Advanced Algorithms

Class Notes for Thursday, November 22, 2012

Bernard Moret

1 Dynamic Programming: Sequence Alignment

Sequence alignment and its many variants have become the computational mainstay of research in biology, in part because of the advent of inexpensive, high-throughput sequencing machines: life sciences researchers today sequence anything at hand, from indiscriminate samplings of environments (gut flora, seawater, dirt, etc.) to understand microbial communities to targeted tissues to understand metabolism, aging, disease, etc. Much of what is done with sequences from a computational point of view predates the advent of DNA sequencing: computer scientists had already investigated many of the same problems in the context of strings, editors, spellcheckers, etc.

1.1 The general problem: edit distances

The most basic purpose of sequence alignment is the computation of a similarity (or distance) measure between pairs of sequences. One such measure, the Levenshtein distance, was published (in Russian) in 1965 by Russian scientist Levenshtein, and corresponds pretty much exactly to today's sequence alignment. The Levenshtein distance is an example of an *edit distance*: given two structures from some family and given a collection of allowable operations on these structures, find the shortest (or least-cost) series of operations that will transform one of the given structures into the other. The operations are editing operations, their name directly suggesting their provenance from string operations. Levenshtein defined his metric in terms of single-character editing: the allowable operations are substitutions (or one character for another) and so-called *indels*, an abbreviation for single-character insertions and deletions. (However, his paper also discusses reversals as another operation.) The alignment version of this edit distance definition is simply the layout, in an indexed array, of the two strings, establishing common indexing that "aligns" the two strings, allowing "gaps" (empty array entries) in one string where an indel is indicated, and putting into 1-1 correspondence identical characters as well as characters that are the result of a substitution (biologists would call that a "mutation"). For instance, consider the two strings ACCATT and ACATA; the edit distance is 2: we need to delete/insert a C, and substitute an A for a T. In terms of alignment, we would create the following pairwise alignment:

$$\begin{array}{cccccc} A & C & C & A & T & T \\ A & C & - & A & T & A \end{array}$$

where the dash was used to indicate a gap corresponding to an indel, where the last position shows the substitution between A and T, and where all other positions are exactly matched.

Many such edit distances have been proposed in Computer Science. The Damerau-Levenshtein distance, published in 1964 in the US by Damerau, adds one more operation, an exchange of adjacent characters– a common problem in typing and hence a reasonable edit operation. The older Hamming distance (introduced in 1950 by Hamming) removes the indels, allowing only substitutions. All of these distances can be fine-tuned to the application by giving different costs to the specific substitutions, indels, and exchanges; for instance, typing substitutions depend directly on the language and the arrangement of characters on the keyboard.

Naturally, computer scientists developped algorithms to compute these various edit distances, with or without weights on the operations. For two strings of length m and n, the Levenshtein and Damerau-Levenshtein distances can be computed in $O(mn)$ time—the full algorithm is generally credited to Lowrance and Wagner (1975); the Hamming distance, of course, is trivially computed in linear time, since it assumes that the two strings have the same length and thus does not need to align them, only to compare the character pairs at each index. In computational biology, these algorithms were re-invented and, in two slightly different variations, go by the name of Needleman-Wunsch (1970), which is exactly the Levenshtein problem, known in biology as *global pairwise sequence alignment*, and Smith-Waterman (1981), which solves a variant of the previous known as *local* pairwise sequence alignment.

All of these distances (and the corresponding alignments) are fundamentally the same problem and can be described by a simple recurrence: assume we have already computed the desired pairwise distances for all prefixes of each string up to, but not including the ith character of the first string and the jth character of the second string. Then the edit distance between the prefix of length i of the first string and the prefix of length j of the second string is given by

$$d(i,j) = \min \begin{cases} d(i,j-1) + c(\text{indel}) \\ d(i-1,j) + c(\text{indel}) \\ d(i-1,j-1) + c(\text{match/mismatch}(i,j)) \end{cases}$$

where $c(\text{indel})$ is the cost of an indel and $c(\text{match/mismatch}(i,j))$ is the cost of matching the ith character of the first string to the jth character of the second—a cost that can be positive or negative, depending on whether or not a substitution is required. This latter cost can be represented by a $k \times k$ matrix, where k is the size of the alphabet, with the diagonal showing perfect matches (no cost, or even a reward) and off-diagonal elements showings the costs of their corresponding substitutions. In computational biology, where sequences are DNA sequences or amino-acid sequences, these are 4×4 (for DNA) or 20×20 (for amino-acids) cost matrices, based on observed probabilities of substitutions. (If the probability of an event is p, then its cost can be taken as $-\log p$.)

