

Grammatical Models

Rune Lyngsø

January 26, 2009

1 Hidden Markov Models

A process is said to be Markovian if the next state only depends on the current state. In bioinformatics a lot of phenomena are perceived to be, or at the very least modelled as, Markovian in nature. A few examples include

- nucleotide substitution models, where the rate of change to a particular nucleotide may depend on the current nucleotide but does not depend on nucleotides previously occupying the position
- the insertion/deletion model of statistical alignment, where the birth at a link or death of a nucleotide does not depend on the history of the link or nucleotide
- the network growth by gene duplication described in [1]

The Markovian nature follows readily from a mechanistic view, where complete knowledge of a system and sufficient computational resources will allow us to determine the state of the system – or distribution over possible states for stochastic systems – at any given point in the future. Markovian need not progress in time, in fact, for many bioinformatics applications the progression will be in one or more sequences. For example, whether a position in a chromosome is maternally or paternally inherited is quite accurately modelled with a dependency only on whether the previous position was maternally or paternally inherited. In alignments we also commonly assume that the homology of a pair of nucleotides only depends on the homology of the predecessor nucleotides, and not the full alignment.

Before continuing it may be informative to remember that not all processes we encounter are Markovian, at least not unless further information is included in the state space. A few examples of this include

- RNA secondary structure formation, where the possible base pairing of a nucleotide depend on the full set of preceding bases that have not been base paired

- a pathogen will often attempt to evade an immune system by mutating; if we only consider the genome of the pathogen, a mutation reverting to a previous state will usually be selected against, as the host organism is already resistant to this variant

In some cases the Markovian property may be restored by including more state information, e.g. the genome of the host organism in the pathogen case above. However, this may not always be desirable – as it can drastically increase the model universe – or even possible.

Markov models can be continuous, both in terms of transition structure and in terms of state space. An obvious example of continuous time Markov models are nucleotide substitution models, where the time at which the next substitution occurs is drawn from an exponential distribution. The Wiener process, also often called Brownian motion, combines a continuous time transition structure with a continuous state space: if we know the position W_t at time t , the position at time $t' > t$ will be normally distributed with mean W_t and variance $t' - t$, independent of the position at any time $t'' < t$. In bioinformatics we will usually only be interested in Markov processes with a discrete, or even finite, state space, like the nucleotides in a substitution model. Also, even when using continuous time models, the focus will most often be on a discrete set of time points where change occurs, and time will be integrated out to essentially render the process a discrete transition process on a discrete state space.

1.1 Hidden Markov Model Structure

An old saying is that if you have nothing else to base your forecast on, your best guess for the weather tomorrow is that it will be the same as today. Rather than random, independent realisations, weather patterns exhibit a great deal of inertia causing a good deal of dependency between consecutive time points (given they are not too far apart). We can model this with a *Markov model* as illustrated in Fig. 1, where for simplicity we assume that weather patterns can be divided into *sunshine* and *rain*¹.

Any meteorologist will tell you that the model in Fig. 1 is too simple to accurately describe weather pattern formation, even if we ignore the limited number of weather patterns included in the model. The weather we observe is caused by underlying parameters, like atmospheric pressure, that we cannot directly observe. Atmospheric pressure again depends on the amount of air sitting above a specific place, and as it takes time to shift large volumes of air around, it is really more these underlying parameters that are governed by a Markovian process. So a slightly more realistic weather model would be the one illustrated in Fig. 2, where we switch between *high* and *low pressure* states which have different distributions over the observed weather pattern.

¹The probabilities may appear unrealistic, as the model was parameterised when the author was living in California.

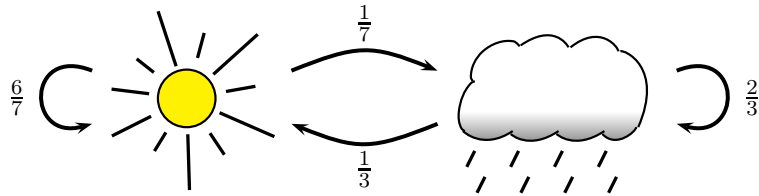


Figure 1: A Markov model for weather forecasting. It is based on the observation that with no further information, predicting that the weather tomorrow will be the same as it is today is the best option.

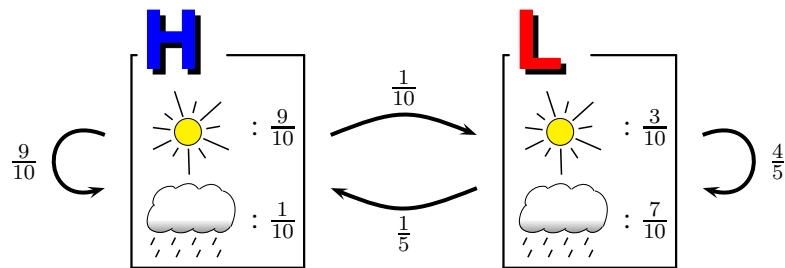


Figure 2: A hidden Markov model for weather forecasting. Usually the observable weather pattern depends on underlying hidden features, like for example the atmospheric pressure, and it is the underlying feature rather than the observable weather pattern that has an inherent inertia.

In this example, unless we have a barometer, we cannot observe the underlying state path of the underlying parameter. All we can observe are the weather patterns that could be emitted from either state. Hence, the state path is unobserved, or hidden, providing the first word of the term *hidden Markov model* (HMM) used for this type of model. One might think that the model in Fig. 2 is of little use in weather prediction, when we don't know the state we are in – high pressure or low pressure – but only a sequence of observed weather patterns for the past few days. However, as we shall see later on, having observed the sequence ☁, ☀, ☁, ☁, ☀ over the last five days would give us a probability of sunshine tomorrow of about 63%.

The weather models above describe infinite processes, where new observations are emitted indefinitely. In many situations we will be more interested in finite sequences of observations, and expect the model to include a distribution over finite lengths. A perfect

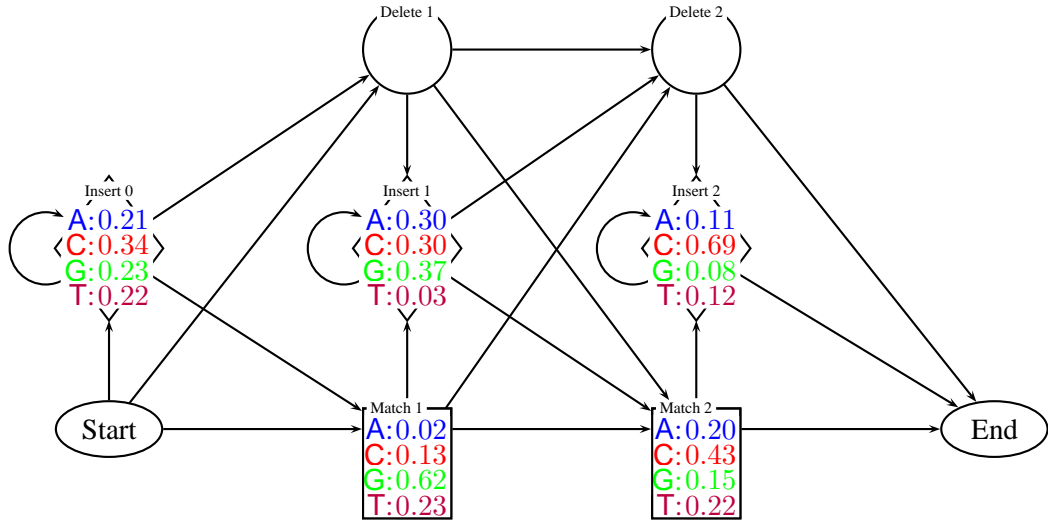


Figure 3: A profile HMM for a DNA sequence family with two positions. Only transitions with non-zero probability are shown, and for simplicity actual transition probabilities have been omitted.

example is the modelling of a family of homologous sequences. We will have a specific start state marking the start of a sequence, and a specific end state marking that we have reached the end of the sequence and no further observations will be generated. In between we will have states corresponding to a match with a position in the ancestral or consensus sequence of the family, and states allowing insertions between any two consecutive positions and deletions of one or more positions. The deletion states corresponds to no character being observed for an ancestral position, so no character should be emitted from these states. States with no emissions are generally denoted *silent* states, and all other states are *non-silent* states. An example of the type of model just described is shown in Fig. 3.

Formally, we describe an HMM M with a tuple $(Q, \Sigma, a, e, \text{start}, \text{end})$ where

- Q is the set of states in the model, e.g. $\{\mathbb{L}, \mathbb{H}\}$ in Fig. 2
- Σ is the set of possible observations, or emission alphabet, e.g. $\{\ast, \emptyset\}$ in Fig. 2
- $a_{p,q}$ describes the transition probability between states p and q in Q , e.g. $a_{\mathbb{L},\mathbb{H}} = \frac{1}{5}$ in Fig. 2; The transitions of M are usually taken to mean the pairs $p \rightarrow q$ with $a_{p,q} > 0$ and a state can have a self-transition, i.e. $a_{p,p} > 0$
- $e_{p,\sigma}$ for $p \in Q$ and $\sigma \in \Sigma$ describes the emission probability of symbol σ when entering state p , e.g. $e_{\mathbb{L},\ast} = \frac{3}{10}$ in Fig. 2; we will usually assume that states can

be classified as either silent, where $e_{p,\sigma} = 0$ for all $\sigma \in \Sigma$, or non-silent, where $\sum_{\sigma \in \Sigma} e_{p,\sigma} = 1$

- start, end $\in Q$ are designated start and end states of the model

The transition probabilities are required to specify a probability distribution over outgoing transitions for each state p , i.e. $\sum_{q \in Q} e_{p,q} = 1$, except for the end state for which $e_{\text{end},q} = 0$ for all $q \in Q$. We will also usually assume that $a_{q,\text{start}} = 0$ for all $q \in Q$, i.e. that there are no transitions to the start state, and that both start and end are silent states. Finally, we will assume that from any state $q \in Q$ the probability of reaching the end state in a finite number of steps is 1, such that M generates a finite sequence with probability 1.

The best way to view an HMM is probably as a generative model: we start in the start state, continue choosing the next state according to the transition probability of the current state, emit a character to an expanding sequence whenever we enter a non-silent state. We continue this process until we enter the end state, at which point we stop and output the constructed sequence. Denote a realisation of this process a *run*. We can decompose a run into the state path followed, $\pi = (\pi_0, \pi_1, \dots, \pi_k)$, and the sequence emitted, $s = s_1 s_2 \dots s_l$. If $\{i_1, \dots, i_l\}$ denotes the sequence of non-silent states in π , the probability of the run $P(r)$ can be written as

$$P(r) = \prod_{j=1}^k a_{\pi_{j-1}, \pi_j} \prod_{j=1}^l e_{\pi_{i_j}, s_j}$$

The probability of a sequence under a model is the sum of the probabilities of all runs generating that sequence.

1.2 HMM Algorithms

The two main uses of hidden Markov models is for annotation and classification of data. In the annotation scenario, it will usually be the case that each individual state has its own semantic connotation. By identifying an optimal run generating a sequence, we can annotate the sequence according to the states emitting each symbol. In the classification scenario, we will consider the model a generator for a set of sequences. A sequence is considered a member of the set iff the probability of generating it by the model is above some set threshold. Finally, in most cases we will not have a parameterisation of an HMM, but will have to learn this from the observed data. Usually this will take the form of a maximum likelihood estimate, i.e. finding parameters such that the probability of the data is maximised. One reason for the wide spread popularity of HMMs, making them almost ubiquitous in bioinformatics, is the existence of efficient methods for solving these central problems.

1.2.1 Viterbi Algorithm

An HMM M will usually only be interesting for annotation purposes if it allows a sequence to be generated in numerous different ways, each offering its own interpretation. In this case we will commonly be interested in the most probable way in which the sequence can be generated. Hence, given a sequence s we will want to compute the most probable run in M that generates s .

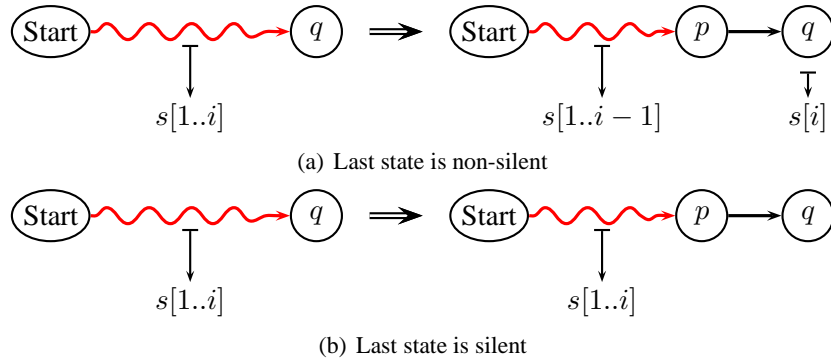


Figure 4: Unravelling the last step of a (partial) run.

A run r generating the sequence s can be decomposed into the last transition of r from some state q to the end state and a preceding *partial run*, starting in the start state and ending in q and emitting s along the way. We can further recursively decompose the preceding partial run in the same way, considering the last transition and the partial run preceding it. Whenever the last state of the partial run considered is silent, the preceding partial run must have generated the prefix of s currently considered. When the last state of the partial run considered is non-silent, it must have emitted the last symbol of the prefix of s currently considered, with the remaining prefix having been emitted by the preceding partial run. This is illustrated in Fig. 4. Of course we do not know which state p precedes q in the most probably run generating s , but maximising over all possibilities allows us to determine this. If $V(q, i)$ denotes the maximum probability of any partial run ending in state q and generating the sequence $s[1..i]$, we obtain the following recursions for computing $V(q, i)$ based on the above considerations:

$$V(q, i) = \max_{p: a_{p,q} > 0} \begin{cases} a_{p,q} V(p, i) & \text{if } q \text{ is silent} \\ a_{p,q} e_{q,s[i]} V(p, i-1) & \text{if } q \text{ is non-silent} \end{cases} \quad (1)$$

The boundary conditions for this recursion are that $V(\text{start}, 0) = 1$ and $V(q, i) = 0$ for $i < 0$ and all q . The maximum probability of a run generating s equals $V(\text{end}, |s|)$.

The recursion in (1) will usually allow an efficient dynamic programming algorithm, known as the *Viterbi* algorithm, for computing the probability of the maximum probability

run in time $O(m|s|)$ and space $O(n)$ where n is the number of states and m is the number of transitions in M (for every symbol in s and state q in M we need to consider all incoming transitions to q). This probability can be traced back to obtain the most probable run. For example, for the sequence ☁, ⚡, ☁, ☁, ⚡ the most probable path through the model in Fig. 2 will always end in state **L**, regardless of the distribution over initial state. Taking this to be the actual state of the last day of observations, this yields a probability of 42%. This does however completely ignore the fact that, though **L** is the end state of the most probable path, there is still a significant probability that the state of the last observation was **H**. In Sec. 1.2.2 we will see how to include information from all paths, rather than just the most probable path.

The progression in the sequence in (1) usually prevents cyclic dependencies. However, if there is a cycle of silent states, i.e. silent states $\{q_1, \dots, q_k\}$ with $a_{q_k, q_0} > 0$ and $a_{q_i, q_{i+1}} > 0$ for $1 \leq i < k$, (or more generally a strongly connected component of silent states) then there will be a cyclic dependency among the values of $V(q_i, j)$ for any j . In the rare circumstances where it is convenient, or even necessary, to include such a cycle in the model, we can still find the maximum probability run generating s by borrowing the relaxation technique [7, Ch. 25.1] of shortest paths algorithms in graphs. Each step added to a partial run decreases its probability. This means that if we compute $V'(q_i, j)$ according to (1), but ignoring $V(q_i, j)$ terms, then $\max_i \{V'(q_i, j)\} = \max_i \{V(q_i, j)\}$. Hence, the q_i with maximal $V'(q_i, j)$ has $V(q_i, j) = V'(q_i, j)$, so we can fix this value, update the remaining $V'(q_i, j)$ values by including the $V(q_i, j)$ term, and repeat until all values have been fixed. This can be done without increasing time requirements by more than a factor of $O(\log k)$.

