18.417 Introduction to Computational Molecular Biology

Lecture 4: September 21, 2004 — Scribe: Russ Cox

Lecturer: Ross Lippert — Editor: Michael Walfish

Dynamic Programming

Introduction

The vast majority of compute cycles in biology go to string matching problems such as DNA sequence alignment, so it is important to be able to solve these problems efficiently. One important technique for doing so is dynamic programming.

Dynamic Programming: Change is a Classic DP Example . . .

Consider the problem of finding the minimum number of coins required to represent a particular sum of money. For example, using the American system of coins representing 1, 5, 10, and 25 cents, the smallest way to represent 40 cents is one 25-cent coin, one 10-cent coin, and one 5-cent coin.

. . . but not for American denominations (greedy approach works)

It turns out that for the American coin set, a greedy approach to making change yields correct solutions. Specifically, if you need to make n cents, use as many of the 25-cent coins as you can, then 10-cent coins, then 5-cent coins, and finally 1-cent coins.

However . . .

If we add a new coin to the set — say, the George Bush 20-cent piece, this greedy approach fails. For example, to make 40 cents, the greedy approach would still use one 25-cent coin, one 10-cent coin, and one 5-cent coin (as it did above), a total of three coins. But we can do better by using just two 20-cent coins. So the greedy approach cannot be used in general.

Recurrence

If we think about the problem for a while, we find that the answer can be stated in terms of answers to other instances of the problem. That is, we can write a recurrence. If $minNumCoins(m)$ denotes the minimum number of coins required to make m cents using the set of coin denominations $\{c_1, c_2, ..., c_d\}$, then:

$minNumCoins(0) = 0$

and

$$minNumCoins(M) = \min_{\text{of}} \begin{cases} minNumCoins(M-c_1) + 1 \\ minNumCoins(M-c_2) + 1 \\ \dots \\ minNumCoins(M-c_d) + 1 \end{cases}$$

In words, in order to make m cents, we have to use some coin first. After having made that choice, you want to make the rest of the change in a minimal way.

We could use this recurrence to write a recursive algorithm for computing $minNumCoins(\cdot)$:

```
RecursiveChange(M,c,d)
  if $M = 0$
    return 0
  bestNumCoins ← infinity
  for i ← 1 to d
    if $M \geq c_i$
      numCoins ← RecursiveChange($M - c_i$, c, d)
      if numCoins + 1 < bestNumCoins
        bestNumCoins ← numCoins + 1
  return bestNumCoins
```

This recursive algorithm implements a brute force exhaustive search of all ways to make m cents out of the given coins. It ends up taking approximately $O(m^d)$ time, which is pretty long.

If we look at the search tree, we find that some subproblems (for example, $minNumCoins(70)$ in the piece of search tree shown in Figure 4.1) are computed many times. To avoid this duplication of work, we can make a table of previously-computed results and then consult the table before invoking the actual computation.

Finally, we can remove the recursion completely, instead iterating through the table filling it out until we have filled in the answer for the problem we are interested in solving.

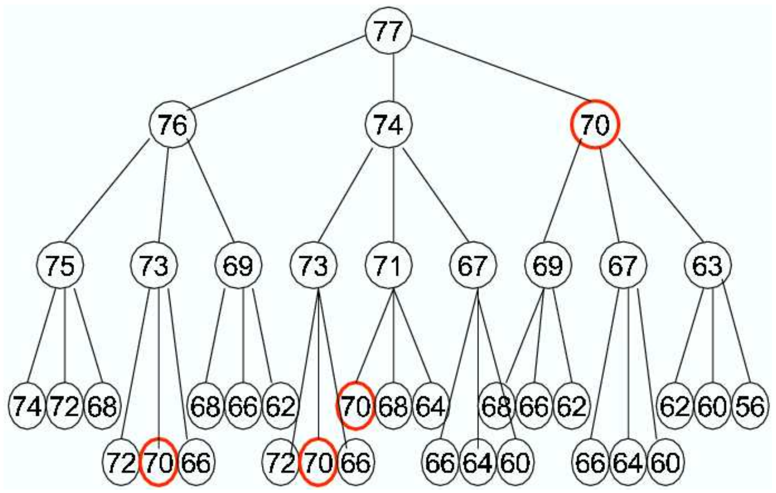


Figure 4.1: Naive search tree for making change. Subproblems are recomputed many times, which is inefficient.

