What is dynamic programming?

Sean R Eddy

Sequence alignment methods often use something called a 'dynamic programming' algorithm. What is dynamic programming and how does it work?

Dynamic programming algorithms are a good place to start understanding what's really going on inside computational biology software. The heart of many well-known programs is a dynamic programming algorithm, or a fast approximation of one, including sequence database search programs like BLAST and FASTA, multiple sequence alignment programs like CLUSTALW, profile search programs like HMMER, gene finding programs like GENSCAN and even RNA-folding programs like MFOLD and phylogenetic inference programs like PHYLIP.

Don't expect much enlightenment from the etymology of the term 'dynamic programming,' though. Dynamic programming was formalized in the early 1950s by mathematician Richard Bellman, who was working at RAND Corporation on optimal decision processes. He wanted to concoct an impressive name that would shield his work from US Secretary of Defense Charles Wilson, a man known to be hostile to mathematics research. His work involved time series and planning—thus 'dynamic' and 'programming' (note, nothing particularly to do with computer programming). Bellman especially liked 'dynamic' because "it's impossible to use the word dynamic in a pejorative sense"; he figured dynamic programming was "something not even a Congressman could object to"[1].

The best way to understand how dynamic programming works is to see an example. Conveniently, optimal sequence alignment provides an example that is both simple and biologically relevant.

Sean R. Eddy is at Howard Hughes Medical Institute & Department of Genetics, Washington University School of Medicine, 4444 Forest Park Blvd., Box 8510, Saint Louis, Missouri 63108, USA.
e-mail: eddy@genetics.wustl.edu

Dynamic programming matrix:

Optimum alignment scores 11:

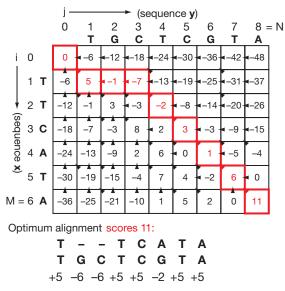| T	–	–	T	C	A	T	A
T	G	C	T	C	G	T	A
+5	–6	–6	+5	+5	–2	+5	+5

Figure 1 The filled dynamic programming matrix for two DNA sequences, x = TTCATA and y = TGCTCGTA, for a scoring system of +5 for a match, –2 for a mismatch and –6 for each insertion or deletion. The cells in the optimum path are shown in red. Arrowheads are 'traceback pointers,' indicating which of the three cases were optimal for reaching each cell. (Some cells can be reached by two or three different optimal paths of equal score: whenever two or more cases are equally optimal, dynamic programming implementations usually choose one case arbitrarily. In this example, though, the optimal path is unique.)

The biological problem: pairwise sequence alignment

We have two DNA or protein sequences, and we want to infer if they are homologous or not. To do this, we will calculate a score that reflects how similar the two sequences are (that is, how likely they are to be derived from a common ancestor). Because sequences differ not just by substitution, but also by insertion and deletion, we want to optimally align the two sequences to maximize their similarity.

Why do we need a fancy algorithm? Can't we just score all possible alignments and pick the best one? This isn't practical, because there are about $2^{2N}/\sqrt{2\pi N}$ different alignments for two sequences of length N; for two sequences of length 300, that's about 10^{179} different alignments.

Let's set up the problem with some notation. Call the two sequences x and y. They are of length M and N residues, respectively. The ith residue in x is x_i; the jth residue of y is y_j. We

need some parameters for how to score alignments: we'll use a scoring matrix $\sigma(a, b)$ for aligning two residues a,b to each other (e.g., a 4×4 matrix for scoring any pair of aligned DNA nucleotides, or simply a match and a mismatch score), and a gap penalty γ for every time we introduce a gap character.

A dynamic programming algorithm consists of four parts: a recursive definition of the optimal score; a dynamic programming matrix for remembering optimal scores of subproblems; a bottom-up approach of filling the matrix by solving the smallest subproblems first; and a traceback of the matrix to recover the structure of the optimal solution that gave the optimal score. For pairwise alignment, those steps are the following:

Recursive definition of the optimal alignment score

There are only three ways the alignment can possibly end: (i) residues x_M and y_N are

aligned to each other; (ii) residue x_M is aligned to a gap character, and y_N appeared somewhere earlier in the alignment; or (iii) residue y_N is aligned to a gap character and x_M appeared earlier in the alignment. The optimal alignment will be the highest scoring of these three cases.

Crucially, our scoring system allows us to define the score of these three cases recursively, in terms of optimal alignments of the preceding subsequences. Let $S(i, j)$ be the score of the optimum alignment of sequence prefix $x_1 \ldots x_i$ to prefix $y_1 \ldots y_j$. The score for case (i) above is the score $s(x_M, y_N)$ for aligning x_M to y_N plus the score $S(M-1, N-1)$ for an optimal alignment of everything else up to this point. Case (ii) is the gap penalty γ plus the score $S(M-1, N)$; case (iii) is the gap penalty γ plus the score $S(M, N-1)$.

This works because the problem breaks into independently optimizable pieces, as the scoring system is strictly local to one aligned column at a time. That is, for instance, the optimal alignment of $x_1 \ldots x_{M-1}$ to $y_1 \ldots y_{N-1}$ is unaffected by adding on the aligned residue pair x_M, y_N, and likewise, the score $s(x_M, y_N)$ we add on is independent of the previous optimal alignment.

