration:**

$$V_k(i) = e_K(x_i) \times \max_j a_{jk} V_j(i-1)$$

**Termination:**

$$P(x, \pi^*) = \max_k V_k(N)$$

**Traceback:**

Follow max pointers back  
Similar to aligning states to seq

**In practice:**

Use log scores for computation

**Running time and space:**

Time:  $O(M^2N)$

Space:  $O(MN)$

Using the log of the probabilities simplifies the algorithm slightly, since all products get transformed into sums.

### 4.3 Forward Algorithm and Evaluation

We begin by discussing the forward algorithm which, given a sequence  $X = x_1, \dots, x_n$  and an HMM which produced it, finds the total probability  $P(X)$  of  $X$  over all possible Markov paths  $\pi = \pi_1, \dots, \pi_n$  that could have produced it. We are interested in total probability for multiple reasons. One is that the Viterbi optimal path may have an extremely small probability of actually occurring; moreover, suboptimal paths may have probabilities very close to the optimum. For example, in the HMM discussed last time, suppose the Viterbi path includes a promoter region, i.e., a sequence of consecutive promoter states. Then an alternate path that merely switches from the promoter state to the background state for one base and then switches back is only slightly less probable than the optimal.

In general, suboptimal paths account for a much greater proportion of the probability distri-

bution than the Viterbi path alone, and so working with our system over all possible paths may allow us to make better inferences about the true structure of the sequence. Also, the optimal path may be very sensitive to the choice of parameters of our model, and since we do not actually know these exactly—and worse, rounding errors may affect the answer—being able to look at more paths is likely to be informative.

Finally, as discussed in the introduction, being able to compute total probability allows us to compare two HMMs. One cheap alternative to actually computing this probability is to simply calculate the probability of the Viterbi path predicted in each case, but we can do better: in fact, the forward algorithm for computing the real answer is no more expensive than Viterbi!

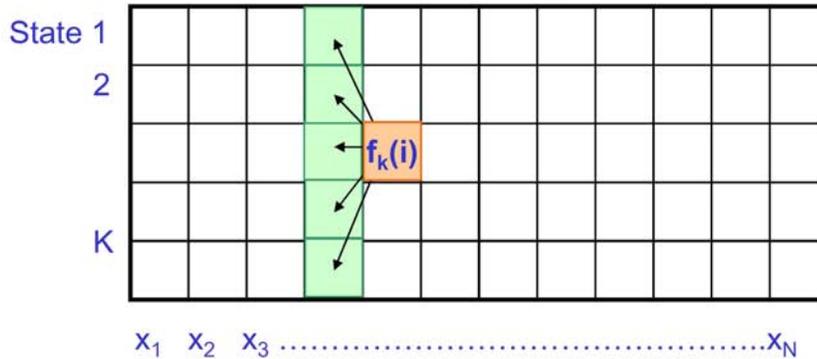
Just as Viterbi found the most likely path in polynomial time, we can find the total probability of a sequence in polynomial time—despite the challenge of an exponential number of possible parses—via the principle of dynamic programming. More precisely, we will calculate total probabilities iteratively by reusing solutions to subproblems. In this case we are not looking for the optimal solution but rather for a sum of previously computed probabilities, but the rough idea is the same as in Viterbi. Indeed, we still “expand out the problem one layer at a time,” the only difference being that this time we take a sum over previous states and transitions rather than a max:

$f_k(l)$  = Probability that the sequence up to position  $l$  was generated ending in state  $k$

$$f_k(l+1) = e_k(x_{l+1}) * \sum_j a_{j,k} f_j(l)$$

In words, we have found a recursion that allows us to compute the total probability of generating the sequence up to a position  $i$  and being in state  $l$  at that position, using the (previously computed) total probabilities of reaching position  $i-1$  and being in each possible state. Thus, we may compute each of these probabilities by filling in a dynamic programming table with columns indexed by position  $i$  and rows indexed by state  $l$ . When the algorithm completes, summing the values in the rightmost column (which takes into account all possible final states) gives the total probability of generating the entire sequence.

## The Forward Algorithm



Input:  $x = x_1 \dots x_N$

**Initialization:**

$$f_0(0)=1, f_k(0) = 0, \text{ for all } k > 0$$

**Iteration:**

$$f_k(i) = e_K(x_i) \times \text{sum}_j a_{jk} f_j(i-1)$$

**Termination:**

$$P(x, \pi^*) = \text{sum}_k f_k(N)$$

**In practice:**

- Sum of log scores is difficult
- approximate  $\exp(1+p+q)$
- scaling of probabilities

**Running time and space:**

- Time:  $O(K^2N)$
- Space:  $O(K)$

The time complexity of the algorithm is  $O(K^2N)$  because filling in each of the  $KN$  cells in the table requires using data from all  $K$  cells in the previous column; this is the same as for Viterbi. The space required is  $O(KN)$  if we wish to remember the entire table of probabilities but only  $O(K)$  if not, since we need only retain values from one previous column to compute the next.