There are m entries in the table and d entries that must be consulted to fill in a new entry, so the work required has been reduced from $O(m^d)$ to $O(md)$.

Dynamic Programming: Formalization

In its essence, dynamic programming is the technique of

(a) stating a computation as a recurrence and then
(b) building up a table of "previous" results to be used in computing future results.

We can formalize this "essence" by saying that the three parts of a dynamic programming solution are:

1. a space of problem 'states' A
2. a recurrence defining the solution s_i to state i in terms of the solution to other problem states s_j: $s_i = f(\{s_j\})$
3. a partial ordering $<$ on A such that $i < j$ if and only if the solution s_i is required to compute s_j.

Alignments

The problem of sequence alignment is to take two input sequences

$$\begin{aligned} s_1 &= a_1, a_2, \ldots, a_m \\ s_2 &= b_1, b_2, \ldots, b_n \end{aligned}$$

and find a way to insert gaps into them to produce new sequences s_1^*, s_2^* such that the Hamming distance (or more generally, some elementwise scoring function) between the two new sequences is optimized.

The alignment score is $\sum_i \delta(a_i^*, b_i^*)$.

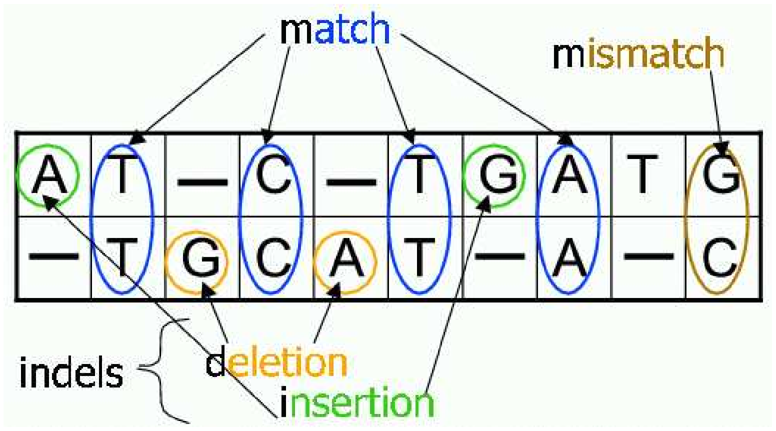


Figure 4.2: Example alignment.

An example alignment is depicted in Figure 4.2.

In computer science, the goal of this problem is to minimize the "edit distance," typically defined as a constant μ times the number of mismatches plus a constant σ times the number of inserted gaps. The dynamic programming solution was first published by Levenstein in 1966.

In biology, the goal is to maximize the "similarity," typically defined as the number of matches minus the computer science "edit distance". The dynamic programming solution was first published by Needleman and Wunsch in 1970.

The two scoring functions are essentially equivalent and do not change the algorithm beyond whether we are trying to minimize or maximize the score.

A DP Solution

As is the case for many problems that can be solved with dynamic programming, determining the problem state space and the recurrence can be hard. It may take a few tries to come up with a recurrence that is both correct and amenable to dynamic programming. Think hard.

In an instance of the alignment problem, let us say that the input sequences

$$\begin{aligned} s_1 &= a_1, a_2, \ldots, a_m \\ s_2 &= b_1, b_2, \ldots, b_n \end{aligned}$$

have gaps inserted to produce

$$\begin{aligned} s_1^* &= a_1^*, a_2^*, \ldots, a_t^* \\ s_2^* &= b_1^*, b_2^*, \ldots, b_t^* \end{aligned}$$

and that the resulting score is given by

$$score(s_1^*, s_2^*) = \sum_{i=1}^{t} \delta(a_1^*, b_1^*)$$

Note that the score can be decomposed into the score for the last symbol pair in the alignment and then the score for the rest of the strings:

$$score(s_1^*, s_2^*) = score(s_1^*/a_t^*, s_2^*/b_t^*) + \delta(a_t^*, b_t^*)$$

(Here, the notation s/a denotes a string s with the final symbol a removed.)

