Algorithms in Bioinformatics

CS 482

Bin Ma

LaTeXed by Sibelius Peng

Preface

Disclaimer Much of the information on this set of notes is transcribed directly/indirectly from the lectures of CS 482 during Winter 2022 as well as other related resources. I do not make any warranties about the completeness, reliability and accuracy of this set of notes. Use at your own risk.

For any questions, send me an email via https://notes.sibeliusp.com/contact.

You can find my notes for other courses on https://notes.sibeliusp.com/.

Sibelius Peng

Contents

1

Introduction

This course is officially titled as "Computational Techniques in Biological Sequence Analysis". However, this is an old title and this course has been offered for over approximately twenty years. It should actually be called "Algorithms in Bioinformatics". Around 1980, this area has a different name: Computational Biology. You may also hear another name: DNA computing. It is another area, and it does not solve biological problem. What is bioinformatics? It is biology + informatics. Biology is the reason, goal, purpose and informatics is the method.

Biology can be studied at different scales. In old days, biology tries to study organisms, namely living things, such as bacteria, animals. This is because people didn't have tools to study at a lower level at that time. Now people study organs and tissues. Then people can look into cell level, molecular level. At molecular level, DNA is chain of nucleotide bases. Protein is chain of amino acids.

There are a lot of public and free molecular data. There are tremendous amount of public biomolecule data and free software. For example:

- NCBI's sequence data bank: https://www.ncbi.nlm.nih.gov/nuccore/NC_045512
- PDB protein structure database: https://www.rcsb.org/structure/6vxx

Why do people do bioinformatics? The goal is to understand life at molecular level. Especially, for human health. For example, sequencing SARS-Cov2 genomes allowed people study the evolution of this virus. Study the structure of the spike protein and its interaction with the host cells. A lot of human diseases are related to genetics.

There are two areas of bioinformatics.

- Determine the molecule information, by analyzing the data produced by measuring instruments. Typically the data is in large scale and high throughput, which is hard for people to look at.
- Use the molecular data to make inference. For example, make prediction given the existing data.

Consider an example of genome sequencing. From wikipedia,

> The Human Genome Project (HGP) was an international scientific research project with the goal of determining the base pairs that make up human DNA, and of identifying, mapping and sequencing all of the genes of the human genome from both a physical and a functional standpoint. It remains the world's largest collaborative biological project. Planning started after the idea was picked up in 1984 by the US government, the project formally launched in 1990, and was declared complete on April 14, 2003. Level "complete genome" was

achieved in May 2021.

Bioinformatics played an essential role in analyzing the data and assemble the genome. Today one can sequence a human's genome with $<$ \$1000 in a couple of weeks. Bioinformatics is the key to utilize the NGS (next generation sequencing) data for genome sequencing. As such, today's cancer treatment starts to become personalized. And many new drugs now require gene sequencing as companion diagnostic.

1.1 Objectives of this course

- Know bioinformatics
  - Purpose and method
  - General topics
- Learn classic problems and algorithms in bioinformatics
- Learn wide-applicable computational techniques
  - String algorithms
  - Hidden Markov Model
  - Log likelihood ratio score
  - Statistical validation
  - A bit of machine learning

Some typical problems:

- Gene prediction problem
- Find the longest shared substring between human and mouse genomes. If we want to find similarities instead of exact matches, this will lead us to the homology search problem.
- Peptide Identification

2

Sequence Alignment

2.1 Biology

Consider two protein sequences:

- AVP78042.1 spike protein: `MLFFL...`
- YP_009724390.1 surface glycoprotein: `MFVFL...`

They look similar, how do we know these two proteins are similar? It's better to visualize them properly so that we can see the similarity. This is done by sequence alignment, and there are many existing tools: such as Clustal Omega. It's a common belief that these two proteins are developed from a common ancestor, then they evolve from an evolution tree. This is called homology.

There are two classes of nucleotide bases:

- Purine: A and G
- Pyrimidine: T and C

Base pairs are bonded by hydrogen bonds. Also, G-C bind stronger because of 3 H-bonds. DNA molecule is oriented.

So we know that DNA is double-helical, with two complementary strands. And the complementary bases: A-T, G-C. For example, the *reverse* complement of `AAGGTAGC` is `GCTACCTT`, because DNA is oriented.

Now consider DNA mutation. DNA mutates with a small probability when inherited by the offspring. For example, one base can be substituted by another, because there might be copy errors. This creates different alleles of the same gene. From wiki, allele is one of two, or more, forms of a given gene variant. When we inherit the DNA from parents, we only inherit half of each parent's genome. These together cause the differences between individuals of the same species.

Single Nucleotide Polymorphisms is a germline substitution of a single nucleotide at a specific position in the genome. Single base variation between members of a species. For Human, 90% of all human genetic variation is caused by SNPs. SNPs occur every 100 to 300 bases along the 3-billion-base human genome. It's a major risk for genetic disease, because when one base pair mutates, it will cause the express protein's functions.

2.2 Compare DNA sequences

The most often used distance on strings in computer science is Hamming distance. This makes some sense on comparing DNA sequences in some cases: substitution. But there are other mutations: insertion/deletion (indel), which cannot be modelled correctly by Hamming distance. Other DNA rearrangements can also happen. But substitutions and indel are the two mutations we concern the most for this course.

Edit distance

Instead, we can use **edit distance**. How "far" away are two sequences from each other? Edit distance is defined to be the minimum number of edit operations needed to convert one to another. Here edit operations include substitutions and indels.

Note that Edit distance is a distance metric:

- Identity: $d(x, y) = 0$ if and only if $x = y$.
- Symmetry: $d(x, y) = d(y, x)$.
- Triangular inequality: $d(x, z) \leq d(x, y) + d(y, z)$.

Now we prepare for the algorithm. For convenience of the proof, we treat each occurrence of the same letter different. For example, `ATAA -> ATA, A` can be done by either deleting the 2nd or 3rd letter `A` from the first string. These are different editing paths. This does not affect our definition of edit distance, but makes our later proof more precise.

Now it's ready to develope the dynamic programming algorithm for edit distance. Let $D[i, j]$ = edit distance between $S[1..i]$ to $T[1..j]$. Consider the edit operations associated with $S[i]$ and $T[j]$ the optimal edit operations. One of the following cases will happen:

1. $S[i]$ is deleted: $D[i, j] = D[i - 1, j] + 1$
2. $T[j]$ is inserted: $D[i, j] = D[i, j - 1] + 1$
3. $S[i]$ becomes $T[j]$: $D[i, j] = D[i - 1, j - 1] + \delta(S[i], T[j])$

where $\delta(S[i], T[j]) = 0$ if $S[i] = T[j]$ and 1 if not.

Algorithm 1: Dynamic Programming Algorithm for Edit distance

```
1 D[0,0] = 0
2 D[0,i] = i for i = 1..|S|
3 D[i,0] = i for i = 1..|T|
4 for i ← 1..|S| do
5 |   for j ← 1..|T| do
6 |   |   D[i,j] = min{D[i-1,j]+1, D[i,j-1]+1, D[i-1,j-1]+δ(S[i],T[j])}
7 return D[|S|,|T|]
```

Longest Common Subsequence

Another way to evaluate the similarity of two sequences is through LCS. A subsequence is obtained by deleting some of the letters from the supersequence and concatenating the remaining letters together. For example, LCS of `ATGCATTTA` and `ATGTACTTTC` is `ATGATTT`. LCS can be computed with dynamic programming as well.

