6.047/6.878 Lecture 2 - Sequence Alignment and Dynamic Programming

September 15, 2008

1 Introduction

Evolution has preserved functional elements in the genome. Such elements often manifest themselves as *homologues*, or similar sequences in descendants of a common ancestor. The two main types of homologous sequences are *orthologous* and *paralogous* sequences. Orthologous sequences typically have similar functions in both the ancestor and the descendant and arise through speciation events, while paralagous sequences arise from common ancestors through gene duplication. Furthermore, paralagous genes imply a common ancestor but, due to mutation and evolution the functionality of that particular gene has shifted considerably. We will mostly be interested in studying orthologous gene sequences.

Aligning sequences is one of the main ways of discovering such similarities between different ancestors. And in solving *sequence alignment* problems, the primary computational tool will be *Dynamic Programming*.

1.1 An Example Alignment

Within orthologus gene sequences, there are *islands of conservation* which are relatively large stretches of nucleotides that are preserved between generations. These conserved elements typically imply functional elements and vice versa. Although, note that conservation is sometimes just random chance.

As an example, we considered the alignment of the Gal10-Gal1 intergenic region for four different yeast species, the first whole genome alignment for crosspieces (Page 1 Slide 5). As we look at this alignment, we note some areas that are more conserved than others. In particular, we note some small conserved motifs such as `CGG` and `CGC`, which in fact are functional elements in the binding of Gal4. This example illustrates how we can read evolution to find functional elements.

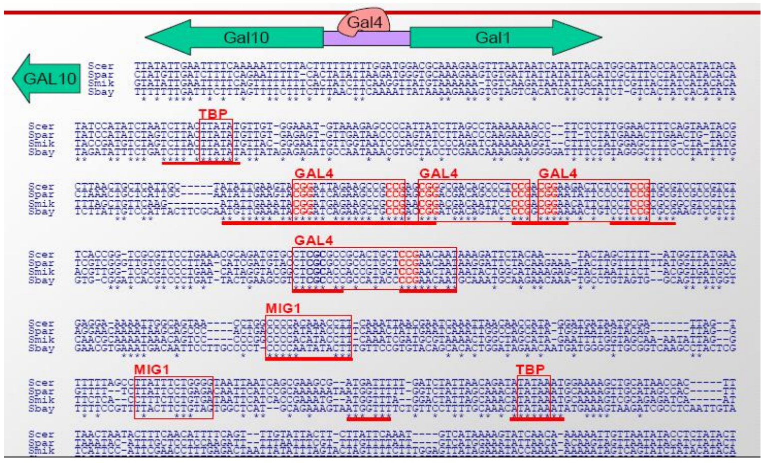


Figure 1: Sequence alignment of Gal10-Gal1.

1.2 Genome Changes are Symmetric

The genome changes over time. In studying these changes, we'll confine ourselves to the level of nucleotide mutations, such as substitutions, insertions and deletions of bases. Lacking a time machine, we cannot compare genomes of living species with their ancestors, so we're limited to just comparing the genomes of living descendants.

For our purposes, time becomes a reversible concept. This means that we can consider the events that change a genome from one species to another as occurring in reverse order (tracing up an evolutionary tree towards a common ancestor) or forward order (tracing downwards from a common ancestor). We consider all DNA changes to be symmetric in time: an insertion in one direction is equivalent to a deletion in the other.

Furthermore, since mutations and genomic changes are independent of each other, it is irrelevant to consider the "direction" of the branches in the evolutionary tree. Therefore to study the common ancestor of two different species, we need only consider the intermediate stages in a mutation process from one of the descendants to the other. In doing so, we can avoid the problem of not having the ancestral nucleotide sequence.

2 Problem Formulations: Biology to CS

We need to translate the biological problem of sequence alignment into a computer science problem that can be attacked by computational tools. In creating a model to represent the problem, we need to balance biological relevance with computational tractability.

2.1 Goal of Alignment

The goal of alignment is to infer the 'edit operations' that change a genome by looking only at the endpoints. We define the set of operations evolutionary operations to contain insertions, deletions, and mutations. And since there are many possible sequences of events that could change one genome to the other, we also need to establish an optimality criterion, e.g. minimum number of events or minimum cost.