Assuming that it never makes sense to align two inserted gaps, there are three possibilities for the pair (a_1^*, b_1^*): they are (a_m, b_n), $(a_m, -)$, and $(-, b_n)$, where $-$ denotes a gap. Thus we can turn our recurrence for the score of the gap-inserted strings into a recurrence for the minimum score of the original strings:

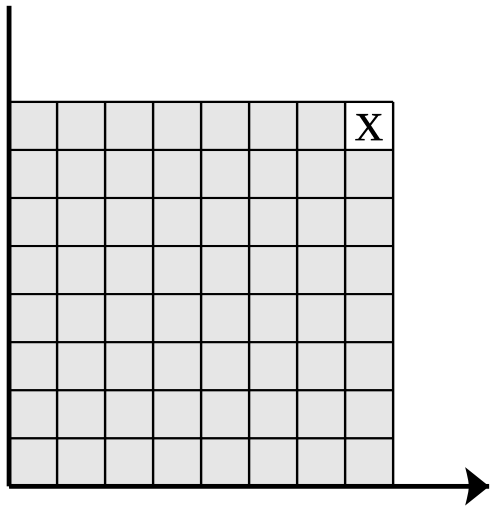
$$\begin{aligned} minscore(s_1, s_2) = \min\{ & minscore(s_1/a_m, s_2/b_n) + \delta(a_m, b_n) \\ & minscore(s_1/a_m, s_2) + \delta(a_m, -) \\ & minscore(s_1, s_2/b_n) + \delta(-, b_n)\} \end{aligned}$$

The space of problem states in this recurrence is simply prefixes of s_1 and s_2. We will denote the i-symbol prefix of s_1 and the j-symbol prefix of s_2 as (i, j).

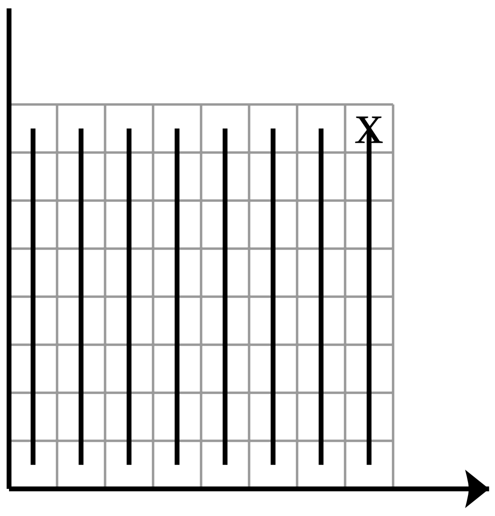
Finally, in order to compute the solution to (i, j), it is clear that the recurrence above requires the solutions to $(i-1, j-1)$, $(i-1, j)$, and $(i, j-1)$. Taking the transitive closure of this relationship, the partial order is given by:

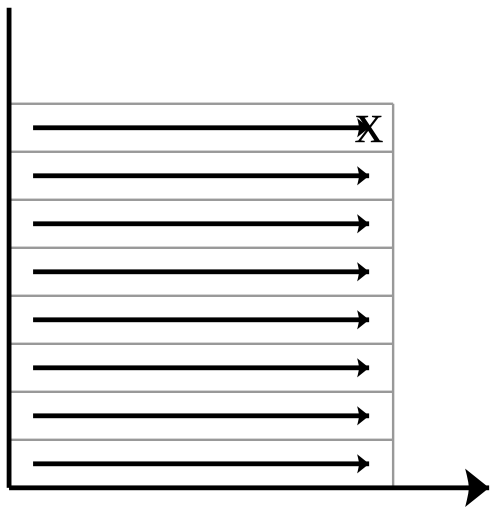
$$(i', j') < (i, j) \qquad \text{iff} \qquad i' \leq i \quad \wedge \quad j' \leq j \quad \wedge \quad (i' < i \quad \vee \quad j' < j)$$

If we consider the state place as a plane, then computing the solution for a point x requires solutions to all points in or on the rectangle with corners at x and the origin, except x itself. In the picture, computing the solution to x requires the solution to all points in the shaded area.

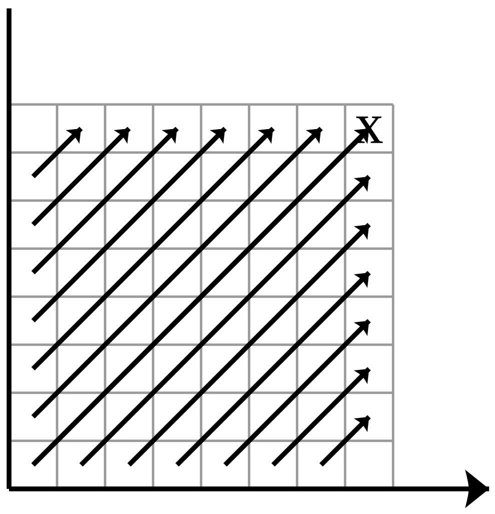


Viewing the state space as an m x n rectangle, we can compute solutions in a variety of orders, including by rows, by columns, and by diagonals:





All these orders are equally correct for solving the problem, but only the diagonal ordering is parallelizable: the solution to an entire diagonal can be split arbitrarily among a set of processors once the previous diagonal is solved.