So, to calculate the score of the three cases, we will need to know three more alignment scores for three smaller problems:

$$S(M-1, N-1), \quad S(M-1, N), \quad S(M, N-1).$$

And to calculate those, we need the solutions for nine still smaller problems:

$$
\begin{array}{lll}
S(M-2, N-2), & S(M-2, N-1), & S(M-1, N-2), \\
S(M-2, N-1), & S(M-2, N), & S(M-1, N-1), \\
S(M-1, N-2), & S(M-1, N-1), & S(M, N-2).
\end{array}
$$

and so on, until we reach tiny alignment subproblems with obvious solutions (the score $S(0,0)$ for aligning nothing to nothing is zero).

Thus, we can write a general recursive definition of all our partial optimal alignment scores $S(i, j)$:

$$S(i,j) = \max \begin{cases} S(i-1, j-1) + \sigma(x_i, y_j), \\ S(i-1, j) + \gamma, \\ S(i, j-1) + \gamma. \end{cases}$$

The dynamic programming matrix

The problem with a purely recursive alignment algorithm may already be obvious, if you looked carefully at that list of nine smaller subproblems we'd be solving in the second round of the top-down recursion. Some subproblems are already occurring more than once, and this wastage gets exponentially worse as we move deeper into the recursion. Clearly, the sensible thing to do is to somehow keep track of which subproblems we are already working on. This is the key difference between dynamic programming and simple recursion: a dynamic programming algorithm memorizes the solutions of optimal subproblems in an organized, tabular form (a dynamic programming matrix), so that each subproblem is solved just once.

For the pairwise sequence alignment algorithm, the optimal scores $S(i, j)$ are tabulated in a two-dimensional matrix, with i running from 0...M and j running from 0...N, as shown in **Figure 1**. As we calculate solutions to subproblems $S(i, j)$, their optimal alignment scores are stored in the appropriate (i, j) cell of the matrix.

A bottom-up calculation to get the optimal score

Once the dynamic matrix programming matrix $S(i, j)$ is laid out—either on a napkin or in your computer's memory—it is easy to fill it in in a 'bottom-up' way, from smallest problems to progressively bigger problems. We know the boundary conditions in the leftmost column and topmost row ($S(0, 0) = 0$; $S(i, 0) = \gamma i$; $S(0, j) = \gamma j$): for example, the optimum alignment of the first i residues of sequence x to nothing in sequence y has only one possible solution, which is to align to gap characters and pay i gap penalties. Once we've initialized the top row and left column, we can fill in the rest of the matrix by using the recursive definition of $S(i, j)$ to calculate any cell where we already know the values we need for the three adjoining cells to the upper left $(i-1, j-1)$, above $(i-1, j)$ and to the left $(i, j-1)$. There are several different ways we can do this; one is to iterate two nested loops, $i = 1$ to M and $j = 1$ to N, so we're filling in the matrix left to right, top to bottom.

A traceback to get the optimal alignment

Once we're done filling in the matrix, the score of the optimal alignment of the complete sequences is the last score we calculate, $S(M, N)$. We still don't know the optimal alignment itself, though. This, we recover by a recursive 'traceback' of the matrix. We start in cell M,N, determine which of the three cases we used to get here (*e.g.*, by repeating the same three calculations), record that choice as part of the alignment, and then follow the appropriate path for that case back into the previous cell on the optimum path. We keep doing that, one cell in the optimal path at a time, until we reach cell (0,0), at which point the optimal alignment is fully reconstructed.

Fine. But what do I really need to know?

Dynamic programming is guaranteed to give you a mathematically optimal (highest scoring) solution. Whether that corresponds to the biologically correct alignment is a problem for your scoring system, not for the algorithm.

Similarly, the dynamic programming algorithm will happily align unrelated sequences. (The two sequences in **Figure 1** might look well-aligned; but in fact, they are unrelated, randomly generated DNA sequences!) The question of when a score is statistically significant is also a separate problem, requiring clever statistical theory.

Dynamic programming is surprisingly computationally demanding. You can see that filling in the matrix takes time proportional to *MN*. Alignment of two 200-mers will take four times as long as two 100-mers. This is why there is so much research devoted to finding good, fast approximations to dynamic programming alignment, like the venerable workhorses BLAST and FASTA, and newer programs like BLAT and FLASH.

Only certain scoring systems are amenable to dynamic programming. The scoring system has to allow the optimal solution to be broken up into independent parts, or else it can't be dealt with recursively. The reason that programs use simple alignment scoring systems is that we're striking a reasonable compromise between biological realism and efficient computation.

Note: Supplementary information is available on the Nature Biotechnology website.

1. Bellman, R.E. *Eye of the Hurricane: An Autobiography.* (World Scientific, Singapore, 1984).

Further study

To study a working example, you can download a small, bare-bones C implementation of this algorithm (see **Supplementary Note**). I used this C program to generate **Figure 1**.

nature biotechnology

Wondering how some other mathematical technique really works? Send suggestions for future primers to askthegeek@natureny.com.