Since this is just a recurrence, we can compute it using recursion, but we need to store intermediate results in order to run efficiently. We have already observed this need for the computation of Fibonacci numbers from their recurrence. Today, this particular aspect of dynamic programming is often called *memoization* (a truly horrible neologism), although

that term refers to a more general process implemented at compilation time of storing the results of function calls for future reuse. Since our recurrence uses two indices, a natural way to store intermediate results is to use an array; the recurrence then dictates that, prior to computing entry (i, j), we have already filled in the three array locations to the left, above, and diagonally left and above position (i, j), which inductively requires that we have computed all elements in the array the left of and above that position. The simplest way to achieve this is to fill in the array row by row, left to right (or column by column, top to bottom). It it then clear that the algorithm will run in $\Theta(mn)$ time and space, since the recurrence requires $O(1)$ time for one step.o

We have not specified the base case for our recurrence—or, equivalently, we have not shown how to initialize our array. We start the work with prefixes of length 0; the first entry we will attempt to fill using the recurrence will be the $(1,1)$ entry. Thus we need an array of $n+1$ rows and $m+1$ columns; the 0th row and 0th columns constitute our initialization, while the algorithm will fill in the array starting with row 1 and fill in each row starting with column 1. Position $(0,0)$ is the starting point of our computation; its cost is 0, since there is neither match or mismatch nor indel. Positions $(i,0)$ and $(0, j)$ for $i, j > 0$ all correspond to sequences of indels with not a single match or mismatch. Thus the cost in position $(i,0)$ is simply i times the cost of an indel; similarly, the cost in position $(0, j)$ is simply j times the cost of an indel. With this setup, our algorithm is complete and correctly computes (in array position (n,m)) the Levenshtein distance between our two strings.

It is important to note that the process is completely symmetric: we worked with prefixes of the two strings, but we could equally well have worked with suffixes and added $n+1$st row and column for initialization. We can transform one string into the other, or the other into the one. Substitution is directly reversible and deletion is the reverse of insertion. These may seem trivial observations, but they will play an important role in the development of a space-efficient algorithm below.

As in any dynamic program, what we actually computed was the optimal solution to every possible subproblem. Our subproblems here are every combination of a prefix of the first string with a prefix of the second string. Most of these subproblems are irrelevant, but had to be computed because we could not predict which would be needed. Once we have computed the optimal solution, however, we can recover which subproblems were on the critical path by backtracing our choices through the array: given that we have backtraced the algorithm to some position (i, j), we next need to know whether the optimal cost at that position was obtained from that in position $(i, j-1)$, or $(i-1, j)$, or $(i-1, j-1)$, so that we can continue the backtracing by moving to the correct choice among these three positions. We can ascertain the correct choice by comparing costs (recomputing the recurrence), but in most cases, the preferred choice is to add a "backpointer" to each array location, which that algorithm simply reads during the backtracing. The alignment itself is simply the path between array locations $(0,0)$ and (n,m) that was recovered by backtracing; this path consists of vertical, horizontal, and diagonal steps; the vertical and horizontal moves denote indels, while the diagonal moves denote matches or mismatches.

1.2 Sequence alignment in computational biology

The algorithm we just described is known in computational biology as the Needleman-Wunsch algorithm and the problem it solves is usually called global pairwise sequence alignment. Some small modifications are usually added, however, the better to reflect some biological constraints. First, as already mentioned, a cost matrix is used for matches and mismatches; this matrix is based on observed frequencies of occurrence of the matches and mismatches. This is particularly important for amino-acid sequences, as certain substitutions are nearly impossible and others quite common. (Some amino-acids are hydrophobic—the molecules orient themselves way from molecules of water—while others are hydrophilic. Since water is present just about everywhere in living tissues, it is very unlikely that a substitution could occur between a hydrophilic and a hydrophobic amino-acids. The same can be said of large amino-acid molecules vs. small ones.) Another change that we have not yet discussed is the cost of indels. If one compares two sequences of different lengths, say differing by k, then a minimum of k indels must appear in the edit sequence; biologists consider that the most likely edit sequences are those where those indels are grouped, so that a single biochemical event can delete (or insert) multiple contiguous characters. The most common so-called "gap model" (we would say "indel cost model") has two components: a "gap opening" cost, c_o, and a "gap extension" cost, c_e. The cost of a gap of length k in one sequence (k consecutive deletions) is then $c_o + (k-1) \cdot c_e$, and not just k times some fixed indel cost. In order to find an optimal solution under this model, we must adjust our recurrence slightly, since now we need to look for consecutive gaps in order to decide whether the current indel cost should be c_o or c_e. The adjustments are minor and easily made and do not change the $Theta(1)$ computational cost for computing one new array entry, so that we can accommodate this slightly more complex indel model with the same running time as before.

