

After the Human Genome Project: Where Do We Go from Here?

Gene and protein databases: The Human Genome Project has given us a 3-billion-letter library of the human genome. This will be added to equally large libraries of genome sequences for other organisms, and also libraries of protein sequences for humans other organisms. How do we use all this information?

Answer: Researchers are continually finding new genes and proteins, or attempting to determine the role, structure, or function of these genes and proteins. In order to do this, they look for **similar sequences** in these databases, to try to discover similarities in structure and/or function to the genes or proteins they are studying.

The problem: How can we search these huge databases quickly and accurately for sequences similar to the one we are investigating?

Enter Operations Research

Operations Research: the study of the efficient organization and use of large-scale systems.

The Human Genome: an excellent example of a large-scale system.

Optimization Problem: the generic problem for operations research. Optimization problems have three components:

system: the structure for which you want to get the “best” solution.

constraints: what restrictions you have on obtaining your solution.

objective: how you can compare two solutions to determine which is “better” .

The Sequence Matching Problem

The basic problem: You have a **main sequence** of letters (such as the human genome), and you have a specific **subsequence** (say a gene you have been studying) that you think is contained somewhere in the larger sequence. How do you find a “good” match for your sequence?

The Hitch: There may be differences between the two sequences (due to errors, different species, or random mutations), so that your sequence may not match exactly with a subset of the larger sequence.

The Goal: You want a “best fit” of your sequence into the main sequence.

Example

Suppose your main sequence is

AGCGACGATGCGTATCCGTC

and you are searching for the sequence

GCGTAT

in this main sequence. Then there is a perfect match

AGCGACGATGCGTATCCGTC

GCGTAT

If, however, the sequence you are searching for were instead

AGTATT

Then there would be no perfect match. There are many “near” matches, such as

AGCGACGATGCGTATCCGTC

AGTATT

or

AGCGATGATGCGTATCCGTC

AGTATT

Questions: What is the “best” match of your sequence in the main sequence? How can you find this best match?

The Sequence Matching Optimization Problem

Constraints: How are you allowed to match sequences?

“No-gap” model: Wherever in the main sequence you want to match the subsequence, this sequence must be matched in order, letter-for-letter, to the main sequence.

A simple objective: Match as many letters as you can. That is, you want to find a position along the main sequence where there are as **few mismatched letters** as possible.

Good . . .

```
AGCGACGATGCGTATCCGTC
AGTATT
```

(4 mismatches).

Better . . .

```
AGCGACGATGCGTATCCGTC
AGTATT
```

(3 mismatches).

Best . . .

```
AGCGACGATGCGTATCCGTC
AGTATT
```

(2 mismatches).

A perfect match has objective 0.

Weighted Matching

A more sophisticated model: Suppose you believe that some mismatches are worse than others. Then you could put a **penalties** on each of the mismatch possibilities AC, AG, AT, CG, CT, GT, and the objective is now the **sum** of the mismatch penalties.

For example, suppose you place a penalty of 6 to CA, a penalty of 2 to GT, and a penalty of 1 to everything else. Then the preference of matching above exactly reverses, with the first match having total penalty 4, the second having total penalty 5, and the third having total penalty 7.

A Solution Algorithm

1. **Start your sequence** at the left-hand end of the main sequence, and simply move it across the main sequence.
2. **At each location** add up all of the mismatch penalties, and that will give you the penalty of the matching if it occurred in that location.
3. **The smallest of all of these totals** determines the starting point of the sequence with the **minimum total mismatch penalty**.

For our example here are the mismatch penalties:

4	7	16	5	5	11	6	12	4	6	7	7	11	11	5						
A	G	C	G	A	C	G	A	T	G	C	G	T	A	T	C	C	G	T	C	
												A	G	T	A	T	T			
												11=1	+1	+0	+6	+1	+2			

Gaps in Sequence Matches

Imperfect sequence matches occur because the DNA — through evolution and mutation in cell replication — has miscopied a portion of DNA. This is caused by three errors in the various RNA transcription processes:

1. A nucleotide can simply be miscopied, causing one letter to be changed to another in the new sequence;
2. An extra nucleotide can be inserted, causing an extra letter to appear in the new sequence;
3. A nucleotide can be deleted, causing a letter to disappear in the new sequence.

How do we include this in our matching model?

A New Model

Allow for the placement of **gaps** in either sequence, that is, put some symbol (we use `_`) that indicates an unmatched symbol. Specifically,

- _ in the **main** sequence means that the adjacent symbol in the **subsequence** will not be matched.
- _ in the **subsequence** means that the adjacent symbol in the **main sequence** will not be matched.

For example, the match

```
AGCGA_CGATGCGTATCCGTC
AGT_ATT
```

has two matches (the two A's), two mismatches (C-T and G-T), and one unmatched base in each sequence (the two other G's).

The Optimization Problem for the Gaps Model

Constraints: A solution for the gaps model now consists of a side-by-side placement of the main and subsequences, after inserting a set of _'s in both sequences.