In reality, insertions and deletions are much less likely than substitutions, and so we would prefer to attribute a different cost to each of those events to make solution sequences prefer substitutions. The algorithm we show will assume equal costs for each operation; however, it can be easily generalized for varying cost. [1] [2]

2.2 Brute-Force Solution

Given a metric to score a given alignment, the simple brute-force algorithm just enumerates all possible alignments, computes the score of each one, and picks the alignment with the maximum score. How many possible alignments are there? If you consider only NBA[3] and $n \approx m$, the number of alignments is [4]

$$\binom{n+m}{m} = \frac{(n+m)!}{n!m!} \approx \frac{(2n)!}{(n!)^2} \approx \frac{\sqrt{4\pi n)}\frac{(2n)^{2n}}{e^{2n}}}{(\sqrt{2\pi n}\frac{n^n}{e^n})^2} = \frac{2^{2n}}{\sqrt{\pi n}} \tag{1}$$

For some small values of n such as 100, the number of alignments is already too big ($\approx 10^{60}$) for this enumerating strategy to be feasible.

[1]Even different substitution events have different likelihood since specific polymerases are more or less likely to make specific mistakes.

[2]There was a note about evolutionary pressure to keep the genome compact. For example, the genome of viruses is very small since they have the evolutionary pressure of fitting it inside the small virus cell. Similarly, bacteria are pressured to keep their genome small to save energy in replication and to allow easy exchange of genomic material between cells. On the other hand, there is little pressure for human cells to maintain a small genome, so it is no surprise that the human genome is so large.

[3]Non Boring Alignments, an alignment that doesn't match gaps on both sequences

[4]We use the stirling approximation: $n! \approx \sqrt{2\pi n}\frac{n^n}{e^n}$

2.3 Formulation 1: Longest Common Substring

As a first attempt, suppose we treat the nucleotide sequences as strings over the alphabet A, C, G, and T. Given two such strings, R and S, we might try to align them by finding the longest common substring between them. In particular, these substrings cannot have gaps in them.

As an example, if R = ACGTCATCA and S = TAGTGTCA, the longest common substring between them is GTCA. So in this formulation, we could align R and S along their longest common substring, GTCA, to get the most matches.

A simple algorithm would be to just try aligning S with different offsets with respect to R and keeping track of the longest substring match found thus far. But this algorithm is quadratic (too slow) in the lengths of R and S. Assuming $|R| = |S| = n$, we loop through each character in R trying to align S at that offset. During each iteration, we then loop over the characters of S and compare them to the corresponding characters in S looking for matching substrings. Thus the total running time would be $O(n^2)$.

2.4 Formulation 2: Longest Common Subsequence

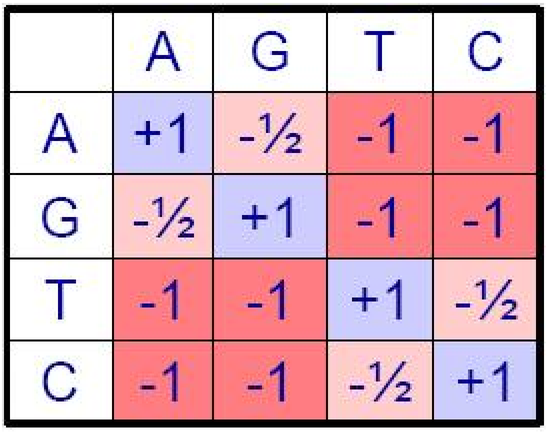
Another formulation is to allow gaps in our subsequences and not just limit ourselves to substrings with no gaps. Given a sequence $X = x_1 \ldots x_m$, we formally define $Z = z_1 \ldots z_k$ to be a *subsequence* of X if there exists a strictly increasing sequence $i_1, \ldots, i_k$ of indices of X such that for all $j = 1, \ldots, k$, we have $x_{i_j} = z_j$ (CLRS 350-1). In the longest common subsequence (LCS) problem, we're given two sequences X and Y and we want to find the maximum-length common subsequence Z.