2.3 Alignment

The third way to compare to sequences is through sequence alignment. We want to insert spaces (-) to two sequences so that we can align them together and they are most similar column-wisely. For example,

```
ATGCA-TTTA
|||  |  || |
ATGTACTT-A
```

By "similar", we usually need to use a scoring function. We define the alignment score to be **the total of column scores**. And each column is assigned by a constant score depending on matching conditions. For example, simple score scheme would be

- Match = 1
- Mismatch = -1
- indel = -1

Consider two alignments with the score scheme above:

```
AATGCGA-TTTT
||  |  |||
G-TG--ACTTTC
```

has score 0.

```
AATG-CGATTTT
||  |  ||
G-TGAC-TTTC-
```

has score -1.

As a side note, alignment can "simulate" LCS and edit distance. We have this following table for the scoring system:

	alignment	LCS	edit dist
match	+1	1	0
mismatch	-1	0	-1
indel	-1	0	-1

Now we can develop the develop the dynamic algorithm for the alignment. Let $f(a, b)$ be the scoring scheme for a column with a and b. Here one of a and b can be the dash character -. Thus $f(\text{-}, x)$ and $f(x, \text{-})$ represent scores of indels. The input is S, T. Let $D[i, j]$ be the optimal alignment score for $S[1..i], T[1..j]$.

Now we want to derive a recurrence relation. Suppose we are to align $S[1..i], T[1..j]$. Consider the *last column* of the optimal alignment. Three cases can happen.

$S[1..i-1]$ $S[i]$	$S[1..i-1]$ $S[i]$	$S[1..i-1]$ -
$T[1..j-1]$ $T[j]$	$T[1..j-1]$ -	$T[1..j-1]$ $T[j]$
(1)	(2)	(3)

Note that in each case, the sub-alignment without the last column is an optimal one. Prove by contradiction, assume there's a better sub-alignment without the last column. Then the optimal alignment (for the whole) should have used this better sub-alignment, instead of the current sub-alignment.

Then we have the recurrence for these three cases:

(1) $D[i,j] = D[i-1,j-1] + f(S[i], T[j])$

(2) $D[i,j] = D[i-1,j] + f(S[i], \text{-})$

(3) $D[i,j] = D[i,j-1] + f(\text{-}, T[j])$

Algorithm 2: Sequence alignment

```
1 D[0,0] = 0
2 for i ← 1..m do
3 |   D[i,0] = i× indel
4 for j ← 1..n do
5 |   D[0,j] = j× indel
6 for i ← 1..m do
7 |   for j ← 1..n do
8 |   |   D[i,j] = max { D[i-1,j-1] + f(S[i], T[j])
  |   |                { D[i-1,j] + f(S[i], -)
  |   |                { D[i,j-1] + f(-, T[j])
9 return D[m,n]
```

Then the time complexity is $O(mn)$ where $|S| = m, |T| = n$. In practice, when we run the DP algorithm and build the DP table, we can add arrows from cell to cell: which of the three cases (sub-alignments) is chosen to get the current cell. Then we can backtrack the arrows and get the actual alignment. For backtracking, it is $O(n+m)$. Space complexity is $O(nm)$.

In practice, we don't need to record/store these arrows. Instead, when we backtrack, we can calculate the arrows based on the max of three options dynamically, which takes $O(m+n)$. This saves the cost of storing arrows, which costs $O(mn)$.

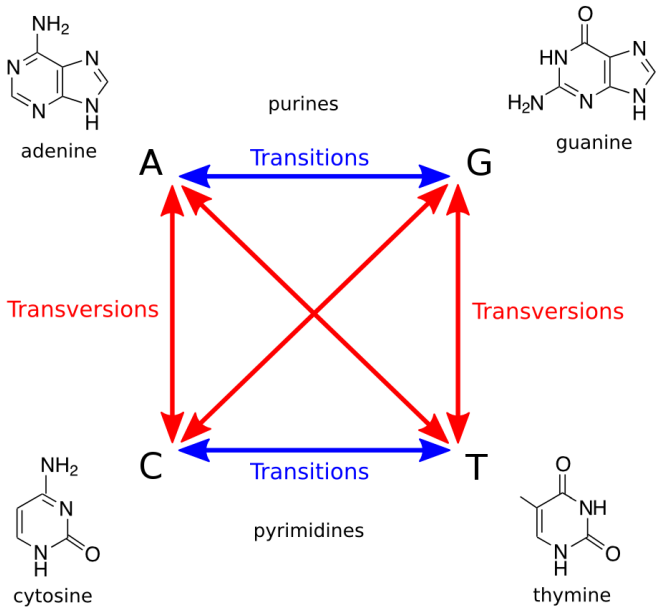
Moreover, if only score is needed, then space complexity can be reduced to linear space.

Note that the algorithm is designed for any score scheme $f(x,y)$. We indeed separate the algorithm and scoring. So we can optimize score scheme and the algorithm independently. Dijkstra once said:

> The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

2.4 Score Function

> Transition vs. Transversion
>
> Recall that within DNA, there are two classes. One is purines (A, G), and the other is pyrimidines (C, T). Within the same class, they have similar chemical structure. Moreover, the mutation within the same class is called transitions; across the two classes, is called transversions. Transition is cheaper than transversions.



Picture from https://dodona.ugent.be/en/activities/1351011626/

Transition happens more frequently 2/3 of SNPs are transitions. In other words, transition is easier and therefore should be less penalized. For example,

```
AAAGCAAA        AAAGCAAA
AAAT-AAA   vs   AAA-TAAA
```

Observe that GT is transversion and CT is transition. The right alignment is probably better. This can be easily achieved by changing score scheme $f(a, b)$.

So how to build a score function? First, we need to know what you want. There are two purposes:

1. the optimal alignment reveals the true evolutionary history.
2. high score indicates homology (derived from same ancestor).

We want purpose 1 if possible, but purpose 2 is also useful.

For purpose 1, note that we might be wrong: score function might not have the power to reveal the history. For example, if the history goes as `A -> T -> A`, then in the sequence alignment, the final comparison is between `A` and `A`, which will not give us the true history. So you should never trust that an alignment must be right. It just optimizes the score. Should we give up purpose 1 at all?

For purpose 1, the optimal alignment may be *approximately* correct *under certain conditions* in practice. As long as we know the limitation, we can still use it. For example, for the following alignment, it is "very likely" the alignment is approximately equal to the evolutionary history.

```
ACGTATTACCGG-TTACCG
|||  ||||||||  ||||||
ACGGATTACCGGATTACCG
```

So we should keep in mind that when the score is low, alignment itself is not too useful.