Biological Applications

This solution to the global sequence alignment problem can be used in many biological settings, tuned by varying the scoring function δ.

In one common setting, the δ function is an energetic scoring function, which tries to model how likely it is that one sequence will bind to another. For example, the score $\delta(A,T)$ will depend on the binding energy between adenosine and thymine.

In another common setting, the δ function is based on evolutionary substitution matrices, which tries to model how likely it is that one symbol evolved into another over time. For example, the score $\delta(A,T)$ will depend on the probability that A changes into T as a sequence is copied over and over for some period of time.

In this latter setting, the symbols may be nucleotides but may also be amino acids. Determining a scoring function for amino acids is harder than determining a scoring function for nucleotides. For example, it turns out that hydrophilic amino acids are more likely to change into other hydrophilic amino acids than they are to change into hydrophobic amino acids, and vice-versa.

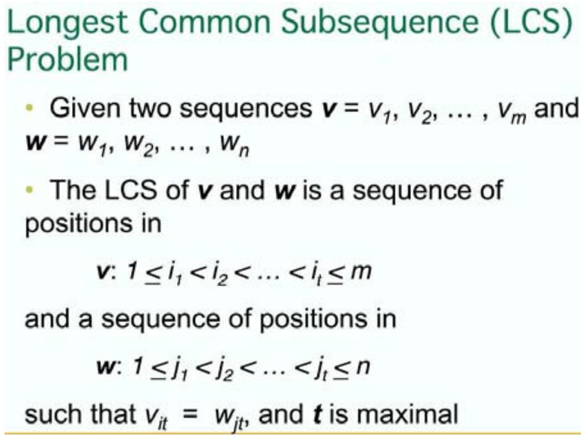
To determine a scoring function for amino acids, biologists typically consider two sequences known to be evolutionarily similar (for example, human and mouse hemoglobin). Such sequences have the same alignment for many reasonable choices of scoring function: typically each mismatch is sandwiched between long runs of perfect matching. Counts of these forced matches can be used to compute the probability that, say, arginine evolves into lysine.

The earliest reasonably justified substitution scoring matrix for amino acids is known as the PAM1 matrix. Exponentiating the PAM1 matrix approximates the scores for longer time periods than the PAM data covers.

Alternate Applications

Other more restricted problems can be framed as alignment problems too.

One such problem is the "Longest Common Subsequence" problem. which allows matches and gaps but no mismatches. This can be handled with an alignment solver by scoring an infinite penalty for a mismatch. A formal statement of the problem is:



Longest Common Subsequence (LCS) Problem

- Given two sequences $\boldsymbol{v} = v_1, v_2, \ldots, v_m$ and $\boldsymbol{w} = w_1, w_2, \ldots, w_n$
- The LCS of $\boldsymbol{v}$ and $\boldsymbol{w}$ is a sequence of positions in

$\boldsymbol{v}$: $1 \leq i_1 < i_2 < \ldots < i_t \leq m$

and a sequence of positions in

$\boldsymbol{w}$: $1 \leq j_1 < j_2 < \ldots < j_t \leq n$

such that $v_{it} = w_{jt}$, and $\boldsymbol{t}$ is maximal

Another problem is computing sparse alignments. In the sparse alignment problem, the input is two sequences and a list of "islands of similarity" between those two sequences. These "islands" are represented by triples (x, y, s), which means that a potential match exists between the two sequences (at positions x in the first sequence and y in the second) and that the match has score s. (The "islands" are very few base pairs each, which is why we represent them as points, e.g., x, and not as a (begin,end) pair.) The output is a sequence of pairs that monotonically increases in x and y and maximizes the sum of the corresponding scores minus a gap penalty. Biologically, this output represents a possible alignment of two sequences in which common points may be in very different places along the actual DNA strand.

The sparse alignment problem can be solved with a dynamic programming approach similar to the one used for the alignment problem. The number of subproblems is equal to the number of triples in the input, but computing the answer to each subproblem here potentially requires looking at all possible predecessor triples rather than just a constant number of previous results, as in the standard alignment problem.