There is one caveat to the forward algorithm which we now note. In the case of Viterbi, we were able to use log probabilities because we only multiplied probabilities of independent events. However, in the forward algorithm, we add probabilities; thus, it appears that we can no longer work with logs. Due to the small probability of generating any given sequence or part of a sequence, numerical accuracy thus becomes a concern.

Nevertheless, it turns out that we can overcome this challenge in practice by approximating when necessary via an algebra trick. We wish to find a function  $F$  giving us  $F(\log m, \log n) =$

$\log(m + n)$ . Write

$$\begin{aligned}\log(m + n) &= \log m + \log\left(\frac{m + n}{m}\right) \\ &= \log m + \log\left(1 + \frac{n}{m}\right) \\ &= \log m + \log(1 + \exp(\log n - \log m)).\end{aligned}$$

We now consider three cases:

- $m \approx n$ : All quantities in the final expression are well-behaved.
- $m \gg n$ : We can ignore the second term and approximate  $\log(m + n)$  to  $\log m$ .
- $m \ll n$ : Swap  $m$  and  $n$  and apply the previous case.

These observations allow us to keep using log probabilities, and so in fact the problem is solved.

## 5 Encoding memory in an HMM

Finally we want to describe how to incorporate “memory” into a Markov model by increasing the number of states. First we present a motivating biological example.

Recall that we built an HMM with two states: a “high-CG” promoter state with elevated probabilities of emitting Cs and Gs, and a background state with uniform probabilities of emitting all nucleotides. Thus, our previous model, which we will call HMM1, characterized promoters simply by abundance of Cs and Gs. However, in reality the situation is more complicated. As we shall see, a more accurate model would instead monitor the abundance of CpG pairs, i.e., C and G nucleotides on the same strand (separated by a phosphate on the DNA backbone, hence the name “CpG”). Note that we call these CpGs simply to avoid confusion with bonded complementary C–G bases (on opposing strands).

Some biological background will shed light on the reason for introducing the new model. In the genome, not only the sequence of bases, but also their *methylation states* determine the biological processes that act on DNA. That is, a methyl group can be added to a nucleotide, thus marking it for future reference. Proteins that later bind to the DNA (e.g., for transcription or replication purposes) “notice” such modifications that have been made to particular bases. To elaborate on one interesting example, methylation allows error-checking in the DNA replication process. Recall that replication is performed by “unzipping” the DNA polymer and then rebinding complementary bases to each strand, resulting in a pair of new double-stranded DNA helices. If, during this process, a discrepancy is found between an old strand of DNA and a new strand (i.e., a pair of bases is not complementary), then the old—correct—base is identified by its being methylated. Thus, the new, unmethylated

base can be removed and replaced with a base actually complementary to the original one.

Now, a side effect of methylation is that in a CpG pair, a methylated C has a high chance of mutating to a T. Hence, dinucleotide CpGs are rare throughout the genome. However, in active promoter regions, methylation is suppressed, resulting in a greater abundance of CpG dinucleotides. Such regions are called CpG islands. These few hundred to few thousand base-long islands thus allow us to identify promoters with greater accuracy than classifying by abundance of Cs and Gs alone.

All of this raises the question of how to encode dinucleotides in a Markov model, which by definition is “memoryless”: the next state depends only on the current state via predefined transition probabilities. Fortunately, we can overcome this barrier simply by increasing the number of states in our Markov model (thus increasing the information stored in a single state). That is, we encode in a state all the information we need to remember in order to know how to move to the next state. In general, if we have an “almost HMM” that determines transition probabilities based not only on the single previous state but also on a finite history (possibly of both states and emissions), we can convert this to a true memoryless HMM by increasing the number of states.

In our particular example, we create a Markov chain such that dinucleotide CpGs have high frequency in promoter states and low frequency in the background. To do this, we use eight states rather than just two, with each state encoding both the promoter/background state as well as the current emission. This gives an 8 state HMM. Note, however, that each emission probability in this HMM from a given state is now either 0 or 1, since the state already contains the information of which nucleotide to emit. Thus the HMM is actually a simple Markov chain as pictured here:

