Advanced Algorithms

Class Notes for Monday, November 26, 2012

Bernard Moret

1 Dynamic Programming: Sequence Alignment

We saw last lecture a divide-and-conquer superstructure on top of the dynamic program for pairwise sequence alignment that enabled us to derive an algorithm that runs in linear space and quadratic time. This may appear to be a great achievement, but in fact the algorithm is slow: the grain in space came at a very significant increase in the leading coefficient for the quadratic running time. In consequence, this elaborate and theoretically optimal algorithm is not used in practice. Instead, practitioners focused on the principle that only good alignments are in fact useful to researchers—if the two sequences have nothing much in common, they cannot be aligned well, but also nothing much can be learned from one that would be of use in the other. Good pairwise global alignments tend not to deviate much from the diagonal, so the basic idea is to restrict the matrix to a banded version—i.e., to the diagonal plus the immediate k subdiagonals and the immediate k supradiagonals, for a band of width $2k+1$. We fill this band using dynamic programming as before, so the cost in both time and space is $\Theta(kN)$, where N is the length of the sequences to be aligned. The initialization is slightly different: we use only a small part of row 0 and of column 0 (the first $k+1$ elements). The optimal choice among 3 choices can sometimes include only two choices: when computing an element of the bottom subdiagonal, there is no element to the left, while, when computing an element of the top supradiagonal, there is no element above. To avoid having to test explicitly for these cases, we can add sentinel diagonals (diagonals $k+1$ and $-(k+1)$) filled with the most negative values possible. The solution returned is not guaranteed to be optimal: it is conceivable that a series of deletions, followed by a long and perfect substring match, followed by a series of insertions, would yield the optimal solution and such a solution, if the series of deletions is long enough, could take us outside the band and so not be discovered. However, in practice, with a largish constant value of k (typical values are in the 40 to 80 range), the solutions obtained are usually optimal and, when not, are within a few percent of optimal—and the computation is extremely fast. It remains an open question whether the quality of solutions obtained by this heuristic can (oerhaps under some additional conditions) be bounded as a function of k, N, and the mismatch and indel penalties.

2 Dynamic Programming: Scoring Phylogenetic Trees

Phylogenetic trees are simple inferences about the relationships between modern species (or, indeed any entities or objects subject to evolutionary forces, such as proteins, languages, painting styles, etc.) They are typically thought of as rooted binary trees, in which current species are associated with leaves and edges denote lines of descent. These trees are inferred from data about modern species, such as DNA sequence data—after alignment!—or morphological characters. The inference process must choose among a very large number of possible trees (the number of possible binary trees on n leaves is on the order of $n!$ and so exponential) and, in order to do that, it needs a way to score a tree.

Consider a tree inferred from sequence data: each leaf has a sequence associated with it, all of the same length and aligned with each other. One can imagine that the common ancestor of all these organisms also had a sequence of that same length, so one way to proceed is to infer sequences at internal nodes and then measure the changes along the tree edges—the score of a tree is then the minimum, over all possible labellings of internal nodes with sequences, of the sum of the changes over the edges. This is known as a *parsimonious* solution (related to Occam's razor) and the problem of scoring a given tree in terms of the total number of changes needed to explain the leaves is called the *small parsimony* problem. (The big parsimony problem is to find the best tree.) Another way to score would be to infer the probability that the observed data (the sequences) could have been generated by the tree—this gives rise to the *maximum likelihood* inference problem, a classic machine-learning problem. As it turns out, both scoring problems can be solved by dynamic programming—and dynamic programming on a tree is particularly simple and efficient: it consists of one postorder traversal (from leaves to root) followed by one preorder traversal (from root to leaves) and thus runs in linear time. The postorder traversal computes locally optimal solutions, one for each subtree, by composing optimal solutions for the children of a node into optimal solutions for that node; this is equivalent to our filling the matrix of entries. The preorder traversal is equivalent to our backtracing in the matrix: it selects solutions by selecting one at the root of the tree and propagating the choices from parent to child throughout the tree.

So how does this work for the small parsimony problem? We are given a rooted binary tree (each node has two children or is a leaf) and a sequence of length k at each leaf. Because these sequences are aligned, biologists make the further assumption that each position evolved independently of all others, so that we can solve our problem position by position, one at a time, with no reference to the others. Of course, this assumption is clearly false: all sorts of dependencies do exist. (For instance, we know that the "code of life" for genes works with groups of three nucleotides called "codons" that together code for an amino-acid.) However, in practice, the assumption enormously simplifies the problem and does not appear to cause problems. So we can rephrase our problem in its final version.

> *Given a finite alphabet Σ, a rooted binary tree T on n leaves, and an assignment of a character from Σ at each leaf, assign a character to each of the $n-1$ internal nodes of T so that the number of edges in T with different characters assigned to its endpoints is minimized.*

An edge with endpoints labelled by different characters represents a mutation, so that minimizing the number of such edges minimizes the total number of mutations needed to explain the observations (the data at the leaves), in accordance with the principle of parsimony.

In the postorder traversal, we begin by considering two leaves and their parent. Now, if the two leaves were given the same character, $x \in \Sigma$, then the locally optimal solution is to assign that character to the parent as well: in that manner our subtree will have zero edges with endpoints labelled by different characters and so will be optimal. (Any other character from Σ would cause our subtree to have two edges with endpoints labelled by different characters.) On the other hand, if one leaf was labelled x and the other y, then we can choose either x or y for the parent, since either choice will cause one edge to have differently labelled endpoints—while all other characters from Σ would cause two edges to be in that situation. In this second case, then, the locally optimal solution is not unique, so we report both. In general, we will consider that each node is to be assigned a subset of characters, any of which can be chosen to define a locally optimal solution for the subtree. Initially, at the leaves, these subsets all have exactly one element each. As we carry out the postorder traversal, we compute the subset for the parent from the already computed subsets for its two children as follows (denoting the character subset by CS):

$$CS(parent) = \begin{cases} CS(lchild) \cap CS(rchild) & \text{if} \quad CS(lchild) \cap CS(rchild) \neq \emptyset \\ CS(lchild) \cup CS(rchild) & \text{otherwise} \end{cases}$$

In the first case, the choice is locally optimal because it is costless: we found character choices that can match the character choices for the two children. In the second case, no such choice exists, so any choice that matches one of the two children is locally optimal. At the end of the postorder traversal, we have thus stored locally optimal choices at each node; we have not, however, determined a consistent, globally optimal assignment of characters to the internal nodes of the tree. Thus we now move to the preorder traversal. Any of the choices (if there is more than one) at the root is globally optimal (because it is locally optimal, but the locale is now the entire tree), so we pick one of the choices arbitrarily. In general, the preorder has assigned a single character choice to the parent and must now assign a single character choice to a child, which it does as follows, denoting the character choice as cc:

$$cc(child) = \begin{cases} cc(parent) & \text{if} \quad cc(parent) \in CS(child) \\ \text{any element of } CS(child) & \text{otherwise} \end{cases}$$

If one of the locally optimal character choices for the child happens to be the character assigned to the parent, then we choose that character for the child, thereby keeping the cost to zero—the edge from parent to child will have identically labelled endpoints. However, if the character assigned to the parent is not among those locally optimal for the child, then we have to incur a cost of one and any of the characters in the locally optimal set of choices for the child will do.

A simple induction proof verifies both properties—local optimality for the postorder traversal, global optimality for the subsequent preorder traversal. This algorithm is known as Fitch's algorithm, after Walter Fitch, a biologist who first published it in 1971 in *Systematic Zoology*—perhaps not your first choice of journal for reading about algorithms!

3 Hidden Markov Models

An introduction to Markov models would take an entire semester, so we shall quickly review salient characteristics. First, a Markov process is a special type of *stochastic process*; stochastic processes represent the evolution over time of collections of random variables. These processes may be continuous (often characterized by differential equations) or discrete (often characterized by recurrences or by automata). In Computer Science, of course, we use the discrete versions; for instance, with discrete time, stochastic processes are just time series—sequences of random variables. A Markov process has the unique property that its past history does not affect its current behavior: only its present state does. This is the *Markov property*, also called the *memoryless* property. If we further assume that states are discrete, then a Markov process can be thought of as an automaton (finite or infinite), in which each state of the process is a state of the automaton and transitions between states govern the behavior of the system according to fixed probabilities. The simplest version of such Markov systems is a *Markov chain*. Consider for instance a client-server model: clients bring jobs to the server, which completes a job, then starts working on the next. The server maintains a queue of submitted jobs. If both the client process (arrival of new jobs) and the server process (completion of the next job in the queue) have the Markov property, then we can model this system with a semi-infinite chain of states in which state i corresponds to a queue length of i, for $i = 0, 1, 2, \ldots$, and transitions from state i to state $i+1$ correspond to arrival of a new job, while transitions from state $i+1$ to state i correspond to completion of a job. Assume for simplicity that arrival is a Markov process and, furthermore, that the arrival rate does not depend on the current state; denote the arrival rate by λ; with similar assumptions, denote the completion rate by μ. The behavior of the system under steady state is then easy to analyze: in steady state, the system does not drift leftward or rightward—it is in equilibrium—so we can write

$$p_i \cdot \lambda = p_{i+1} \cdot \mu$$

(where p_i is the probability that the system is in state i), expressing that the net flow between two adjacent states in the chain is zero. We can extract $p_{i+1} = \frac{\lambda}{\mu} \cdot p_i$ and substitute to get the general solution $p_i = \rho^i p_0$, where we have written ρ for the arrival-to-completion ratio. The binding equation is simply $\sum_{i \in \mathcal{N}} p_i = 1$—the system must be in one of the states at any given time and probabilities sum to one. Substituting yields

$$p_0 \cdot \sum_{i \in \mathcal{N}} \rho^i = 1$$

which has a solution only when ρ is strictly less than 1. (Note that, with equal arrival and completion rates, the system diverges—the queue grows infinitely long. This behavior is not as odd as it may seem: the more often the queue is empty, the worse the situation gets, because an empty queue means idle time for the server, while the client is never idle. The server cannot catch up on the client, since its completion rate is the same as the client's arrival rate, so the server steadily accumulates more and more delay—a larger and larger queue.) Probably the most famous use of Markov chains today is Google's PageRank. Let

N be the total number of web pages known to Google and let page i have n_i links; Page, Brin, *et al.* (1998) proposed to set up a Markov chain whose states are the pages, with a transition from any page to any other page; page i has a transition probability of α/N to a non-linked page and of $\alpha/N + (1-\alpha)/n_i$ to a linked page, where α is an experimentally determined constant. The resulting Markov model is then analyzed to extract the probability of occupation (the "rank") of each page. The sheer size of N and the need to compute the dominant eigenvalue of an $N \times N$ matrix is what makes PageRank fascinating from a computational point of view—to say nothing, of course, of its success as a model of relevance.

Now, a *Hidden* Markov Model, or *HMM*, is a Markov Model where the current state of the system cannot be observed directly; instead, the system produces some output (e.g., by producing a value or a character at each state or during each state transition) that in some probabilistic way depends on the current state. The classic example is the so-called *occasionally dishonest casino*. A casino offers its patrons a simple game of dice: both the patron and the casino employee roll a pair of dice and the winner of the bet is the person who rolled the larger sum. This game is fair, however—the odds of winning on each side are exactly 50–50, which means that the casino cannot make money. Thus, the casino occasionally substitutes, for short periods, loaded dice for the fair dice normally used. The loaded dice tilt the odds in favor of the casino. Now, the patron does not know whether the current pair of dice is loaded or not; but the patron can observe the outcomes. The resulting model is a very simple Hidden Markov model with just two states: fair and loaded. There is a high-probability transition from the fair state back to itself, say of probability $1-\varepsilon$; and a low-probability transition from the fair state to the loaded state (probability ε). Similarly, there is a low probability of transition from the loaded state to itself and a significantly higher probability of transition from the loaded state to the fair state. In each state, the system produces two dice throws, with known (but different) probability distributions. The patron can observe the dice throws and use knowledge of the system attempt to learn something about the current state or make predictions about the statistics of the new throw. It should be noted that, in such a model, any learning is probabilistic: any sequence of dice throws that can be produced with fair dice can also be produced with loaded dice and vice versa—what distinguished the two sets of dice is the probability of observing a particular throw.

HMMs date from the 1950s, but it was in the 1960s that their mathematics were developed, mostly by L. Baum. HMMs are central to modern speech recognition and other dynamic, time-dependent recognition tasks.

Formally, an HMM for us will be a finite automaton with a set of states, S, a set of transitions $\delta\colon S \longrightarrow S$, each with a probability, an output alphabet Σ, and a so-called *emission* function, which associates with each state of S a probability distribution over Σ. (Many complex variants are possible.) In addition, we have *observations*, which are strings over Σ. Given all parameters (states, alphabet, transition function, and emission function), we can ask what is the probability that the model produces a particular output string; or we can ask the probability that the system will find itself in a particular state after outputting this string. Given a large population of samples produced by the phenomenon we are trying to model, we can attempt to infer the parameters of the model.