Now let's try to improve our score function. Consider two alignment (with gaps):

```
AGATTTTTTTC     AGATTTTTTTTTC
AGA---TTTTC     AGA-T-T-T-T-C
```

The left seems "simpler" than the right, intuitively. Indeed, during evolution, indels are relatively rare. However, insertion or deletion a segment of k consecutive bases is much easier than k scattered indels. But our current scoring method (adding up column scores) cannot distinguish the two. Currently, a gap of length k costs $k\times$indel. Thus, this is called the **linear gap penalty**. Denote the penalty function by $g(x)$, and it should have negative value. Left's penalty is $g(3)$ and right is $3 \times g(1)$.

Consecutive insertions or deletions are called a gap. Suppose the gap penalty of a length k gap is $g(k)$ instead of the simple $c \times k$. Assume $g(x) + g(y) \leq g(x + y)$, thus a convex function. Otherwise does not serve the purpose of grouping indels. In this case, can the old DP algorithm still work? Recall the three cases:

$S[1..i-1]$ $S[i]$	$S[1..i-1]$ $S[i]$	$S[1..i-1]$ -
$T[1..j-1]$ $T[j]$	$T[1..j-1]$ -	$T[1..j-1]$ $T[j]$
(1)	(2)	(3)

First case is the same. The second case might be wrong. The last column might depend on previous columns. If the second to the last column of T is already a dash, then adding a new dash will increase the length of gap. The we cannot use DP algorithm anymore, in the sense that we cannot prove its correctness. We do not know the contribution of the last column to the gap penalty in the last two cases.

Assume we know the length of gap, then we can apply the recurrence relation. We still use $D[i, j]$ to denote the optimal alignment score of $S[1..i]$ and $T[1..j]$. We change cases 2 and 3 to include the last gap (not the last column). Then three cases change accordingly:

$S[1..i-1]$ $S[i]$	$S[1..i-k]$ $S[i-k+1..i]$	$S[1..i]$ - - - - -
$T[1..j-1]$ $T[j]$	$T[1..j]$ - - - - -	$T[1..j-k]$ $T[j-k+1..j]$
(1)	(2)	(3)

Then $D[i, j] =$ max of the following three cases:

(1) $D[i-1, j-1] + f(S[i], T[j])$

(2) $\max_{1\leq k\leq i} D[i-k, j] + g(k)$

(3) $\max_{1\leq k\leq j} D[i, j-k] + g(k)$

Now the time complexity will change: $O(mn(m+n))$, which is cubic.

In bioinformatics, very often we face the choice between:

- Reality: How close it approximates the real biology.
- Simplicity: How easy it can be computed.

We can simplify $g(k)$ a little bit. We basically want a function that grows slower than linear. We introduce **affine gap penalty**, in contrast to linear gap penalty:

$$g(k) = a + b \cdot k$$

where a is the gap open penalty: whenever we have a gap, we pay for it; b is the gap extension penalty: for example, once we open the gap, we pay less for the second indel within a group.

Example: Affine gap penalty

match $= 1$; mismatch $= -1$; gap open $= -5$; gap extension $= -1$.

```
ATAGG--AAG
||  ||  | |
ATTGGCAATG
```

6 match, 2 mismatch, 1 gap open, 2 gap extension, then the score is

$$6 - 2 + (-5 - 1 \times 2) = -3$$

```
ATAGG-AA-G
||  ||  || |
ATTGGCAATG
```

Similarly, the score is

$$7 - 1 - 5 - 1 - 5 - 1 = -6$$

Now the old algorithm doesn't work anymore. Consider the last column of an alignment again:

```
AT-GG-    ATGG--
||  ||    ||  |
ATTGGC    ATTGGC
```

When considering the last column, we need to know the second to the last column. When the last column is an indel, the added cost depends on the previous column. Because by induction we know the optimal solution for $D[i, j-1]$, we can encode the previous column's configuration. We compute the optimal solution by limiting the last column to one of the following three configurations:

```
ATAGG      ATAGG-     ATAGGC
||  ||     ||  ||     ||  ||
ATTGG      ATTGGC     ATTGG-
```

$D_0[i,j]$ $D_1[i,j]$ $D_2[i,j]$

$D_0[i,j]$ requires that the last column must be $S[i]$ v.s. $T[j]$, not indel. $D_1[i,j]$ is - v.s. $T[j]$, and $D_2[i,j]$ is $S[i]$ v.s. -. We then can develop recursion relationship easier by defining more subproblems. We only distinguish them by the last column, there is no constraint for columns before the last column.

Now let's examine how to calculate $D_0[i,j]$. There are three cases:

$$\begin{matrix} S[1..i-2]S[i-1] & S[i] \\ T[1..j-2]T[j-1] & T[j] \end{matrix} \longrightarrow D_0[i,j] = D_0[i-1,j-1] + f(S[i], T[j])$$

$$\begin{matrix} S[1..i-1] & \text{-} & S[i] \\ T[1..j-2] & T[j-1] & T[j] \end{matrix} \longrightarrow D_0[i,j] = D_1[i-1,j-1] + f(S[i], T[j])$$

$$\begin{matrix} S[1..i-2] & S[i-1] & S[i] \\ T[1..j-1] & \text{-} & T[j] \end{matrix} \longrightarrow D_0[i,j] = D_2[i-1,j-1] + f(S[i], T[j])$$

We then can do the similar thing for $D_1[i,j]$:

$$\begin{matrix} S[1..i-1]S[i] & \text{-} \\ T[1..j-2]T[j-1] & T[j] \end{matrix} \longrightarrow D_1[i,j] = D_0[i,j-1] + \text{gapopen} + \text{gapext}$$

$$\begin{matrix} S[1..i] & \text{-} & \text{-} \\ T[1..j-2] & T[j-1] & T[j] \end{matrix} \longrightarrow D_1[i,j] = D_1[i,j-1] + \text{gapext}$$

$$\begin{matrix} S[1..i-1] & S[i] & \text{-} \\ T[1..j-1] & \text{-} & T[j] \end{matrix} \longrightarrow D_1[i,j] = D_2[i,j-1] + \text{gapopen} + \text{gapext}$$

The relation for $D_2[i,j]$ is symmetric. In summary, we have the recurrence relation

$$D_0[i,j] = f(S[i],T[j]) + \max \begin{cases} D_0[i-1,j-1]; \\ D_1[i-1,j-1]; \\ D_2[i-1,j-1]; \end{cases}$$

$$D_1[i,j] = \text{gapext} + \max \begin{cases} D_0[i,j-1] + \text{gapopen}; \\ D_1[i,j-1]; \\ D_2[i,j-1] + \text{gapopen}; \end{cases}$$

$$D_2[i,j] = \text{gapext} + \max \begin{cases} D_0[i-1,j] + \text{gapopen}; \\ D_1[i-1,j] + \text{gapopen}; \\ D_2[i-1,j]; \end{cases}$$

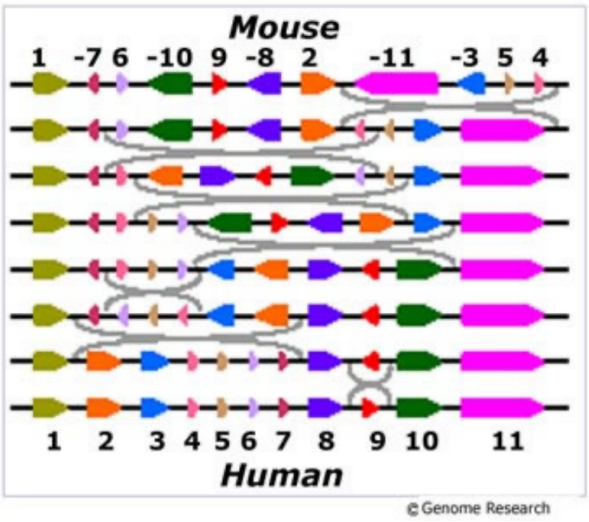
Note the grayed cases can't be optimal so can be safely removed.

This is still DP algorithm. We need to be careful when backtracking. The running time is $O(nm)$, might be approximately 3 times slower, but better than the general gap penalty's cubic time. This is okay because the model is more expressive. This model is first published in 1982, by Gotoh, O.: *An improved algorithm for matching biological sequences*.

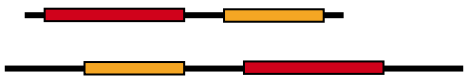
3

Local Alignment and Linear Space Alignment

In biology, we are not comparing S, T directly. Instead, we are comparing small areas of S, T.



Interestingly, if we compare chromosome X of mouse and human, they are quite similar. Each colored block is relatively conserved, but different in orders and orientations. Seven inversions are required to put them in the correct order and orientation. This is called "sorting by reversals". This is an interesting study from Pavel Pevzner.



In this case, if we do the global/sequence alignment, we can align only one part: we will lose yellow part when we align red. Conserved regions are "local" to the genome/chromosome. But previous alignment is "global". We need a proper model to define "local" similarity.

For the problem of local alignment, we are given two sequences S and T. We want to find substrings of S and T that maximizes the alignment score. I.e., The indels at the beginning and end of the two strings are free.

Local alignment score is at least 0, because for the worst case, we have two empty strings. The model only makes sense for alignment but not edit distance nor LCS. Is the optimal local alignment a local part of an optimal "global" alignment? No. If we align `AT` and `TA`, the global alignment would be $\begin{matrix}\texttt{AT}\\ \texttt{TA}\end{matrix}$

if the indel gets the most penalty. While the local alignment would simply be $\begin{matrix}\texttt{A}\\\texttt{A}\end{matrix}$

3.1 Prefix and suffix alignment

Consider a related different problem: **prefix alignment**: find the highest-scoring alignment between two prefixes of the two sequences. We want to find i, j to maximize alignment-score$(S[i..i], T[1..j])$.

Similarly, consider **suffix alignment**. That is, we choose two suffixes, and align them together optimally. We want to compute $\max_{i',j'}$ score$(S[i'..m], T[j'..n])$

Let $D[i, j]$ denote the optimal "suffix alignment" alignment score of $S[1..i]$, $T[1..j]$. That is, $D[i, j]$ is the maximum alignment score for $S[i'..i]$ and $T[j'..j]$ for all i' and j'. Consider the last column of this optimal "suffix" alignment. Four cases arise:

1. $S[i]$ v.s. $T[j]$
2. $S[i]$ v.s. -
3. $T[j]$ v.s. -
4. an empty alignment

Case 4 is the only new case comparing to the basic alignment. Then the DP algorithm's recurrence relation would be

$$D[i,j] = \max \begin{cases} D[i-1, j-1] + f(S[i], T[j]); \\ D[i-1, j] + f(S[i], \text{-}); \\ D[i, j-1] + f(\text{-}, T[j]); \\ 0 \end{cases}$$

Then the $D[m, n]$ will be the last cell in the DP table: optimal suffix alignment between S and T. Previously, initial cases might have negative scores. Due to the new rule here, we just put zeros.

Consider a suffix alignment example where match = 1, mismatch = indel = -1.

		C	A	T	T	C
	0	0	0	0	0	0
A	0	0	1	0	0	0
T	0	0	0	2	1	0
T	0	0	0	1	3	0
G	0	0	0	0	2	2
A	0	0	0	0	1	1

This gives alignment $\begin{matrix} & \texttt{ATTGA}\\ \texttt{C} & \texttt{ATTC-}\end{matrix}$

3.2 Local Alignment

Recall that for suffix alignment, $D[i, j]$ denote the optimal "suffix alignment" alignment score of $S[1..i], T[1..j]$. I.e., $D[i, j]$ is the maximum alignment score for $S[i'..i]$ and $T[j'..j]$ for all i' and j'. Therefore, optimal local alignment score is just $\max_{i,j} Di, j$. The algorithm will be straightforward:

Algorithm 3: Local alignment

1 Fill the dynamic programming table is the same as suffix alignment.
2 Find (i, j) to maximize $D[i, j]$, and backtrack from there.

For example,

		C	A	T	T	C
	0	0	0	0	0	0
A	0	0	1	0	0	0
T	0	0	0	2	1	0
T	0	0	0	1	(3)	0
G	0	0	0	0	2	2
A	0	0	0	0	1	1

Then the local optimal alignment is the optimal suffix alignment of $T[1..4]$ and $S[1..3]$.

The algorithm was first proposed by Temple Smith and Michael Waterman in 1981. It works for both linear and affined gap penalty. It is known popularly as the Smith-Waterman algorithm. The global alignment algorithm was called the Needleman-Wunsch algorithm, which was published in 1970.

Time complexity is quadratic. Space complexity is $O(mn)$. If we only want the score, we can just use a max variable, which takes $O(1)$ space. If we want the end positions, namely i, j, we can still introduce extra two variables $\max I, \max J$.

If we want the start positions, namely i', j', it would be a bit harder. Recall in suffix alignment, there are four cases. For case 4, we have $i' = i + 1, j' = j + 1$. For other three cases, we simply copy i', j' from the smaller alignment (sub-alignment) because they are same. So we just introduce two variables $\max I', \max J'$.

Similarly, we can do affine gap local alignment:

$$D_0[i,j] = f(S[i],T[j]) + \max \begin{cases} D_0[i-1,j-1]; \\ D_1[i-1,j-1]; \\ D_2[i-1,j-1]; \\ 0 \end{cases}$$

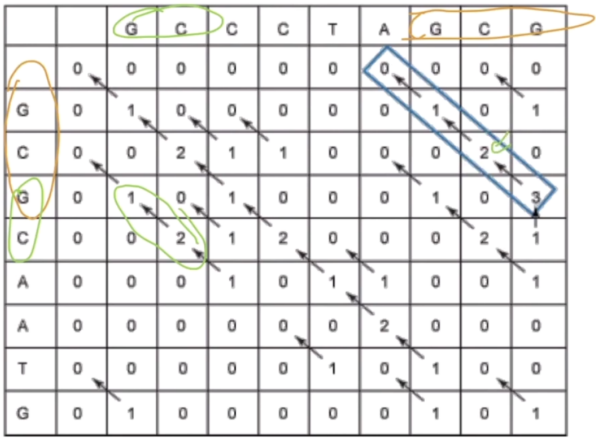
$$D_1[i,j] = \text{gapext} + \max \begin{cases} D_0[i,j-1] + \text{gapopen}; \\ D_1[i,j-1]; \\ D_2[i,j-1] + \text{gapopen}; \\ 0 \end{cases}$$

$$D_2[i,j] = \text{gapext} + \max \begin{cases} D_0[i-1,j] + \text{gapopen}; \\ D_1[i-1,j] + \text{gapopen}; \\ D_2[i-1,j]; \\ 0 \end{cases}$$

Algorithm is as before, except that score is now lower bounded by 0. Afterward, find maximum element in all 3 tables, and backtrack until reaching a 0.

3.3 Many local alignments

It's sometimes useful to find many local alignments of S and T. For example, when there are multiple similar regions between the two input strings. We can let the algorithm output multiple alignments.



		G	C	C	C	T	A	G	C	G
	0	0	0	0	0	0	0	0	0	0
G	0	1	0	0	0	0	0	1	0	1
C	0	0	2	1	1	0	0	0	2	0
G	0	1	0	1	0	0	0	1	0	3
C	0	0	2	1	2	0	0	0	2	1
A	0	0	0	1	0	1	1	0	0	1
A	0	0	0	0	0	0	2	0	0	0
T	0	0	0	0	0	1	0	1	0	0
G	0	1	0	0	0	0	0	1	0	1

3.4 Fit Alignment

There are some scenarios where local alignment is not best model. Given sequences S and T. Find a global alignment between S and a substring of T, maximizing the alignment score. We are trying to fit S into T. Deleting the prefix of T is free, deleting the suffix of T is free.

We can use similar idea as before. The DP table aligns T horizontally, S vertically. We can start from anywhere from T, so all zeros. Then for S, we need to initialize properly, $-1, -2, \ldots$. For local alignment, we can stop at any i, j. For fit alignment, we can stop at any j but not any i: we need to find the max value in the last row.

3.5 Linear Space Alignment

Why linear space? Computer RAM used to be very expensive in 80s. There was a prediction " The cost for 128 kilobytes of memory will fall below 100 bucks in the near future" Creative Computing magazine. December 1981, page 6. Even today, keeping everything in the L2 cache may speed up the computation. We have learned the linear space if only alignment score, instead of the alignment, is required. Let's now develop a linear space alignment. We focus on **global** alignment model first.

The idea is to use **divide and conquer**. We want to find j such that the optimal alignment between S and T consists of two parts:

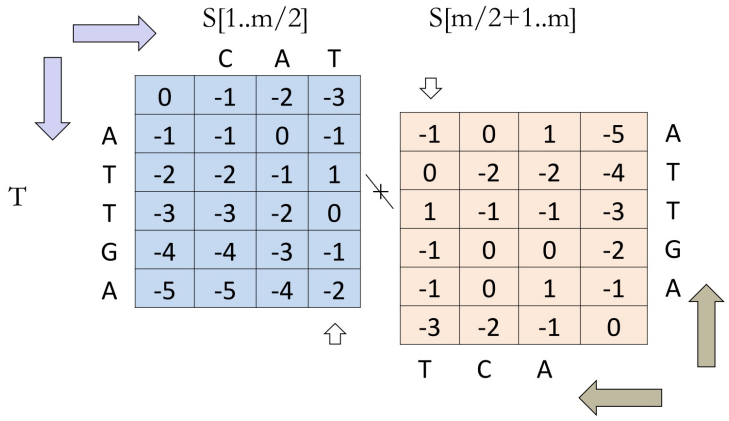
- $S[1..m/2]$ aligns with $T[1..j]$
- $S[m/2+1..m]$ aligns with $T[j+1..n]$

Then we can use divide and conquer. However, we need to compute j in linear space. Note that there may be more than one j satisfying the condition. Any one of them will do the job.

Claim j satisfies the desired condition iff it maximizes

$$\text{alignScore}(S[1..m/2], T[1..j]) + \text{alignScore}(S[m/2+1..m], T[j+1, n])$$

Then we can loop for all j. Let D be the dynamic programming table for aligning S and T. Note that $\text{alignScore}(S[1..m/2], T[1..j])$ is stored in the middle column $(m/2)$ of the table, which takes linear space only. For $\text{alignScore}(S[m/2+1..m], T[j+1, n])$, we can reverse the alignment, which doesn't affect the score. So we can fill DP table backward.



Computing the two center columns requires linear space. Then the algorithm is as follows:

Algorithm 4: Linear Space Alignment

```
1 Align(S, T):
2     if |S| = 1 then
3         return a trivial alignment
4     Use the previous idea to find j that maximizes
        alignScore(S[1..m/2], T[1..j]) + alignScore(S[m/2..m], T[j+1..n])
5     return Concatenation of Align(S[1..m/2], T[1..j]) and Align(S[m/2..m], T[j+1..n])
```

To see its time complexity, we can either use induction or do in in a sloppy way. We know that

$$T(m, n) \leq mn + T(m/2, j) + T(m/2, n-j)$$

Then if we expand the subproblems, the time complexity becomes

$$\begin{aligned} T(m,n) &\leq mn + T(m/2, j) + T(m/2, n-j) \\ &\leq mn + \frac{mn}{2} + \frac{mn}{4} + \cdots \\ &\leq 2mn \end{aligned}$$

But we have linear space complexity.

Related papers:

- *A linear space algorithm for computing longest common subsequences* by D.S. Hirschberg.
- *Optimal alignments in linear space*.

  > "The goal of this paper is to give Hirschberg's idea the visibility it deserves by developing a linear-space version of Gotoh's algorithm."

How to do local alignment in linear space? Recall the trick to find the boundaries of optimal local alignment in linear space. We can use this trick, which takes linear space, to find i, i', j, j'. Then take these four numbers, and then call global linear alignment to get the actual alignment. This is called reduction. Similarly, we can do affined gap penalty in linear space.

4

Score and Significance

Consider the sequence from https://www.ncbi.nlm.nih.gov/protein/6WPT_D:

```
>pdb|6WPT|D Chain D, S309 neutralizing antibody heavy chain
QVQLVQSGAEVKKPGASVKVSCKASGYPFTSYGISWVRQAPGQGLEWMGWISTYNGNTNYAQKFQGRVTM
TTDTSTTTGYMELRRLRSDDTAVYYCARDYTRGAWFGESLIGGFDNWGQGTLVTVSS
```

This is the heavy chain sequence neutralizing antibody in the COVID. If we search this sequence in BLAST, we can find other antibody which has similar sequences, in terms of the scores.

Most optimization problems are of the format:

- **Instance**: describes the input
- **Feasible** Solution: describes the format of the output
- **Score Function**: measures how good a solution is
- **Objective**: either maximize or minimize the score.

For example, in the context of sequence alignment, we have

- Instance: two sequences S and T
- Feasible Solution: insert gaps into S and T so that they have the same length
- Score Function: $\sum_{\text{col}_i} \text{score}(i)$
- Objective: to maximize the score

Consider three purposes of the score function.

- Purpose 1: It helps us to compare solutions of the same instance.

  Which alignment is the best for the same input (s, t).

- Purpose 2: It helps us to compare solutions of different instances.

  Which of (s, t) and (x, y) is more likely to be a homology?

- Purpose 3: It helps us to tell how significant the solution is.

  Does the alignment between s and t indicate that they are homologous? Or if we tell our friend our GPA is 4.0, and they know what that means.

4.1 Purpose 1 of score function

We first examine how the scoring function is designed for the first purpose - compare two alignments and tell which one is better. Recall that what we really want is to find out homologies. Consider two sequences `ATGCATGTA` and `ATGTACTGA`. We want to find their evolution path. Here first `A` doesn't change, while fourth `C` and `T` are different. The better alignment would reflect a higher probability that this alignment happened in the revolution history.

Now considered a simple (oversimplified) evolutionary model. We assume

- evolution only contains substitution and indel.
- two mutations do not overlap. For example, we do not consider the possibility that $A \to C \to A$. This guarantees that all evolutionary information but the order is represented by the alignment.
- different columns are independent to each other.

Along the path of evolution, we denote the probability: p unchanged, q substitution, r indel and $p + q + r = 1$.

Consider the alignment

```
ATGCA-TGTA      (S)
|||  |  ||  |
ATGTACTG-A      (T)
```

we can see that under this model, the probability of the alignment is $p^7 \cdot q \cdot r^2$. It's not convenient to do multiplication in the program, then we convert it into summation. Thus we want to maximize this probability

$$\log(p^7 \cdot q \cdot r^2) = 7 \log p + \log q + 2 \log r$$

Let match $= \log p$, mismatch $= \log q$, indel $= \log r$. We get a scoring scheme. Then maximizing the score is equivalent to maximizing the probability of evolutionary history.

This is sufficient to compare the alignments of the same two sequences. However, there are some problems:

- As probability is always less than 1, log is *always negative*, which is not intuitive.
- Local alignment becomes *meaningless*. The score is always negative, and the empty alignment gives 0 score, which is always the maximum.
- Repeating the alignment twice make the *score lower*. Imagine we have two strings S, T aligned perfectly well. Then consider $S' = S \mid S, T' = T \mid T$. Intuitively, S' and T' produce longer good alignment, but it has lower (more negative) score.

And above example shows that a score works well when comparing the different alignment of the same string, but might be useless when in comparison of two alignments of different pairs of sequences.

For the first problem, if we simply add a minus sign to the score, we are basically change maximization problem to minimization problem, which is the same as before, which doesn't solve the problem.

4.2 Purpose 2 of score function

Given a scheme that match = 1, mismatch = indel = -1, the motivation behinds it is that in homology, we see a lot of matches. This brings us to the idea of **likelihood ratio**.

Consider two contrast models:

- Model 1 (homology): the alignment A between S and T reflects evolutionary history.
- Model 2 (random): the alignment A between S and T is merely a random event.

If homology model gives us higher probability than random model, then we consider it as homology. We want to examine the likelihood ratio:

$$\Pr(\text{alignment} \mid \text{homology}) / \Pr(\text{alignment} \mid \text{random})$$

If it's much bigger than 1 (such as 100000), it's evidence towards model one being the truth. If it's much below 1 (such as 0.00001), it's evidence towards model two being the truth.

Assume for the homology and random models, we have established the probabilities for each column type:

- Match: p and p'
- Substitution: q and q'
- Indel: r and r'

For the alignment

```
ATGCA-TGTA      (S)
|||  |  ||  |
ATGTACTG-A      (T)
```

we have

$$\Pr(\text{alignment} \mid \text{homology}) / \Pr(\text{alignment} \mid \text{random}) = (p/p')^7 \cdot (q/q') \cdot (r/r')^2$$

We usually take a logarithm. The sore becomes

$$7\log(p/p') + \log(q/q') + 2\log(r/r')$$

If this is very positive, then homology model explains the alignment better than random model. And vice versa.

It turns out that this is a better scoring scheme. It prefers column types happens more often in the homology model than in the random model. Usually,

- $p > p'$, therefore a matching column has a positive score
- $q < q'$, therefore a mismatching column has a negative score
- $r < r'$, therefore an indel column has a negative score.

Thus, $\log(p/p') > 0, \log(q/q') < 0$. This avoids the problems we had when only probabilities (not the ratio) were used. In the previous "failed" case, if we put two positive alignments together, we increase the homology chance. Moreover, it can be not only used to compare the alignments of the same input, but also compare the alignments of different inputs, or different lengths. This is indeed the scoring scheme we have seen and have used in practice (in BLAST etc.)

The probability values used in the homology and random models may be obtained by simple counting their frequencies in some "real" alignments and "random" alignments, respectively. Often, the statistics is only approximate and does not need to be precise. In particular, the "random" model often uses some (over)- simplified values. For example: $\Pr(indel) = 0.2, \Pr(match) = 1/20 \times 0.8, \Pr(mismatch) = 19/20 \times 0.8$.

We can still refine this statistics model. Some substitutions are between letters that have similar properties, which then happen more often. The non-indel columns can be further refined to have different scores for different pairs of letters.

For each pair of letters a and b, assume the probability of seeing (a, b) in a column is

$$p(a, b) = \Pr(a, b \mid \text{homology})$$

for the homology model, and is

$$q(a, b) = \Pr(a, b \mid \text{random})$$

for the random model. Then substitution score is then $\log(p(a, b)/q(a, b))$. This is called a substitution matrix.

Let us assume that $q(a, b) = p(a)p(b)$, i.e., independent and random. Here $p(a)$ is the frequency of letter a in the sequences. Note that this is an (over)-simplification. But it provides "good enough" values in practice.