Objective: We determine in advance a set of **mismatch penalties** that apply to straight miscopies of sequence letters, and **gap penalties** that apply when there is a gap in either sequence. Then the objective for any solutions is now determined by going through the positions of the subsequence, adding up the mismatch or gap penalties according to how this sequence matches the main sequence.

Simplified penalties: For this class we will assume that all mismatch penalties are the same, and all gap penalties are the same (but not necessarily the same as the mismatch penalty).

Example: For the solution above, suppose we had a mismatch penalty of 3 (for any mismatch), and a gap penalty of 1. Then the total weight of the solution is $2 \times 3 + 2 \times 1 = 8$.

```
AGCG A _ C G A T G CGTATCCGTC
      A G T _ A T T
      0+1+3+1+0+0+3 = 8
```

The Matching Problem with Gaps

Given: a main sequence, a subsequence and specified mismatch and gap penalties

Find: the best matching of the subsequence into the main sequence — allowing gaps in either sequence — so as to obtain the **smallest total mismatch + gap penalty.**

Note: We never need to consider sequences with a `_ _` match. (Why?)

For the sake of simplicity, we will not allow gaps to be placed at the very **beginning** of either sequence, that is, the first element of the subsequence must be matched (or mismatched) with an element of the main sequences.

Finding an Algorithm for the Gaps Model

Suppose we try to solve the problem similarly to the no-gaps model. Then we would slide the subsequence along the main sequence, and then at each point we would place gaps at all possible locations along both sequences, and then total up all the mismatch-gap penalties.

Question: How long would this take?

Answer: Finding the minimum weight matching of a 50-base sequence into a 50-base main sequence allowing only 20 gaps in either sequence involves testing over 10^{30} possible gap placements in the two sequences!

We need to find a better algorithm . . .

A Matrix Representation of the Matching Problem

Let's do a slightly smaller example, matching only AGTA into main string GACGATG. We will use mismatch penalty 3 and gap penalty 1.

A good way to represent matching solutions is by using a **matching grid**. This is a grid whose columns are labeled with the letters of the main sequence, and whose rows are labeled by the letters of the sub-sequence.

	G	A	C	G	A	T	G
A							
G							
T							
A							

Representing the Matching

A solution can be represented by filling in the squares of the matching grid as follows:

1. Start at the top row of the grid. This will be the starting point for the matching, and the row and column letters are the first letters to be matched.
2. From each point in your construction of a matching solution, you may go in 3 possible directions:
 - Right:** This will insert a gap in the **subsequence**.
 - Down:** This will insert a gap in the **main** sequence.
 - Diagonally (\searrow):** This will match the next row and column letters in the two sequences.
3. When you get to the bottom line of the grid, you have completed your match.

A Shortest Path Model

Looking at the above procedure for constructing a matching solution, we see that we are actually forming a **path** through the matching grid, starting at the top and moving down the grid, going right, down, or diagonally at each step, until we reach the bottom.

Each of the steps in this path can incur a possible penalty, depending upon whether it corresponds to a gap insertion or mismatch. In particular,

Going right incurs a cost equal to the **gap penalty** for the **subsequence**.

Going down incurs a cost equal to the **gap penalty** for the **larger** sequence.

Going diagonally incurs **no** cost if the two letters identifying the row and column you reach are the **same**, and a cost equal to the **mismatch penalty** for that pair if they are different.

For example The match

GA_CGATG

AGT_A

corresponds to the path

	G	A	C	G	A	T	G
A		(0)					
G		(1)					
T			(3)	(1)			
A					(0)		

with value $0+1+3+1+0=5$.

Goal: Among all paths from the top of the grid to the bottom of the grid which move only right, down, or diagonally, find the **shortest path**, that is, the path with the **smallest total penalty**.

This is a classic problem in **operations re-
search**.

Finding Shortest Paths

Although this is a nice way to visualize the matching problem, it does nothing to improve our ability to find the best match, since there is still an exorbitant number of possible paths of this sort.

What we need is a **systematic way** of finding the shortest path, and specifically, one that **does not** try all possible paths to find the shortest one.

All Shortest Paths for the Matching Example

	G	A	C	G	A	T	G
A	3	0	1	2	0	1	2
G	4	1	2	1	1	2	1
T	5	2	3	2	2	1	2
A	6	3	4	3	2	2	3

We could actually start **anywhere** on the grid, and, using the penalties for moving as given above, find the minimum penalty path to the top of the grid, or equivalently, the minimum penalty of matching the initial parts of the main sequence and subsequence **ending at that row and column of the grid**.

The numbers on the grid above are the total distances of a shortest path from that point to the top of the grid. The colored lines represent the corresponding paths.

Finding the Best Match

We are going to find the best match — or equivalently, the shortest path — by successively matching larger and larger pieces of the subsequence to the main sequence. At each stage, we will be **using information about the best shorter sequences** to make it easier to match larger ones.

In particular, we are going to try to find the shortest paths from the top of the grid to each square of the **top level**, then use these to find shortest paths to each square of the **next level**, and so on, until we finally get shortest paths to each square of the **bottom level**. The **smallest penalty** of path to a bottom level will correspond to the minimum penalty matching of the entire subsequence.