A more serious problem in computational biology is the storage requirements. Biologists today are often interested in aligning whole genomes. The genome of humans has roughly 3 billion characters; some plants and amoebas have genomes ten or even a hundred times larger. Nobody wants to align entire genomes as single sequences, if only because large genomes are packaged into somewhat smaller chromosomes, but also because it is relatively easy to identify corresponding regions and align these regions separately. Nevertheless, even regions of tens of millions of characters pose a challenge: an array of size $10^8 \times 10^8$ is simply not possible—we can use machines with 1TB of main memory, but we would need an exabyte of main memory, not something on the current horizon. On the other hand, using 10^{16} operations is no big deal with GHz clocks, taking only a day or so of computation. Hence we would like to reduce the memory requirements of our algorithm. If all we wanted was the final score (the value that gets stored in location (n,m) of the array, this would be very easy: we need only keep in memory the current and previous row in order to carry out our computations, so we can simply roll over the storage, using only 3 rows' worth, or $O(m+n)$ storage, at a very slight increase in time (the cost of shuffling indices to the three rows). However, biologists are not really interested in the score: they want the alignment itself. To get this alignment, we needed to backtrace through the entire

array; but if we do not store this array, how can we still recover the alignment?

Interestingly, in view of the fact that dynamic programming can be viewed as a generalization of divide-and-conquer, a divide-and-conquer approach to dynamic programming gives us a solution. Consider cutting the array in half, say by looking at row index $n/2$. (To keep notation simple, we will ignore issues of noninteger division, as we usually do in divide-and-conquer algorithms.) The alignment, which we noted is just a path in the array from $(0,0)$ to (n,m), crosses this row as some column index j. If we knew this index ahead of time, we could run a pure divide-and-conquer algorithm by separately aligning a prefix of length $n/2$ of the first sequence with a prefix of length j of the second, and then aligning a suffix of length $n/2-1$ of the first sequence with a suffix of length $m-j-1$ of the second sequence, and concatenating the results. But of course we used dynamic programming rather than divide-and-conquer precisely because we could not determine the value of this j ahead of time. So we are going to do both: use dynamic programming to determine the value of j (where the alignment path crosses row $n/2$), which gains us one point, $(n/2, j)$, through which the alignment path passes, then use the recursive divide-and-conquer framework to determine further points on the alignment path, some between $(0,0)$ and $(n/2, j)$ and some between $(n/2, j)$ and $9n,m)$. Thus our entire problem reduces to the determination of this j.

Since the only algorithm we can run in $O(mn)$ time and $O(m+n)$ space (for now) is the one that keeps track of just the current and last few rows, call it algorithm $\mathcal{A}$, we are going to leverage that algorithm to identify j. We begin by running $\mathcal{A}$ over the entire array; as we do so, we store a copy of row $n/2$, call it $R1$, and then retain the value of the optimal solution for the entire array, call it OPT. The values in $R1$ reflect alignment paths the move from $(0,0)$ to end at row $n/2$; as such they do not tell us what lies on the optimal alignment path for the entire matrix. However, they do tell us about the first half of the optimal path, so now we need to compute similar information about the second half. This is where we put to use our observation about symmetry: we now apply algorithm $\mathcal{A}$ from $(m+1,n+1)$ to row $n/2$, running up and left, filling successive rows of the array from right to left, to produce a new, different version of row $n/2$, call it $R2$. Entries in $R2$ tell us about the optimal alignment paths from $(m+1,n+1)$ to row $n/2$, or, equivalently, the optimal alignment from any position $(n/2, j)$ to the bottom right-hand position in the whole array. Therefore, at location j, the sum of the cost stored in $R1$ and the cost stored in $R2$ must equal OPT, the cost of the optimal alignment for the entire problem. Thus it is a simple matter to test each location j, $1 \leq j \leq m$, and return the first location j such that $R1[j]+R2[j] = OPT$, which takes $O(m)$ time, and is thus absorbed by the time taken to compute OPT, $R1$, and $R2$. Hence finding where the optimal alignment path intersects row $n/2$ can be done in $\Theta(mn)$ time. Now we can write a recurrence relation for the entire process:

$$t(m,n) = t(j,n/2)+t(m-j,n/2)+\Theta(mn)$$

This is a rather complex recurrence to solve, but we can get a quick idea of its solution by unrolling the first few terms as follows:

$$t(m,n) = \Theta(mn)+\Theta(jn/2)+\Theta((m-j)n/2)+\Theta(kn/4)+\Theta((j-k)n/4)+\Theta(ln/4)+\Theta((m-j-l)n/4)+\ldots$$

Every term is proportional to mn, but they diminish rapidly even as they multiply. If we assume $m = n$, we can consider the two extreme cases: where the division along a row is always even and where it is always completely unbalanced. In the first case, the sum becomes

$$t(n) = \Theta(n^2) + \Theta(n^2/4) + \Theta((n^2/4) + \Theta(n^2/16) + \Theta(n^2/16) + \Theta(n^/16) + \Theta(n^2/16) + \dots$$

and we can see that the solution is $\Theta(n^2)$. In the second case, the one column never matters, so we get the sum

$$t(n) = \Theta(n^2/2) + \Theta(n^2/4) + \Theta(n^2/8) + \dots$$

and again the solution is $\Theta(n^2)$. Indeed, the solution of the original recurrence is simply $\Theta(mn)$—that is, we managed to reduce the storage from quadratic down to linear at no asymptotic increase in running time, albeit at the cost of a seriously increased leading coefficient.