The substitution matrix is particularly important when aligning protein sequences because there are 20 amino acids, some of them share significant similarities and protein alignments have fewer matching columns.

```
Conserved domain database 22426:
KOG4652, HORMA domain [Chromatin structure and dynamics]

Conserved domain length = 324 residues, 100% aligned   ungapped alignment

CT46          15  VFPNKISTEHQSLVLVKRLLAVSVSCITYLRGIFPECAYGTRYLDD--DLCVKILREDKNCPG--STQLVKWMLGC
                    PN +  E QSL  + RLL V++S I   RGIFPE  +  RY+D    L + +LR      G   + L K +
KOG4652       1   TLPNGLENEKQSLEFMTRLLYVAISTILRERGIFPEEYFKDRYVDGNLLVMTLLRRQDAPEGRLVSWLEKGV---

CT46          85  YDALQKKYLRMVVLAVYTNPEDPQTISECYQFKFKYTNNGPLMDFISKN------QSNESSMLSTD-TKKASILL
                  +DA+++K L+ + L V T  EDP+ I E Y F F Y   G +   I+        ++ E S LS D T++    L
KOG4652       73  HDAIRQKLLKKLSL-VITESEDPEDI-EVYIFSFVYDEEGSVSARINYGINGQSSKAFELSQLSMDDTRRQFAKL

CT46         154  IRKIYILMQNLGPLPNDVCLTMKLFYYDEVTPPDYQPPGFKDGDCEGVIFEGEPMYLNVGEVSTPFHIFKVKVTT
                  IRK++I  Q L PLP    +     YY E  PPDYQP GFKD          P  +N+G VSTP H   VKV
KOG4652      146  IRKLHICTQLLEPLPQ-GLILSMRLYYTERVPPDYQPEGFKDSTRAFYTLPVNPEQINIGAVSTPHHKGFVKVL-

CT46         229  ERERMENIDSTILSPKQIKTPFQKILRDKDVEDEQEHYTSDDLDIETKMEEQEKNPASSELEEPSLVCEEDEIMR
                                                  SD  D   K E
KOG4652      219  ----------------------------------------SDATDSMEKAER----------------------T

CT46         304  SKESPDLSISHSQVEQLVNKTSELDMSESKTRSGKVFQNKMANGNQPVKSSKENRKRSQHESGR---IVLHHFDS
                   K S D       V+Q +NK+ E D S S+ ++  +  N + N   PV  S+E+  +SQ   G          D
KOG4652      232  DKISDDP-FDLILVQQELNKSEEADKSFSQEKTTSITPNVLGNPLVPVDQSEEDLLKSQDSPGTGRCSCECGLDV

CT46         376  SSQESVPKRRKFSEPKEHI
                  S Q SVPK RK     EH
KOG4652      306  SKQASVPKTRKSCRKTEHG


Homology between CT46 and MGC26710 hypothetical protein

Identities = 136/249 (54%), with conservative changes = 180/249 (72%)

CT46          1   MATAQLQR-----TPMSALVFPNKISTEHQSLVLVKRLLAVSVSCITYLRGIFPECAYGTRYLDDLCVKILREDK
                  MATAQL              VFP++I+ EH+SL +VK+L A S+SCITYLRG+FPE +YG R+LDDL +KILREDK
MGC26710      1   MATAQLSHCITIHKASKETVFPSQITNEHESLKMVKKLFATSISCITYLRGLFPESSYGERHLDDLSLKILREDK

CT46         71   NCPGSTQLVKWMLGCYDALQKKYLRMVVLAVYTNPEDPQTISECYQFKFKYTNNGPLMDF--ISKNQSNESSMLS
                   CPGS  +++W+ GC+DAL+K+YLRM VL +YT+P   + ++E YQFKFKYT  G  MDF   S + S ES   +
MGC26710     76   KCPGSLHIIRWIQGCFDALEKRYLRMAVLTLYTDPMGSEKVTEMYQFKFKYTKEGATMDFDSHSSSTSFESGTNN

CT46         144  TDTKKASILLIRKIYILMQNLGPLPNDVCLTMKLFYYDEVTPPDYQPPGFKDG-DCEGVIFEGEPMYLNVGEVST
                   D KKAS+LLIRK+YILMQ+L PLPN+V LTMKL YY+ VTP DYQP GFK+G +   ++F+ EP+ + VG VST
MGC26710     151  EDIKKASVLLIRKLYILMQDLEPLPNNVVLTMKLHYYNAVTPHDYQPLGFKEGVNSHFLLFDKEPINVQVGFVST

CT46         218  PFHIFKVKVTTERERMENIDSTIL  241
                   FH  KVKV TE  ++ ++++ +
MGC26710     226  GFHSMKVKVMTEATKVIDLENNLF  249
```

The first gap is circled in red. Anything before the gap is called *ungapped alignment*. For the purpose of this course, we call it a block. Blocks do not have indel.

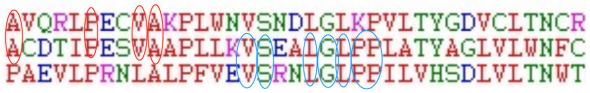
	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V	B	Z	X	*
A	4	-1	-2	-2	0	-1	-1	0	-2	-1	-1	-1	-1	-2	-1	1	0	-3	-2	0	-2	-1	0	-4
R	-1	5	0	-2	-3	1	0	-2	0	-3	-2	2	-1	-3	-2	-1	-1	-3	-2	-3	-1	0	-1	-4
N	-2	0	6	1	-3	0	0	0	1	-3	-3	0	-2	-3	-2	1	0	-4	-2	-3	3	0	-1	-4
D	-2	-2	1	6	-3	0	2	-1	-1	-3	-4	-1	-3	-3	-1	0	-1	-4	-3	-3	4	1	-1	-4
C	0	-3	-3	-3	9	-3	-4	-3	-3	-1	-1	-3	-1	-2	-3	-1	-1	-2	-2	-1	-3	-3	-2	-4
Q	-1	1	0	0	-3	5	2	-2	0	-3	-2	1	0	-3	-1	0	-1	-2	-1	-2	0	3	-1	-4
E	-1	0	0	2	-4	2	5	-2	0	-3	-3	1	-2	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
G	0	-2	0	-1	-3	-2	-2	6	-2	-4	-4	-2	-3	-3	-2	0	-2	-2	-3	-3	-1	-2	-1	-4
H	-2	0	1	-1	-3	0	0	-2	8	-3	-3	-1	-2	-1	-2	-1	-2	-2	2	-3	0	0	-1	-4
I	-1	-3	-3	-3	-1	-3	-3	-4	-3	4	2	-3	1	0	-3	-2	-1	-3	-1	3	-3	-3	-1	-4
L	-1	-2	-3	-4	-1	-2	-3	-4	-3	2	4	-2	2	0	-3	-2	-1	-2	-1	1	-4	-3	-1	-4
K	-1	2	0	-1	-3	1	1	-2	-1	-3	-2	5	-1	-3	-1	0	-1	-3	-2	-2	0	1	-1	-4
M	-1	-1	-2	-3	-1	0	-2	-3	-2	1	2	-1	5	0	-2	-1	-1	-1	-1	1	-3	-1	-1	-4
F	-2	-3	-3	-3	-2	-3	-3	-3	-1	0	0	-3	0	6	-4	-2	-2	1	3	-1	-3	-3	-1	-4
P	-1	-2	-2	-1	-3	-1	-1	-2	-2	-3	-3	-1	-2	-4	7	-1	-1	-4	-3	-2	-2	-1	-2	-4
S	1	-1	1	0	-1	0	0	0	-1	-2	-2	0	-1	-2	-1	4	1	-3	-2	-2	0	0	0	-4
T	0	-1	0	-1	-1	-1	-1	-2	-2	-1	-1	-1	-1	-2	-1	1	5	-2	-2	0	-1	-1	0	-4
W	-3	-3	-4	-4	-2	-2	-3	-2	-2	-3	-2	-3	-1	1	-4	-3	-2	11	2	-3	-4	-3	-2	-4
Y	-2	-2	-2	-3	-2	-1	-2	-3	2	-1	-1	-2	-1	3	-3	-2	-2	2	7	-1	-3	-2	-1	-4
V	0	-3	-3	-3	-1	-2	-2	-3	-3	3	1	-2	1	-1	-2	-2	0	-3	-1	4	-3	-2	-1	-4
B	-2	-1	3	4	-3	0	1	-1	0	-3	-4	0	-3	-3	-2	0	-1	-4	-3	-3	4	1	-1	-4
Z	-1	0	0	1	-3	3	4	-2	0	-3	-3	1	-1	-3	-1	0	-1	-3	-2	-2	1	4	-1	-4
X	0	-1	-1	-1	-2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-2	0	0	-2	-1	-1	-1	-1	-1	-4
*	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	-4	1

Consider a substitution matrix shown above: BLOSUM 62. "BLO" stands for block, "SU" for substitution, "M" for matrix. It is the most used amino acid substitution matrix. Note that `B, Z, X, *` are non-standard letters:

1. `B = D or N`
2. `Z = E or Q`
3. `X = any`
4. `* = translation stop`

We can see the red cell in the matrix is also positive even row and col indices are different. Let's study how this is constructed.