Consider the example from the previous section with sequences R and S. The longest common subsequence is $AGTTCA$, a longer match than just the longest common substring.

2.5 Formulation 3: Sequence Alignment as Edit Distance

The previous LCS formulation is almost what we want. But so far we haven't specified any cost functions that differentiate between the different types of edit operations: insertion, deletions, and substitutions. Implicitly our cost function has been uniform implying that all operations are equally possible. But since substitutions are much more likely, we want to bias our LCS solution with a cost function that prefers substitutions over insertions and deletions.

So we recast Sequence Alignment as a special case of the classic *Edit-Distance* problem in computer science (CLRS 366). We add varying penalties for different edit operations to reflect biological occurences. Of the four nucleotide bases, A and G are purines (larger, two fused rings), while C and T are pyrimidines (smaller, one ring). Thus RNA polymerase, the enzyme the transcribes the nucleotide sequence, is much more likely to confuse two purines or two pyrimidines since they are similar in structure. Accordingly, we define the following scoring matrix:



	A	G	T	C
A	+1	-½	-1	-1
G	-½	+1	-1	-1
T	-1	-1	+1	-½
C	-1	-1	-½	+1

Figure 2: Cost matrix for matches and mismatches.

Now the problem is to the find the least expensive (as per the cost matrix) operation sequence transforming the initial nucleotide sequence to the final nucleotide sequence.

3 Dynamic Programming

Before we can tackle the problem algorithmically, we need to introduce a few computational tools.

Dynamic programming is a technique use to solve optimization problems by expressing the problem in terms of its subproblems. Its hallmarks are optimal substructure and overlapping subproblems. A problem exhibits optimal substructure "if an optimal solution to the problem contains within it optimal solutions to subproblems." (CLRS 339) Unlike in greedy algorithms, there is no locally optimal choice we can make to reach a globally optimal solution. In dynamic progamming, we need to trace back through several choices and we keep track of these choices by storing results to subproblems in a table. By filling in this table, we build up a solution to entire problem from the "bottom-up" by first solving increasing larger sub-problems.

3.1 Fibonacci Example

The Fibonacci numbers provide an instructive example of the benefits of dynamic programming. The Fibonacci sequence is recursively defined as $F_0 = F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$. We wish to develop an algorithm to compute the nth Fibonacci number.

3.2 Recursive (Top-Down) Solution

The simple top-down approach is to just apply the recursive definition. Here's a simple way to do this in Python:

```
# Assume n is non-negative
def fib(n):
  if n == 0 or n == 1: return 1
  else: return fib(n-1) + fib(n-2)
```

But this top-down algorithm runs in exponential time. If $T(n)$ is how long it takes to compute the n-th Fibonacci number, we have that $T(n) = T(n-1) + T(n-2)$, so $T(n) = O(\phi^n)$.[5] The problem is that we are repeating work by solving the same sub-problem many times.

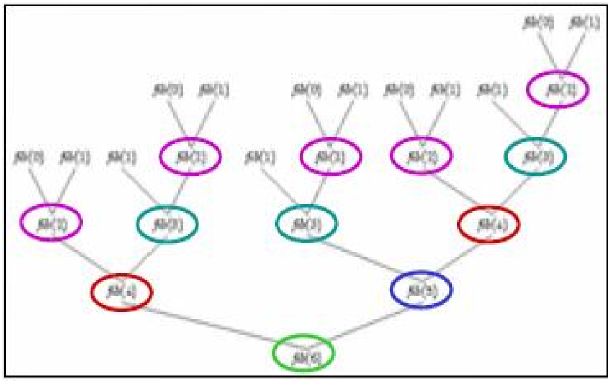


Figure 3: The recursion tree for the `fib` procedure showing repeated subproblems.

[5]Where ϕ is the golden ratio. In the slides it was claimed that $T(n) = O(2^n)$, which is incorrect, nonetheless this is still exponential.

3.3 Dynamic Programming (Bottom-Up) Solution

For calculating the nth Fibonacci number, instead of beginning with $F(n)$, and using recursion, we can start computation from the bottom since we know we are going to need all of the subproblems anyway. This way, we will save a lot of repeated work that would be done by the top-down approach and we will be able to compute the n-th Fibonacci number in $O(n)$ time. Here's a sample implementation of this approach in Python:

```
def fib(n):
  x = y = 1
  for i in range(1, n):
    z = x+y
    x = y
    y = z
  return y
```

This method is optimized to only use constant space instead of an entire table since we only need the answer to each subproblem once. But in general dynamic programming solutions, we want to store the solutions to subproblems in a table since we may need to use them multiple times and we should be able to quickly look them up in a table. Such solutions would look more like the sample implementation from lecture (tweaked to properly compile):

```
def fib(n):
  fib_table = [1, 1] + (n-1)*[0]  # initialize table with zeros
  for i in range(2,n+1):
    fib_table[i] = fib_table[i-1] + fib_table[i-2]
  return fib_table[n]
```

3.4 Memoization Solution

Yet another solution would be to use *memoization* which combines the top-down approach with the dynamic programming approach. In this approach, we still use the recursive formula to compute the sequence but we also store the results of subproblems in a table so that we don't have to resolve them. Here's a sample memoization implementation in Python:

```
def fib(n):
  fib_table = [1, 1] + (n-1)*[0]
  def get(i):
    if fib_table[i] != 0:
      return fib_table[i]
    else:
      fib_table[i] = get(i-1) + get(i-2)
      return fib_table[i]
  if n == 0 or n == 1: return 1
  else: return get(n-1) + get(n-2)
```

4 Needleman-Wunsch: Dynamic Programming Meets Sequence Alignment

We will now use dynamic programming to tackle the harder problem of general sequence alignment. Given two strings S and T, we want to find the longest common subsequence which may or may not contain gaps. Further we want the optimal such longest common subsequence as defined by our scoring function in Figure 2. We will use the notation $S = S_1 \ldots S_n$ and $T = T_1 \ldots T_m$ when talking about individual bases in the sequences. Furthermore, we'll let d represent the gap penalty cost and $s(x, y)$ represent the score of aligning an x and a y. The alogorithm we willl develop in the following sections to solve sequence alignment is known as the *Needleman-Wunsch* algorithm.

4.1 Step 1: Find a Subproblem

Suppose we have an optimal alignment for two sequences in which S_i matches T_j. The key insight is that this optimal alignment is composed of optimal alignments between $S_1 \dots S_{i-1}$ and $T_1 \dots T_{i-1}$ and between $S_{i+1} \dots S_n$ and $T_{j+1} \dots T_n$. If not, we could "cut-n'-paste" a better alignment in place of the suboptimal alignments to achieve a higher score alignment (which contradicts the supposedly optimality of our given alignment). Also note that the optimality score is additive, so we can obtain the score of the whole alignment by adding the score of the alignment of the subsequences.

4.2 Step 2: Index Space of Subproblems

We need to parameterize the space of subproblems. We identify by F_{ij} the score of the optimal alignment of $S_1 \dots S_i$ and $T_1 \dots T_j$. Having this parameterization allows us to maintain an $m \times n$ matrix F with the solutions (i.e. optimal scores) for all the subproblems.

4.3 Step 3: Make Locally Optimal Choices

We can compute the optimal solution for a subproblem by making an locally optimal choice based on the results from the smaller subproblems. Thus, we need to establish a recursive function that shows how the solutions to a given problem depends on its subproblems. And we use this recursive definition to fill up the table F in a bottom-up fashion.

We can consider the 4 possibilities (insert, delete, substitute, match) and evaluate each of them based on the results we have computed for smaller subproblems. To initialize the table, we set $F_{1j} = d \cdot j$ and $F_{i1} = d \cdot i$ since those are the scores of aligning j and i gaps respectively.

Then we traverse the matrix row by row computing the optimal score for each alignment subproblem by considering four possibilities which are summarized pictorially in Figure 4:

- Sequence S has a gap at the current alignment position.
- Sequence T has a gap at the current alignment position.
- There is a mutation (nucleotide substitution) at the current position.
- There is a match at the current position.

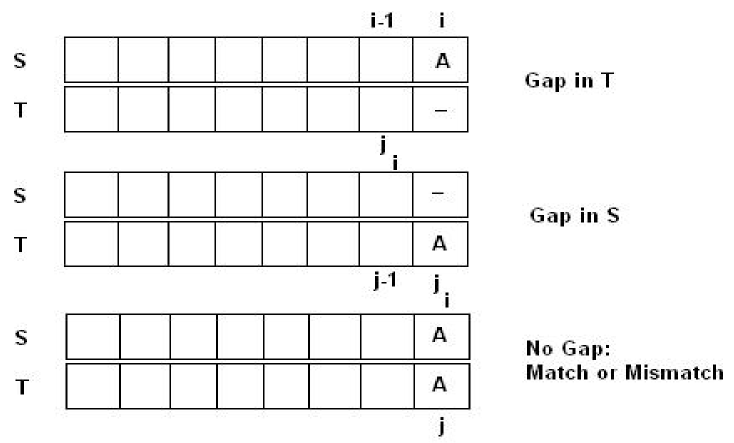


Figure 4: The different cases to consider in making a locally optimal choice.

We then use the possibility that produces the maximum score. We express this mathematically by the recursive formula for F_{ij}:

$$F_{ij} = \max \begin{cases} F_{i-1j} + d & \text{insert gap in } T \\ F_{ij-1} + d & \text{insert gap in } S \\ F_{ij} + s(i,j) & \text{match or mutation} \end{cases} \tag{2}$$

After traversing the matrix, the optimal score for the global alignment is given by F_{mn}. The traversal order needs to be such that we have solutions to given subproblems when we need them. Namely, to compute F_{ij}, we need to know the values to the left, up, and diagonally above F_{ij} in the table. Thus we can traverse the table in row or column major order or even diagonally from the top left cell to the bottom right cell. Now, to obtain the actual alignment we just have to remember the choices that we made at each step.

4.4 Step 4: Constructing an Optimal Solution

Note the duality between paths through the matrix F and a sequence alignment. In evaluating each cell F_{ij} we make a choice in choosing the maximum of the three possibilities. Thus the value of each (unitialized) cell in the matrix is determined either by the cell to its left, above it, or diagonnally above it. A match and a substitution are both represented as traveling in the diagonal direction; however, a different cost can be applied for each, depending on whether the two base pairs we are aligning match or not.

To construct the actual optimal alignment, we need to traceback through our choices in the matrix. It is helpful to maintain a pointer for each cell while filling up the table that shows which choice we made to get the score for that cell. Then we can just follow our pointers backwards to reconstruct the optimal alignment.

4.5 Analysis

The runtime analysis of this algorithm is very simple. Each update takes $O(1)$ time, and since there are mn elements in the matrix F, the total running time is $O(mn)$. Similarly, the total storage space is $O(mn)$.

For the more general case where the update rule is perhaps more complicated, the running time may be more expensive. For instance, if the update rule requires testing all sizes of gaps (e.g. the cost of a gap is not linear), then the running time would be $O(mn(m+n))$.

4.6 Optimizations

The algorithm we showed is much faster than the originally proposed strategy of enumerating alignments and performs well for sequences up to 10 kilo-bases long. However, for whole genome alignments, the algorithm shown is not feasible, but we can make modifications to it to further improve its performance.

4.6.1 Diagonal

One possible optimization is to ignore Mildly Boring Alignments (MBA), that is, alignments that have too many gaps. In other words, we can limit ourselves to stay within some distance W from the diagonal in the matrix of subproblems. This will lead to a time/space cost of $O((m+n)W)$.

Note, however, that this strategy no longer guarantees that we will obtain the optimal alignment. If one wants to maintain this guarantee, this strategy can still be used to obtain a lower bound on the optimal score, and then use this bound to limit the space of the matrix that we have to cover.

4.6.2 Using stretches of high conservation

Suppose now that we can quickly find common stretches of very high conservation between the two sequences.

We can then use these stretches to anchor the optimal alignment. Then we are left with finding optimal alignments for the subsequences between these stretches, greatly reducing the size of the problem and thus the running time.

The question left, though, is how can we find those conserved stretches very fast. This question of finding local alignments will be the topic of the next lecture.