The annotation found by the Viterbi algorithm is the annotation of s most likely to be correct, if s was indeed generated by M . For just medium length sequences, even the most likely annotation will still have negligible probability of being correct. Hence, this score might not be our best choice of optimality for obtaining a good annotation. There is an increasing realisation that in many cases we will be better off finding an annotation with the maximum expected number of symbols that are annotated correctly. In Sec. 1.2.2 we shall see how we for each symbol $s[i]$ in s can compute the probability $P(q, i)$ that it was emitted by a state q . However, just annotating each symbol with the state most likely to emit it could lead to an annotation inconsistent with the HMM structure. For example, for the profile HMM in Fig. 3 we may have more than one symbol annotated with the same match state, which would be inconsistent with each position in an ancestral sequence having at most one homologue in extant sequences. To find an annotation consistent with the transition structure of M but maximising the expected number of symbols correctly annotated we can apply a modified version of the Viterbi algorithm based on the recursion

$$V_{\text{MAP}}(q, i) = \max_{p: a_{p,q} > 0} \begin{cases} V_{\text{MAP}}(p, i) & \text{if } q \text{ is silent} \\ P(q, i) + V_{\text{MAP}}(p, i-1) & \text{if } q \text{ is non-silent} \end{cases} \quad (2)$$

with boundary conditions $V_{\text{MAP}}(\text{start}, 0) = 0$ and $V_{\text{MAP}}(q, i) = -\infty$ for $i < 0$ and all q .

1.2.2 Forward & Backward Algorithms

When using an HMM M for classification, we need to be able to compute the total probability that a sequence s is generated by M , rather than just the maximum probability of any run generating M . We can, however, proceed in exactly the same way, using the recursive decomposition of partial runs illustrated in Fig. 4 but summing over possible predecessors p to q rather than maximising. This leads to the recursion for the *forward* algorithm,

$$F(q, i) = \sum_{p: a_{p,q} > 0} \begin{cases} a_{p,q} F(p, i) & \text{if } q \text{ is silent} \\ a_{p,q} e_{q,s[i]} F(p, i-1) & \text{if } q \text{ is non-silent} \end{cases} \quad (3)$$

again with boundary conditions $F(\text{start}, 0) = 1$ and $F(q, i) = 0$ for $i < 0$ and all q . Here $F(q, i)$ denotes the total probability of generating the prefix $s[1..i]$ of s on a partial run ending in state q , and the total probability of generating s from M is $F(\text{end}, |s|)$. Whereas the Viterbi recursions relied on the fact that $\max_{a \in A} \{a \cdot b\} = \max_{a \in A} \{a\} \cdot b$, the forward recursions rely on the fact that $\sum_{a \in A} (a \cdot b) = (\sum_{a \in A} a) \cdot b$.

Again we can use the recursion of (3) to define an efficient dynamic programming algorithm for computing the probability of M generating s , and again the presence of a strongly connected component of silent states will be a complicating factor as it introduces cyclic dependencies. In this case we cannot apply relaxation techniques to address the cyclic dependency problem. However, one can identify (3) as a system of linear equations over variables $F(q, i)$ and solve it accordingly. This may increase time requirements to $O(n^3|s|)$ in the worst case.

If we return to our example sequence of ☁, ✨, ☁, ☁, ✨ and analyse it under the model in Fig. 2, we get a probability of the next observation being ✨ of 62.4% if we start in state **L** and of 63.9% if we start in state **HL**. For any distribution on initial state the probability will fall between these two extreme values, confirming our previous claim that the probability of the next observation being ✨ is around 63%. This is in marked contrast to the 42% computed using a Viterbi algorithm approach, and highlights the difference between only considering the most probable path and considering all paths.

So far we have only described how to compute the total probability $P(s)$ of generating the entire sequence s by M . In the previous section we also promised a method for computing the probability $P(q, i)$ of the i 'th symbol of s being emitted by state q . Evidently we could add boundary conditions $F(p, i) = 0$ for $p \neq q$ to (3) to compute the probability of generating s under the restriction that $s[i]$ is emitted by q ; dividing by $P(s)$ we get $P(q, i)$.

If we are only interested in $P(q, i)$ for one choice of q, i , this may be acceptable. However, to compute V_{MAP} we will need $P(q, i)$ for all q, i pairs, in which case the

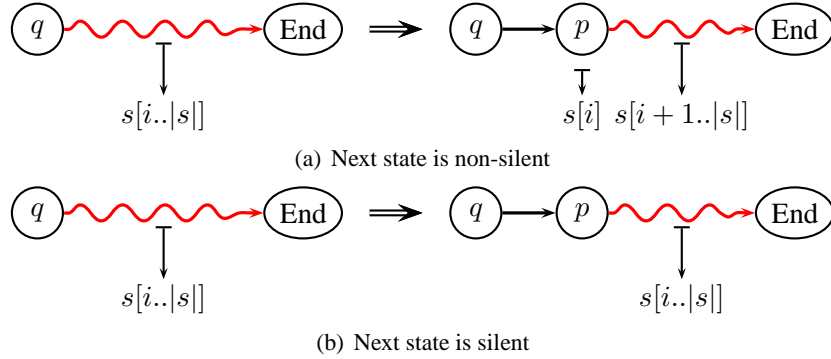


Figure 5: Unravelling the first step of a partial run.


approach becomes unnecessarily cumbersome. For the Viterbi and forward algorithms we chose to decompose runs into the last step and a preceding partial run. However, an equally valid decomposition would be into the first step and a succeeding partial run, as illustrated in Fig. 4. This decomposition leads to recursions

$$B(q, i) = \sum_{p: a_{q,p} > 0} \begin{cases} a_{q,p} B(p, i) & \text{if } p \text{ is silent} \\ a_{q,p} e_{p, s[i+1]} B(p, i+1) & \text{if } p \text{ is non-silent} \end{cases} \quad (4)$$

with boundary conditions $V(\text{end}, |s|) = 1$ and $V(q, i) = 0$ for $i > |s|$ and all q . Combining a partial run ending with emission of $s[i]$ from q with a partial run beginning immediately after emission of $s[i]$ from q we get a full run generating s and emitting $s[i]$ from q . Hence, $P(q, i) = F(q, i)B(q, i)/P(s)$.

1.2.3 Baum-Welch Algorithm

So far we have assumed that we are working with fully parameterised models. In most cases, we will have sufficient knowledge about the application area to reasonably unambiguously decide on the hidden Markov model structure, i.e. which transitions that can have non-zero probability, symbol emission restrictions, correlations between parameters, etc. For example, in the HMM in Fig. 3 it is natural that we need to go through an intermediate state representing an ancestral position to get from one insert state to the next. However, the exact values of the emission and transition parameters usually have to be inferred from data. This inference will normally take the form of attempting to obtain a maximum likelihood estimate for the parameters, i.e. set parameters such that the probability of the data is maximised.

If we have data with known annotation, e.g. we borrowed a barometer to obtain the , the MLE parameters are simply the observed frequencies in the data.

This follows from the fact that

$$\arg \max_{\mathbf{x}: \bar{\mathbf{x}}=1} \left\{ \prod_{i=1}^n p_i^{x_i} \right\} = (p_1/\bar{p}, \dots, p_n/\bar{p}) \quad (5)$$

where $\bar{\mathbf{x}} = \sum_{i=1}^n x_i$ and $\bar{\mathbf{p}} = \sum_{i=1}^n p_i$. In our continuing example, we would e.g. set $e_{\mathbb{H}, \mathbb{H}} = 1/2$ and $a_{\mathbb{L}, \mathbb{H}} = 1/3$.

Most often the annotation will not be known and all we have is sequences of unannotated observations. We can still define the parametrisation problem as finding MLE parameters. However, with unannotated data there is no analytical solution to this problem. With annotated data we had observed frequencies with which events happened. This is absent in the case of unannotated data, but using the forward-backward algorithms of Sec. 1.2.2 we can still find *expected* frequencies with which events happen, given a particular parameterisation of the HMM. For example, given sequences s and a non-silent state q we can find new emission parameters for q according to

$$e'_{q,\sigma} = \frac{\sum_{i:s[i]=\sigma} P(q, i)}{\sum_i P(q, i)}. \quad (6)$$

The update for transition probabilities looks slightly more complex, as we have not yet had the need to define the probability of utilising a specific transition. We can, however, still use the forward and backward values ending and starting at the source and target of a transition, respectively, to obtain

$$a'_{p,q} = \begin{cases} \frac{\sum_i F(p,i) a_{p,q} e_{q,s[i+1]} B(q,i+1)/P(s)}{\sum_{r:a_{p,r}>0} F(p,i) B(r,i)/P(s)} & \text{if } q \text{ is non-silent} \\ \frac{\sum_i F(p,i) a_{p,q} B(q,i)/P(s)}{\sum_{r:a_{p,r}>0} F(p,i) B(r,i)/P(s)} & \text{if } q \text{ is silent} \end{cases}. \quad (7)$$

Applying these update rules are guaranteed to not decrease the probability of s [2], and the general method of iterating the updates until a stopping criteria is met is known as the Baum-Welch algorithm for training an HMM on data with unknown annotation.

In both these update rules the denominator simply normalises the expected counts by the total expected number of expected emissions and transitions, respectively, from the state q . With more than one sequence, this normalisation should only occur once we have summed expectations over all sequences. That is, we should sum over sequences in both the numerator and the denominator, which is also the reason $P(s)$ has been explicitly included in both the numerator and denominator of (7) despite the fact that they cancel out for a single sequence. Just averaging updated values over all sequences wouldn't account for the fact that some sequences contain more information about the behaviour of a specific state and that long sequences contain more information in general than short sequences.

Returning to our example with annotated data at the beginning of this section, we can observe that we would obtain the estimate $a_{\text{H,H}} = 1$ and correspondingly $a_{\text{H,L}} = 0$. One would probably be rather suspicious about this parameterisation, which means that once we reach the high pressure state we stay in the high pressure state forever. The dubiousity of this parameterisation is further reinforced by the fact that it is based on just a single observed transition out of the high pressure state. With small data sets we usually run the risk of over fitting the model, as we do not have sufficient data to obtain reliable statistics for all variables. The standard, and simplest, way to address this problem is to add a small number of observations of each type of event, also called pseudocounts, before computing observation frequencies. Similarly, in (6) and (7) we would add these pseudocounts to the expected number of observations of each type. With no prior knowledge we would normally add 1 to observed or expected value for each parameter.

The pseudocounts do however not have to be identical, nor do they have to be integral. We can adjust pseudocounts to reflect prior knowledge, e.g. in Fig. 2 we would probably use a higher pseudocount for sunshine in than for rain in the high pressure state. We can even go step one further and let the pseudocounts depend on the observed or expected emission pattern from a state. Assume that we expect the emission probability of a state to be from one of a set of distributions $\{\rho_1, \dots, \rho_k\}$, with distribution ρ_i being chosen with probability q_i . If \mathbf{n} denotes the observed or expected number of emission of each $\sigma \in \Sigma$, we can compute the posterior probability of having emitted according to distribution ρ_i by Bayes rule:

$$\Pr(\rho_i | \mathbf{n}) = \frac{q_i \Pr(\mathbf{n} | \rho_i)}{\sum_{j=1}^k q_j \Pr(\mathbf{n} | \rho_j)} \quad (8)$$

If each distribution comes with pseudocount priors, we can add these to observed or expected counts according to the posterior probability of the distribution for each state.

An example of this principle applied to HMMs for families of protein sequences is presented in [4]. For DNA sequences it may be difficult to find an interesting set of ‘typical’ distributions, though sets representing purines and pyrimidines, respectively, could be a candidate. For the more general stochastic models applicable to RNA secondary structure modelling that we will discuss in Sec. 2, it would be natural to investigate a similar approach with distributions either based on or compared to the isostericity classes described in [12].


1.3 HMM Uses

2 Stochastic Grammars

Transformational grammars are generally used to refer to models where we are repeatedly allowed to replace a subcomponent of a structure by another component according to a

given set of rules. In sequence terms the rules would usually specify that the occurrence of a specific sequence as subsequence in our current sequence can be replaced by another sequence. Sequence generating grammars are categorised according to the generality of the replacement rules. We will pay special attention to the two simplest types of the four types of grammars constituting the so-called Chomsky hierarchy [5], and briefly discuss other grammar types at the end.

2.1 Regular Grammars

When emitting sequences according to the hidden Markov model of Fig. 2, we would usually think of this as writing down the sequence emitted so far and remembering the current state. For example, to emit the (annotated) sequence  we would go through the stages of $(, \mathbf{L})$, (\mathbf{L}, \mathbf{L}) , (\mathbf{L}, \mathbf{L}) , (\mathbf{L}, \mathbf{H}) , (\mathbf{L}, \mathbf{H}) , and (\mathbf{L}, \mathbf{H}) . However, rather than keeping the emitted sequence and the current state as two separate entities of a pair, we might as well think of them as one sequence with a symbol representing the current state at the end. In each step we then replace the current state symbol with a new observation and a new current state symbol. If we let S be a special current state symbol indicating that we need to choose the initial state, we would get the sequence $S \Rightarrow \mathbf{L} \Rightarrow \mathbf{L} \Rightarrow \mathbf{L} \Rightarrow \mathbf{L} \Rightarrow \mathbf{L} \Rightarrow \mathbf{L}$.


Formally, we describe a sequence generating grammar by a tuple (V, Σ, P, S) where

- V is a (finite) set of variables, e.g. in the grammar sketched above we would have $V = \{S, \mathbf{L}, \mathbf{H}\}$
- Σ is a (finite) set of terminal symbols, e.g. in the grammar sketched above we would have $\Sigma = \{\mathbf{L}, \mathbf{H}, \star\}$
- P is a (finite) set of productions of the type $x \rightarrow y$ where $x \in (V \cup \Sigma)^* V (V \cup \Sigma)^*$ (i.e. a sequence of variables and terminal symbols containing at least one variable) is the left hand side and $y \in (V \cup \Sigma)^*$ (i.e. any, possibly empty, sequence of variables and terminal symbols) is the right hand side – applying a production consists of replacing an occurrence of the left hand side with the right hand side; in the grammar sketched above we would have the following production rules in P :

$$\begin{aligned}
 S &\rightarrow \mathbf{L} \mid \mathbf{H} \\
 \mathbf{L} &\rightarrow \star \mathbf{L} \mid \mathbf{L} \mid \star \mathbf{H} \mid \mathbf{L} \mathbf{H} \mid \star \mid \mathbf{L} \\
 \mathbf{H} &\rightarrow \star \mathbf{L} \mid \mathbf{L} \mid \star \mathbf{H} \mid \mathbf{L} \mathbf{H} \mid \star \mid \mathbf{L}
 \end{aligned}$$

Here we have used the standard abbreviated notation where, instead of listing all production rules explicitly, for each left hand side we list all possible right hand sides it can be replaced with, e.g. $S \rightarrow \mathbf{L} \mid \mathbf{H}$ rather than the more cumbersome $\{S \rightarrow \mathbf{L}, S \rightarrow \mathbf{H}\}$.

- $S \in V$ is the special starting variable, e.g. in the grammar sketched above we would have S as this special starting variable

When deriving a sequence according to a grammar we start from the sequence consisting of only the special starting variable S and repeatedly transform the sequence according to the production rules. We continue doing this until the current incarnation of our sequence contains no variables, and this sequence is the result of the derivation. One example of how to derive the sequence  in the grammar based on the HMM in Fig. 2 is given above.

If we compare the format of production rules in the formal definition to the production rules in our example grammar, we can see that the latter are quite simplistic compared to the allowed range of possibilities:

- The left hand side of all the productions consists of just a single variable
- The right hand side of all productions consists of zero or one terminal symbols followed by zero or one variables

Grammars where all productions fall in one of the three categories

- $U \rightarrow \epsilon$ where $U \in V$ is a variable and ϵ is the empty sequence (note that in some texts λ is used to denote the empty sequence)
- $U \rightarrow \sigma$ where $U \in V$ is a variable and $\sigma \in \Sigma$ is a terminal symbol
- $U \rightarrow \sigma V$ where $U, V \in V$ are variables and $\sigma \in \Sigma$ is a terminal symbol

are called right regular grammars. In some definitions the first type of productions, replacing a variable with the empty sequence, are not allowed. This difference is of limited consequence, though.

In our example, the only production rules not falling in any of these three categories are $S \rightarrow \text{L}$ and $S \rightarrow \text{H}$. These two productions do however just replace the variable S with another variable. By simply replacing these production rules with production rules allowing S to be replaced with whatever either L or H can be replaced with, we would obtain an equivalent right regular grammar (where production rules for S in the new grammar essentially correspond to two production rule applications in the old grammar). Similarly, if we had a production rule of the form $U \rightarrow \sigma_1 \sigma_2 \dots \sigma_k V$ we could transform it to rules on the correct form by introducing new variables U_1, U_2, \dots, U_{k-1} and replace the production with productions $U \rightarrow \sigma_1 U_1, U_1 \rightarrow \sigma_2 U_2, \dots, U_{k-1} \rightarrow \sigma_k V$. Here a single production in the old grammar would correspond to k production rule applications in the new grammar. Grammars where productions are restricted to

- $U \rightarrow x$ where $U \in V$ is a variable and $x \in \Sigma^*$ is a finite, possibly empty, sequence of terminal symbols

the HMM in Fig. 2 is equivalent to the grammar

$$\begin{aligned}
 S &\rightarrow \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \\
 &\qquad \frac{1}{2} \quad \frac{1}{2} \\
 \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} &\rightarrow \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \\
 &\qquad \frac{6}{25} \quad \frac{14}{25} \quad \frac{3}{50} \quad \frac{7}{50} \\
 \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} &\rightarrow \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} \mid \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array} , \\
 &\qquad \frac{9}{100} \quad \frac{1}{100} \quad \frac{81}{100} \quad \frac{9}{100}
 \end{aligned}$$

with a uniform initial distribution on the states $\begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}$ and $\begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}$. This is assuming that the HMM/grammar is already parameterised. If unparameterised, the HMM has only five free parameters – one transition and one emission probability for each state, and a probability describing the initial distribution – while the grammar has six free parameters – one for the start variable production rules and three for each of the other variables’ production rules. This is due to emissions and transitions being coupled in the production rules of the grammar. These can easily be decoupled, e.g. as in the grammar

$$\begin{aligned}
 S &\rightarrow \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} \mid \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} \\
 \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} &\rightarrow \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} \mid \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} \\
 \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} &\rightarrow \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} \mid \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} \\
 \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} &\rightarrow \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} \mid \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} \\
 \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} &\rightarrow \begin{array}{c} \color{red}{\text{L}}_e \\ \color{blue}{\text{H}}_e \end{array} \mid \begin{array}{c} \color{red}{\text{L}}_t \\ \color{blue}{\text{H}}_t \end{array} ,
 \end{aligned}$$

which has five left hand sides, each with two production rules and hence one free parameter.

It is straight forward to convert a grammar in (extended) left or right regular form to an HMM, and subsequently apply the algorithms described in Sec. 1.2 for parameterisation and data analysis. When choosing an HMM or regular grammar as modelling tool, one should however be aware of the limitations of the formalism. By its Markov property, it does not allow arbitrary long range dependencies. One can quite easily model a dependency on a fixed amount of information, for example whether an ATG start codon has been encountered in a DNA sequence. However, keeping track of the number of times a pattern has been repeated is beyond the capabilities of HMMs/regular grammars, as seen in the following example.

Example 1 Assume we want to construct a (finite) HMM that emits the sequences $\begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}^i \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}^i$, i.e. i $\begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}$ s followed by the same number of $\begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}$, with probability 2^{-i} for $i > 0$, and all other sequence with probability 0. Let M be such an HMM and let n denote the number of states in M . Let r be a run in M that emits the sequence $s = \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}^n \begin{array}{c} \color{red}{\text{L}} \\ \color{blue}{\text{H}} \end{array}^n$ with non-zero probability. As $2n$ symbols are emitted there must be a non-silent state q that is visited at least twice by r . Now divide s into

Evidently the above grammar cannot be regular. It is an example of a more general type of grammars, called context-free grammars (CFGs, or SCFGs for their stochastic versions). Context free grammars are grammars where productions are only restricted on the left hand side, i.e. all productions have

- a left hand side that consists of just a single variable
- a right hand side that can be any sequence of variables and terminal symbols

The context-free part of the name for this type of grammar reflects the fact that production rules are not allowed to depend on the context of the variable being replaced, but only the variable itself.

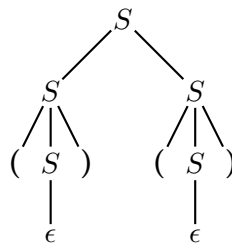


Figure 7: Derivation tree corresponding to the derivation of the sequence $()()$ by one application of the production rule $S \rightarrow SS$, and two applications each of the production rules $S \rightarrow (S)$ and $S \rightarrow \epsilon$.

The grammar above actually does not utilise the full power of context-free grammars, as at any time we will have at most one variable in the sequence being derived. An example epitomising context-free grammars is the set of all balanced sequences of parentheses. A sequence of parentheses is balanced if every left (or opening) parenthesis has a matching right (or closing) parenthesis later in the sequence. In formal terms, a sequence $s \in \{(\,)\}^*$, i.e. of left and right parentheses, we can recursively define balanced sequences of parentheses as

- the empty sequence
- a (, followed by a balanced sequence of parentheses, followed by a)
- two balanced sequences of parentheses concatenated.

This immediately suggests the context-free grammar

$$S \rightarrow \epsilon \mid (S) \mid SS$$

To generate e.g. the sequence $()()$ we will need to proceed through a sequence containing at least two variables at some point, e.g. as in the derivation $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$. The descendant sequences to each of the two occurrences of S in SS each has to be a balanced sequence of parentheses. However, due to the context-free limitation, once the initial S has been split into SS there is no possibility of coordinating the descendant sequences of each S . Where HMMs allow the modelling of *sequential* dependencies – the next state only depends on its predecessor – CFGs allow the modelling of *hierarchical* dependencies, i.e. dependencies that have a tree-like structure.

This dependency structure is clearly seen in Fig. 7, where the derivation of the sequence $()()$ listed above is given by a so called derivation tree. A derivation tree is a tree with internal nodes labelled by variables, leaves labelled by terminal symbols (or ϵ), and where the children of an internal node are the symbols – variables and terminal symbols – of the right hand side of the production rule used to replace the variable in the derivation. The final sequence of the derivation is read off the leaves from left to right. Any derivation in a CFG can be represented by a derivation tree, and as there is an equivalence between trees and sequences with balanced parentheses – as utilised e.g. in the Newick’s 8:45 format for describing phylogenies – the parenthesis example captures exactly the capabilities and limitations of CFGs. We have already mentioned the capabilities in terms of capturing hierarchical dependencies. Conversely, the formalism is limited in not being able to capture generally crossing dependencies, as for example the ones present in the following example.

Example 2 Assume we want to construct a CFG that allow us to derive exactly the set of sequences $\star^i \bowtie^i \circlearrowleft^i$ over the three letter alphabet $\{\star, \bowtie, \circlearrowleft\}$. Let G be such a grammar with n variables and let d denote the maximum number of symbols on the right hand side of any production rule in G . Consider a derivation tree T for the sequence $s = \star^m \bowtie^m \circlearrowleft^m$, where $m = d^n$. There must be a leaf at depth at least $n + 1$ in T , i.e. a leaf where we go through at least $n + 1$ internal nodes on the path connecting the leaf to the root of T . Hence there must be a variable V occurring at least twice on this path. We can now, similarly to what we did in Ex. 1, split s into five parts as $s = vwxyz$ where

- v and z are the terminal symbols descendant from the initial S , but not descendant from any occurrences of V
- w and y are the terminal symbols descendant from the first occurrence of V on the path, but not descendant from the last occurrence of V on the path
- x are the terminal symbols descendant from the last occurrence of V on the path

as illustrated in Fig. 8. As we shall see in Sec. 2.2.1, we can assume that at least one of w and y are not the empty sequence. By replicating the part of the derivation tree between

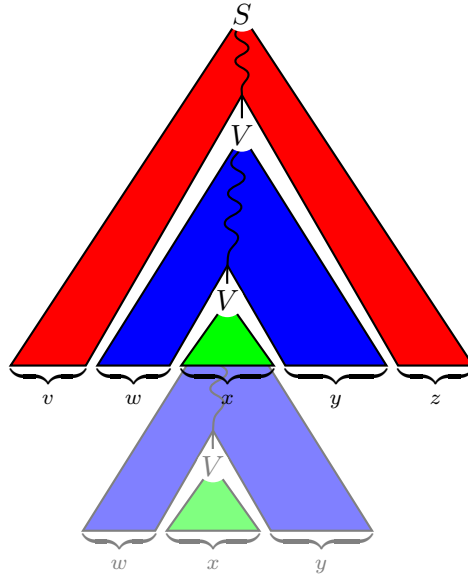


Figure 8: When deriving a sufficiently long sequence there will be at least one variable that is revisited on a path from the root of the derivation tree to a leaf. We can replicate the part of the derivation tree between two occurrences of the same variable.

the two occurrences of V , again illustrated in Fig. 8, we can construct a derivation tree for the sequence $vwxyz$ using the production rules of G . But this sequence cannot be of the form $\ast^i \text{ } \text{ } \text{ }^i$. If either w or y consists of at least two different symbols it will not have the ordering of all \ast s before all $\text{ } \text{ } \text{ }$ s before all $\text{ } \text{ } \text{ }$ s. If not, as at least one of w and y is not the empty sequence there will be at least one of the three symbols with more than m repetitions and at least one of the three symbols with exactly m repetitions.

Example 3 A curious fact of CFGs is that we can easily define a grammar generating all palindromic sequences, but it is not possible to define grammars that generate all sequences consisting of two identical copies of the same sequence. For palindromic sequences, i.e. sequences that read the same front-to-back as back-to-front, we just need production rules $S \rightarrow \sigma S \sigma$ and $S \rightarrow \sigma$ for all $\sigma \in \Sigma$, as well as a null production for S .

Assume there was a grammar G generating exactly the sequences s where $s = uu$ for some $u \in \{\ast, \text{ } \text{ } \text{ }\}^*$. Then it can generate the sequence $\ast^m \text{ } \text{ } \text{ }^m \ast^m \text{ } \text{ } \text{ }^m$, where $m = d^{n+1}$ with n the number of variables in G and d the maximum right hand side length. We can again identify a situation as illustrated in Fig. 8, but with the further restriction that the length of the sequence wxy is at most m : start from a subtree rooted at an internal node having longest paths to a descendant leaf going through exactly n further internal nodes.

It follows that wxy , and consequently both w and y , consist of at most two blocks of

identical symbols. If either w or y contains both \star and \square the other cannot, and vw^2xy^2z will contain three blocks each of \star s and \square s. If both w and y consist of just one block of identical symbols, then $vw^2xy^2z = \star^a \square^b \star^c \square^d$ where either $a \neq c$ or $b \neq d$ as one block of either \star s or \square s is extended but the other is not. If either w or y consists of both a \star block and a \square block

In either case, $vw^2xy^2z \neq uu$ for all $u \in \{\star, \square\}^*$. Hence, CFGs can capture reverse-order dependencies of any length, but not linear-order dependencies.

As previously mentioned, we obtain a stochastic context-free grammar (SCFG) if we for each variable assign a probability distribution over the production rules having the variable as left hand side. The probability of a derivation is just the product of the probabilities of the production rules applied. It would seem obvious that the probability of deriving a particular sequence is just the sum over the probabilities of all derivations generating it. However, we need to be a little bit careful when defining the set of such derivations.

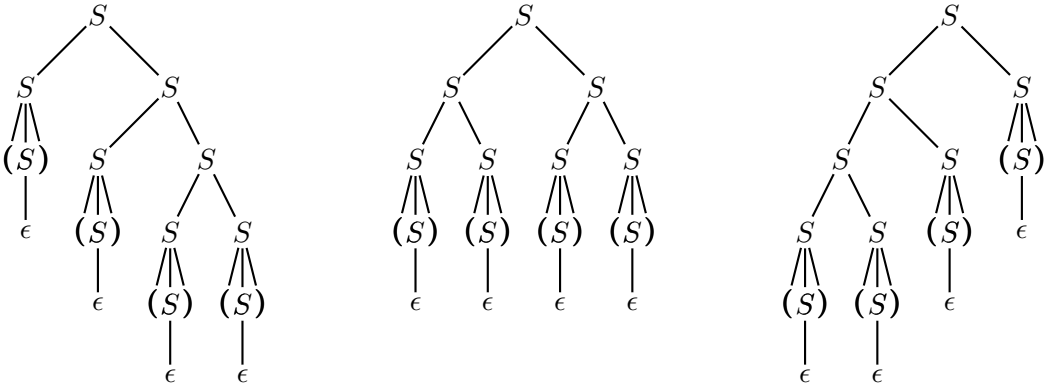


Figure 9: Three different derivation trees for the sequence $()()()$ in the parenthesis grammar, corresponding to the leftmost derivations $S \Rightarrow SS \xrightarrow{2} ()S \Rightarrow ()SS \xrightarrow{2} ()()S \Rightarrow ()()SS \xrightarrow{2} ()()()S \xrightarrow{2} ()()()()$, $S \Rightarrow SS \Rightarrow SSS \xrightarrow{2} ()SS \xrightarrow{2} ()()S \Rightarrow ()()SS \xrightarrow{2} ()()()S \xrightarrow{2} ()()()()$, and $S \Rightarrow SS \Rightarrow SSS \Rightarrow SSSS \xrightarrow{2} ()SSS \xrightarrow{2} ()()SS \xrightarrow{2} ()()()S \xrightarrow{2} ()()()()$, where $\xrightarrow{2}$ is used to denote that we have contracted the two step replacing a variable S with the terminal symbols $()$. Observe that though the trees are similar and utilises each of the three production rules the same number of times, they are still distinctly different.

Consider the simple stochastic context-free grammar

$$\begin{aligned} S &\rightarrow \underset{1}{TT} \\ T &\rightarrow \underset{1}{a} \end{aligned}$$

This grammar can only generate the sequence aa. At first hand it may appear that there are two derivations generating this sequence, namely $S \Rightarrow TT \Rightarrow aT \Rightarrow aa$ and $S \Rightarrow TT \Rightarrow Ta \Rightarrow aa$. But each of these derivations have probability 1, resulting in the sequence aa being derived with probability 2! Clearly this is not right.

The new problem, compared to regular grammars, is that we have an apparent choice of which variable first to replace. However, the combinatorial factors originating from these choices should be ignored. When determining the probability of deriving a sequence we should only sum over the set of *distinctly different* derivations. Two derivations are distinctly different if they have different derivation trees, a property illustrated in Fig. 9. We could equivalently have defined distinctly different in terms of being different *leftmost derivations*, i.e. derivations where it is always the leftmost variable in the current sequence that is replaced by applying one of its production rules. In the simple grammar above, only the first derivation of the sequence aa is leftmost, and consequently this should be the only one summed over to obtain the correct probability of 1 of the grammar generating aa.

If there is a sequence with more than one distinctly different derivation in a grammar, the grammar is said to be ambiguous. This is for example the case for the parenthesis grammar as evidenced by the three distinctly different derivations of the sequence $()()()$ shown in Fig. 9. In bioinformatics it is usually desirable to have ambiguous grammars, as different derivations of a sequence would usually correspond to different interpretations of it, for example in terms of different RNA secondary structures or alignments as discussed in Sec. 2.2.2. However, if the ambiguity extends to interpretations, such that there is a sequence with an interpretation that can be obtained by two or more distinctly different derivations, it introduces the problem discussed for class HMMs in Sec. To find the most probable interpretation we need to maximise the sum over all derivations with the same interpretation – or class annotation in the case of class HMMs – a problem that was **NP** hard even for HMMs, i.e. stochastic regular grammars. An investigation into the drawbacks of using grammars with ambiguity in interpretation in the context of RNA secondary structure was presented in [8]. Even determining whether a CFG is ambiguous is a hard problem, actually undecidable, but can in many cases – like the RNA grammars investigated in [8] – be successfully automated [3].

2.2.1 SCFG Algorithms

Just as regular grammars are equivalent to HMMs, context-free grammars have do have an automaton equivalent, called push-down automaton. However, we only mention these in passing as the algorithms for analysing languages with a context-free structure a more easily formulated using a grammar specification. Though not quite as efficient as the corresponding algorithms for HMMs, they still allow computation of the most probable parse tree for a sequence and the total probability of deriving a sequence in time polynomial in the sequence length.

Chomsky Normal Form The definition of context-free grammars allow arbitrary right hand sides. Just as there were several possible ways we could restrict regular grammars without changing expressibility (left or right, extended or not), there are several ways the right hand side of the productions of a grammar can be restricted without losing expressibility. These are known as normal forms, and requiring grammars to be on normal form simplifies algorithms as only the restricted set of production types need to be considered. We will use the Chomsky normal form (CNF) restriction where all productions are restricted to be one of the following three types:

- The start variable is replaced with the empty sequence, $S \rightarrow \epsilon$
- A variable is replaced with a single terminal symbol, $U \rightarrow \sigma$ where $\sigma \in \Sigma$
- A variable is replaced with two variables where neither is the special starting variable, $U \rightarrow VW$ where $V, W \in V \setminus \{S\}$

Any context-free grammar can be converted to a CNF grammar. For right hand sides with a sequence of more than two symbols we can break the sequence into the constituent symbols by introducing auxiliary variables, just as we saw it for converting extended regular grammars to regular grammars. If a symbol in a right-hand side of length two is a terminal symbol we can replace it with an auxiliary variable that can only be replaced with this terminal symbol, and if it is the special start variable S we can replace it with an auxiliary variable with exactly the same production rules as S . However, when the right hand side is shorter than two symbols but the production rule is not in concordance with the CNF restrictions, we need to proceed with a bit more caution.

An empty sequence right hand side, or *null production*, is only allowed for the special start variable S . To eliminate all other null productions we first identify all *nullable* variables, i.e. variables that can in one or more steps be replaced with the empty sequence. A variable U is nullable iff

- $U \rightarrow \epsilon \in P$
- $U \rightarrow V_1 \dots V_k$ and V_1, \dots, V_k are all nullable

Based on this recursive rule we can formulate Algorithm 1 to efficiently determine the set of nullable variables. Once these have been identified, for each nullable variable with the exception of S we

- remove the null production for this variable
- for every production where it occurs on the right hand side we make two copies of this production: one where the variable still occurs and one where it has been removed from the right hand side sequence

When the right hand side consists of just a single variable – remember that right hand sides with just a single symbol are required to be a terminal symbol for grammars in CNF – the transformation is even less complicated. All we need to do is copy all the productions of the right hand side variable to the left hand side variable

Algorithm 1 Nullable variables

```
N =  $\emptyset$ 
repeat
  for  $U \rightarrow x \in P$  do
    if  $x$  does not contain terminal symbols or variables not in N then
      Add  $U$  to N
until no further variables are added to N in this iteration
```

Algorithm 2 Eliminating replacement productions

```
repeat
  for  $U \rightarrow V \in P$  do
    for  $V \rightarrow x \in P$  do
      Add  $U \rightarrow x$  to P
until no updates where required in this iteration
Eliminate all  $U \rightarrow V$  productions from P
```

This all appear simple enough, so why the warning to proceed with caution? As long as only the language of a grammar, i.e. the set of finite sequences it can generate, is of concern, the last two types of transformations do not require special attention. However, in bioinformatics mostly parameterised, in particular stochastic, context-free grammars are used.

Manipulation of probabilities can easily be included in the first set of transformations mentioned: breaking a long right hand side sequence into pairs, replacing a terminal symbol in a pair with a variable, and adding a copy of S for right hand side use. The only

part not strictly trivial would be tying of the probabilities between the productions of S and the copy of S when inferring probabilities from data.

The manipulations eliminating null productions and replacement productions, on the other hand, requires more complicated manipulation of probabilities. For an already parameterised SCFG the new probabilities, after elimination of replacement productions, can be described by equations linear in (algebraic) variables describing the probability of eventually replacing variable U with variable V by a finite series of replacement production steps. This, in turn, can be solved to find the probabilities of the transformed grammar. Inferring probabilities from data, however, can become a difficult if not impossible task, if parameters are required to reflect the structure of the original grammar (as will usually be the case). Under maximum likelihood estimation we would need to maximise an equation system similar to the one for parameterised grammars, but with the parameters of the original grammar occurring polynomially, rather than linearly, as variables.

Elimination of null productions is potentially even more complicated when the grammar is stochastic. Even when the grammar is already parameterised, determining probabilities of the transformed grammar may not be straightforward. If the grammar has a production $U \rightarrow VW$ where both V and W are nullable, the probability that U is eventually replaced by the empty sequence depends on the product of the probabilities that V and W are eventually replaced by the empty sequence. If there are no cycles in these dependencies, the nullability probability can easily be determined for every variable. However, with cyclic dependencies the nullability probabilities may have to be specified by a quadratic equation system, which in general can be hard to solve [9].

Because of these complications, when designing a SCFG the best advice is, if at all possible, to avoid introducing replacement productions with cyclic structure and in general null productions. When presenting the algorithms for SCFGs we will assume CNF grammars but briefly discuss the complications arising when the grammar also contains replacement and null productions.

Cocke-Younger-Kasami Algorithm As for HMMs, the two main problems for SCFGs is to determine the most likely derivation of a sequence, and to compute the total probability of deriving a particular sequence. We first consider the problem of determining the most likely derivation of a sequence. The SCFG equivalent of the Viterbi algorithm for HMMs is the *Cocke-Younger-Kasami (CYK)* algorithm [6, 11, 13].

Assume that we are given a SCFG G in CNF and a sequence s , and want to determine the most probable derivation of s in G . Once again, derivation trees turns out to be very helpful for understanding the behaviour of SCFGs. Depending on its length, any derivation tree for s must have one of the three forms shown in Fig. 10. Moreover, the two subtrees rooted at V and W in this figure will again be of the form of one of the two

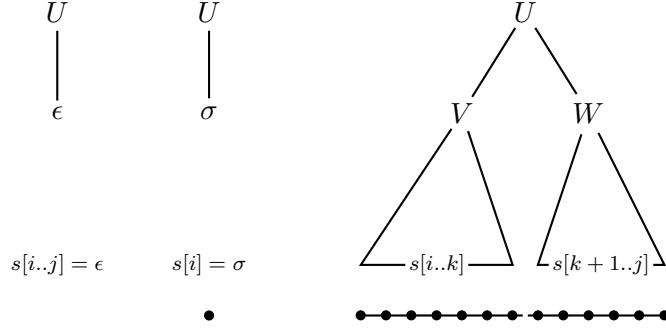


Figure 10: The three possible forms of derivation trees when deriving the subsequence from position i to position j in s from variable U in a CNF grammar. Either it is directly generated by an $S \rightarrow \epsilon$ or $U \rightarrow \sigma$ production, or the first step is an $U \rightarrow VW$ production followed by an initial part of the subsequence being generated from V and the remaining part of the subsequence being generated from W .

right hand trees (as S cannot occur on the right hand side of productions and only S can have null productions in CNF grammars).

This immediately suggests a recursion for determining the probability of the most probable derivation for s . If s is of length zero or one, we can just look up the probability of the corresponding null or terminal symbol production for S . Otherwise, we maximise over deriving a prefix, respectively the corresponding suffix, from the two variables of a production for S . If we let $C(U, i, j)$ denote the maximum probability of deriving $s[i..j]$ from U , then

$$C(U, i, j) = \begin{cases} Pr(S \rightarrow \epsilon) & \text{if } U = S \text{ and } j = i - 1 \\ Pr(U \rightarrow s[i]) & \text{if } i = j \\ \max_{\substack{U \rightarrow VW \in P \\ i \leq k < j}} Pr(U \rightarrow VW)C(V, i, k)C(W, k + 1, j) & \text{if } i < j \\ 0 & \text{otherwise} \end{cases} \quad (9)$$

where Pr describes the probability distributions over productions in P and $Pr(p) = 0$ for $p \notin P$. The desired probability is the value of $C(S, 1, |s|)$, and this value can be traced back to find the maximum probability derivations of s .

If G is in CNF, we can find $C(S, 1, |s|)$ in time $O(|P||s|^3)$ and space $O(|V||s|^2)$, e.g. as outlined in Algorithm 3 where we compute the $C(U, i, j)$ values in order of subsequences of s of increasing length. As only non-empty sequences can be derived from V and W for any production $U \rightarrow VW$, to compute a value for a subsequence of length l we only need values relating to subsequences of length strictly shorter than l .

Algorithm 3 CYK algorithm

if $s = \epsilon$ **then**
 $C(S, 1, |s|) = Pr(S \rightarrow \epsilon)$
else
 for $i = 1$ **to** $|s|$ **do**
 for $U \in V$ **do**
 $C(U, i, i) = Pr(U \rightarrow s[i])$
 for $l = 1$ **to** $|s| - 1$ **do**
 for $i = 1$ **to** $|s| - l$ **do**
 for $U \in V$ **do**
 $C(U, i, i+l) = \max_{\substack{U \rightarrow VW \in P \\ 0 \leq j < l-1}} Pr(U \rightarrow VW)C(V, i, i+j)C(W, i+j+1, i+l)$

Algorithm 4 Probability of most likely null derivations

for $U \in V$ **do**
 Set initial probability $N(U) = Pr(U \rightarrow \epsilon)$
 Initialise set of variables with known probability $F = \emptyset$
 while $F \neq V$ **do**
 $U = \arg \max_{U \in V \setminus F} \{N(U)\}$
 Add U to F
 for $V \rightarrow UW \in P$ where $W \in F$ **do**
 $N(V) = \max\{N(V), Pr(V \rightarrow UW)N(U)N(W)\}$
 for $V \rightarrow WU \in P$ where $W \in F$ **do**
 $N(V) = \max\{N(V), Pr(V \rightarrow WU)N(W)N(U)\}$

If replacement and null productions were not eliminated so the grammar is not quite in CNF, for example due to the considerations mentioned in the discussion of converting a grammar to CNF, this is where complications arise. If G contains replacement rules allowing the two derivations $U \xrightarrow{*} V$ and $V \xrightarrow{*} U$, then $C(U, i, j)$ and $C(V, i, j)$ will be mutually dependent for all $1 \leq i \leq j \leq |s|$. Also null productions can introduce cyclic dependencies, as if $V \rightarrow \epsilon$ then any rule $U \rightarrow VW$ or $U \rightarrow WV$ allows the two step derivation replacement $U \xrightarrow{2} W$. Applying the relaxation technique discussed for cycles of silent states in Sec. 1.2.1 we can still determine the probability of the most likely derivation, although the time complexity may increase by an extra factor of $O(\log |V|)$. The most complicated part is computing the probability of the most likely way to replace a variable with the empty sequence, when null productions have not been eliminated. Using a similar application of the relaxation technique as described in [10], these probabilities can still be computed efficiently by Algorithm 4.

Inside and Outside Algorithms As for HMMs, the difference between computing the most likely derivation of s in G and the total probability of deriving s in G simply consists of replacing maximisation with summation in (9) – rather than taking the maximum probability choice we sum over all choices. The resulting recursion,

$$I(U, i, j) = \begin{cases} Pr(S \rightarrow \epsilon) & \text{if } U = S \text{ and } j = i - 1 \\ Pr(U \rightarrow s[i]) & \text{if } i = j \\ \sum_{\substack{U \rightarrow VW \in P \\ i \leq k < j}} Pr(U \rightarrow VW) I(V, i, k) I(W, k + 1, j) & \text{if } i < j \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

forms the basis of the *inside* algorithm for computing the total probability of deriving s . The algorithm is identical to Algorithm 3, except that maximisation is replaced with summation. Correspondingly, the time and space complexities remain the same, $O(|P| |s|^3)$ and $O(|V| |s|^2)$ respectively.

However, for the inside algorithm we cannot use the same relaxation techniques as for the CYK algorithm when replacement and null productions have not been eliminated. If G contains replacement productions but no null productions except possibly for S , the recursions for the relevant entities for any s will be amenable to solution by linear equation system techniques. This is also mostly the case if G contains other null productions than $S\epsilon$, except for the fundamental part of computing the total probability of deriving the empty string from each variable. As can be observed from the presence of products of $N(U)$ and $N(W)$ in Algorithm 4, the recursions in this case lead to a system of quadratic equations of interdependent entities.

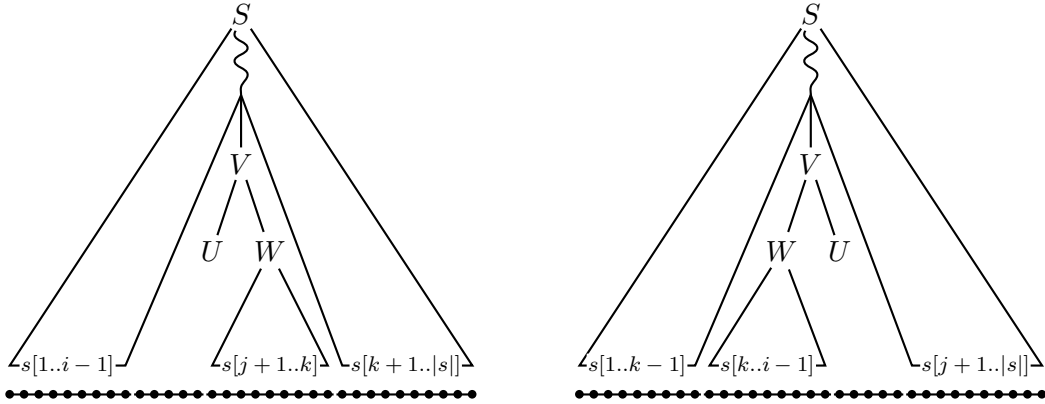


Figure 11: The intuition behind the outside algorithm is to progressively fill out the subsequence not yet derived.

The probabilities of null derivations are of course independent of s . So for a fixed grammar they could be computed once and for all, even if computationally very expensive. The values can then be looked up whenever a particular sequence is analysed. However, just as the forward algorithm is frequently used for training HMMs by the Baum-Welch expectation-maximisation procedure, the inside algorithm forms part of the expectation-maximisation procedure used for training SCFGs. In this case the parameters will be changing for each expectation-maximisation iteration. Hence, the probability of null derivations will have to be recomputed for each iteration. Once again the best advice is to design G such that this complication is avoided.

We have already mentioned that the inside algorithm forms part of the expectation-maximisation procedure for SCFG training from unannotated data, corresponding to the role the forward algorithm takes for HMMs. The part corresponding to the backward algorithm for SCFGs is called the outside algorithm. This computes the probability of deriving s , *except* for the subsequence between positions i and j which is left to be derived from variable U – i.e. the total probability of $S \xrightarrow{*} s[1..i-1]Us[j+1..|s|]$ – according to the following recursions:

$$O(U, i, j) = \begin{cases} 1 & \text{if } U = S, i = 1, \text{ and } j = |s| \\ \sum_{\substack{V \rightarrow UW \in P \\ k > j}} Pr(V \rightarrow UW)O(V, i, k)I(W, j + 1, k) & \text{if } i \leq j \\ + \sum_{\substack{V \rightarrow WU \in P \\ k < i}} Pr(V \rightarrow WU)O(V, k, j)I(W, k, i - 1) & \\ 0 & \text{otherwise} \end{cases} \quad (11)$$

The intuition behind this recursion is illustrated in Fig. 11. The recursion can be solved by an algorithm similar to the inside algorithm, this time going from longer to shorter subsequences that are still to be filled in, with the same time and space complexities.

Once we have computed inside and outside values, $I(U, i, j)O(U, i, j)$ gives the expectation of $s[i..j]$ being derived from an occurrence of U and $O(U, i, j) \sum_{k=i}^{j-1} Pr(U \rightarrow VW)I(V, i, k)I(W, k+1, j)$ the expectation of this happening by an initial application of the $U \rightarrow VW$ production. Summing over all subsequences of s we can find the expected number of uses of $U \rightarrow VW$. Similarly we can find the expected number of uses of terminal production rules $U \rightarrow \sigma$ from $O(U, i, i)Pr(U \rightarrow \sigma)$ for i with $s[i] = \sigma$. Based on these expectations the parameters can then be updated in the maximisation step.

2.2.2 SCFG Uses

RNA Secondary Structure Prediction

Alignment with Reversals

2.3 Other Grammar Types

Context-sensitive, unrestricted, linear, tree adjoining

References

- [1] A.-L. Barabási and Z. N. Oltvai. Network biology: Understanding the cell's functional organisation. *Nature Reviews Genetics*, 5:101–113, 2004.
- [2] L. E. Baum. An equality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. *Inequalities*, 3:1–8, 1972.
- [3] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. In *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, volume 4783 of *LNCS*. Springer-Verlag, July 2007. Extended version submitted for journal publication.
- [4] M. P. Brown, R. Hughey, A. Krogh, I. S. Mian, K. Sjölander, and D. Haussler. Using Dirichlet mixture priors to derive hidden Markov models for protein families. In L. Hunter, D. Searls, and J. Shavlik, editors, *Proceedings of the 1st International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 47–55, Menlo Park, California, U.S.A., July 1993. AAAI/MIT Press.

- [5] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.
- [6] J. Cocke and J. T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.
- [7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.
- [8] R. Dowell and S. R. Eddy. Evaluation of several lightweight stochastic context-free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 5:71, 2004.
- [9] J. Håstad, S. Phillips, and S. Safra. A well characterized approximation problem. *Information processing letters*, 47(6):301–305, 1993.
- [10] A. Jagota, R. B. Lyngsø, and C. N. S. Pedersen. Comparing an HMM and an SCFG. In *Proceedings of the 1st Workshop on Algorithms in Bioinformatics (WABI)*, pages 69–84, 2001.
- [11] T. Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Lab, 1965.
- [12] N. B. Leontis and E. Westhof. The non-Watson–Crick base pairs and their associated isostericity matrices. *Nucleic Acids Research*, 30(16):3497–3531, 2002.
- [13] D. H. Younger. Recognition and parsing of context-free languages in time n^3 . *Information and Control*, 10(2):189–208, 1967.