Finding a Shortest Path From What Went Before

Suppose I want to find the shortest path to a particular square from the top of the matching grid. **What** would I need to know in order to do this?

Answer: I would need to know

- what my **last step was**, and
- how long the (shortest) path was to the point **from which** I made this last step.

I can then determine the shortest path to that square by going through all of the ways I can make that final step to get to that square.

Note that I only need to know the length of the shortest paths to the 3 squares to the left, upper-left, and upper squares to the current square.

The Computations

Let's take our example, and consider the grid square corresponding to Row 4 and Column 4. Suppose we knew the shortest path to each of the three squares above, to the left, and diagonally up from this square:

	...	C	G	...
⋮				
T	—	3	2	—
A	—	4	?	—
⋮				

Now take a step to this square from each of these previous squares:

final step	penalty for step	total penalty
down	1	$2+1=3$
right	1	$4+1=5$
diagonal	3	$3+3=6$

The shortest path is obtained by choosing the **smallest** of these numbers, in this case 3, and the shortest path comes from above (corresponding to placing a gap in the main sequence above letter A of the subsequence.)

What Happens on the Edge of the Grid?

We also need to tell what happens on the top and along the left-hand edge, where some or all of the adjacent numbers are missing.

upper-right-hand corner: Any path that starts here *must* match the first letter of the subsequence to that of the main sequence, so the number here the same as the diagonal-step penalty. In our example:

	G	...	
A	?		grid entry = 3 (mismatch)
⋮			

rest of right-hand-side: Any path that goes through this square must correspond to a **gap in the main sequence**, and so the number here is the same as the down-step penalty. In, say, the third row of our example:

	G	...	
⋮			
G	4		grid entry = 4+1 = 5
T	?		
⋮			

rest of top row: Any path that goes through this square either **starts** a match at this point or corresponds to a **gap in the subsequence**, and so corresponds to the **minimum** of the right-step and the match/mismatch penalties. In, say, the fourth column of our example:

	...	C	G	...
A		1	?	
⋮				

final step	penalty for step	total penalty
right	1	1+1=2
“diagonal”	3	3

The minimum is 2, and the path comes from the left (corresponding to placing a gap in the subsequence at letter G of the main sequence).

Order is Important

Now we can compute the shortest-path value for each square, but **only if we have already computed the values of the three squares up-left-diagonally adjacent to that square.**

Question: How can we guarantee to always have these values available when we get ready to compute the value for a square?

Answer: We must compute the values for the matching grid in a specific order:

1. The rows must be processed in order from top to bottom.
2. The values in each row must be processed from left to right.

Using the Computer: **IDEAS**

The **IDEAS** *Sequence Matching* module will allow you to see how paths correspond to matchings and also see what the smallest-penalty matching is. Simply give your two sequences, along with the mismatch and gap penalties. Then you can either fill in the path by clicking on each succeeding square, or have the computer find the shortest path for you. Just follow the directions.

Using the Web: **BLAST**

BLAST is a web-based public sequence finder. You can get it by clicking on the BLAST item on the OR006 webpage, or by going to

<http://www.ncbi.nlm.nih.gov/blast/Blast.cgi>

and then clicking on **nucleotide BLAST** under Basic Blast. If you want to search for a protein sequence, simply click on **protein BLAST** under Basic Blast.

How Blast Works

BLAST takes as input any sequence of nucleotides and searches for a DNA match in several databases. (The current number of sequences available for search is over 90 *billion!*) It actually looks for best matches of *subsequences* of your sequence to those of one of the elements in the database, scoring these with an **E-value** representing the expected number of times a match this good would occur in an arbitrary sequence.

Low E-values means an excellent match. Most of the values are written as decimals, but very small E-values look like e-132, which means that the expected number of times you would find this good a match at random is about 1 in 10^{58} , a *very* small number!

Inputting Your Sequence at the BLAST Website

Place your sequence into the box at the top, and hit **BLAST!** After some period of time (many minutes in the middle of the day) the computer will list the best choices, along with alignment data for each of them, and the precise match as we showed above.

You can also control how BLAST looks for a match, in terms of organism class and similarity of match, by using the options on the webpage.

The output from BLAST lists the best matches found within the category you specified, with details of the precise sequence match that was made.

For example, consider the following sequence — corresponding to the first part of the SRY gene:

```
gcagtgtcac taggccggct gggggccctg ggtacgctgt agaccagacc gcgacaggcc  
agaacacggg cggcggcttc gggccgggag acccgcgag ccctcggggc atctcagtgc  
ctcattccc accccctccc ccgggtcggg ggaggcggcg cgtccggcgg agggttgagg
```

When matched with the “primate” database, the match was essentially perfect into a subset of the SRY gene, with E-value $3e-64$ = a *very* small number! When searched with the “rodentia” database, there were no significant matches. (If you chose “Somewhat similar sequences (blastn)” as the BLAST algorithm you get a match, but a poor and unrelated one). This indicates that SRY essentially does not have a corresponding gene in rats.