Now we want to find out the probability that `A` and `B` appears in the same column of the real homology alignment. We first need to find many real homology examples. Henikoff's published two papers: *Automated assembly of protein blocks for database searching*, *Amino acid substitution matrices from protein blocks*. If we see ungapped alignment between two sequences that have lots of matches, then we consider it as *conserved regions*, then we assume it is real homology.



The identity level is high therefore we know they are homologous without a score matrix. In the two papers mentioned above, there was such a database consisting those blocks.

In the "block" above, there are 37 columns, and each column 3 pairs. Thus in total 111 pairs. For example, the pair `I-L` occurs 3 times; the pair `L-L` occurs 13 times, then

$$P_{IL} = \frac{3}{111}, P_{LL} = \frac{13}{111}$$

There are 111 amino acid in total [1]. There are total 2 I's and 21 L's, thus

$$P_I = \frac{2}{111}, P_L = \frac{21}{111}$$

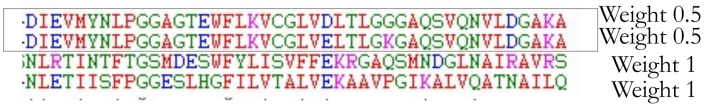
BLOSUM's formula for score are

$$\text{score}(x, y) = 2\log_2 \frac{P_{xy}}{2P_x P_y}, \text{ if } x \neq y$$

$$\text{score}(x, x) = 2\log_2 \frac{P_{xx}}{P_x P_x}$$

In BLOSUM matrices these values are rounded to the nearest integer. The multiplication by 2 at the front provides tiny benefit when rounding to the nearest integer: the absolute round error gets halved, which means the rounding is more accurate.

There's problem with the database. Some blocks might be big, while some might be small. Moreover, there's sample bias: some protein families are more well studied so they are overrepresented in the database. Such bias is caused by the studies, not reflecting what's going on during evolution. To remove this bias in statistics, those "redundant" proteins are classified together before BLOSUM calculation.



Consider the first two sequences in the picture. We can see they are overly similar. The sequences that are 62% or above similarity are grouped together and given total weight 1. This way, the `AA` pairs are counted between groups that are 62% similar or below. The lower this number is, the better is the matrix suitable to distant homology search. The original BLOSUM paper found out 62 is best at the time the paper was prepared.

4.3 Purpose 3 of score function

Consider that BLAST matches a query sequence with all database sequences, and return the highest scoring local alignments. The best local alignment score is 100. Does it mean good or bad? The answer depends on the score scheme you use so we need to standardize it for effective communication. Intuitively, we would ask "can this happen randomly"? This is the intuition for the significance. This is formalized as the p-value.

> *p-value*
>
> Imagine we want to prove that a coin is biased on its two sides. We first create *null hypothesis* H_0: the coin is fair. Or *alternative hypothesis*: the coin is biased, which is what we try to prove. Then we conduct the experiment: draw the coin 20 times, and found 14 heads and 6 tails.
>
> $$\Pr(\#\text{ heads } \geq 14 \text{ or } \#\text{ tail } \geq 14 \mid H_0) = 0.1154$$
>
> This is called the p-value.
>
> In statistical hypothesis testing, the p-value is the probability of obtaining a result at least as extreme as the one that was actually observed, given that the null hypothesis is true. A small p-value **rejects** the null hypothesis. So, we choose to believe the alternative hypothesis.
>
> In the coin example, 0.1154 is certainly not good enough. But what if we draw 40 times, and found 28 heads and 12 tails? Now p-value is 0.0115. This is how scientists show the effectiveness of a treatment through clinical trials. (null hypothesis: it is not effective).

[1] a this is coincident, not the same as 111 pairs before. It depends of number of sequences in each block.

> A small p-value *rejects* the null hypothesis. So, we choose to *believe* the alternative hypothesis. This does not mean that the alternative hypothesis is surely correct. It is just the null hypothesis is unlikely to be correct. It is just a way to communicate the significance. The union of null hypothesis and alternative hypothesis is not the whole universe. For example, if the two sequences are irrelevant, then we should not have seen the alignment with such a high score. But they may be related for reasons other than homology - e.g. convergence evolution.
>
> In practice an arbitrary threshold 0.05 is often used, for no apparent reason.

Now we can use p-value for local alignment to communicate its significance. Null hypothesis: *the query S and the database T are irrelevant*. Now we observe a high scoring local alignment with score x. This is an extreme case. Then the p-value is $\Pr(\text{score} \geq x \mid H_0)$. How to compute the p-value. From now on, S and T are long sequences.

Statistics of Ungapped Local Alignment

First bioinformatics studied statistics form ungapped local alignment. Without indel, things become easier. For ungapped alignment (no indel), each local ungapped alignment is called an **HSP** (High Scoring (Segment) Pair) in the BLAST program. For an individual HSP, the alignment score is the sum of n identical independent variables. So the score is a binomial distribution. As $|S| = n, |T| = m$ are quite long, there are about $O(mn)$ HSPs. As we have a lot of such ungapped alignments, and we only output the best one (with the highest alignment score). For p-value, we need to check how *rare* this best alignment has a high score. The maxima of many identical independent variables is a so-called extreme value distribution.

There are two papers in 90s studying the distribution of ungapped local alignment (HSPs) among two big sequences via both E-values and P-values: "Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes" and "Limit distribution of maximal nonaligned two-sequence segmental score".

Theorem 4.1: E-value

In the limit of sufficiently large sequence lengths m and n, the statistics of HSP scores are characterized by two parameters, K and λ. Most simply, the expected number of HSPs with score at least x is given by the formula

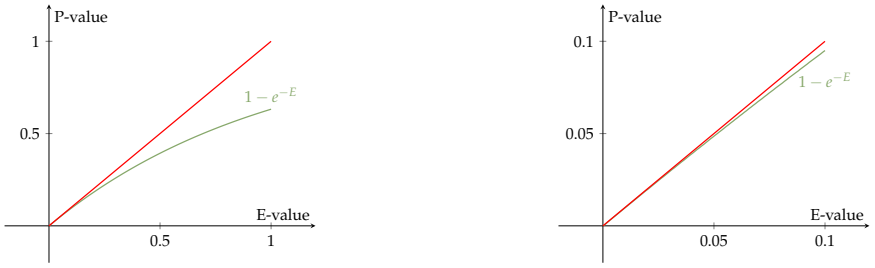
$$E = Kmne^{\lambda x}$$

This is called the E-value of the alignment. The parameters K and λ can be somehow determined from the scoring matrix. (Details not given here). Thus, we just convert the alignment score x to E-value. If it is very small, then random sequences will not often produce HSPs with score $\geq x$. This is likely caused by homology. There are not many false positives if we treat every local alignment with score $\geq x$ as homologies.

Theorem 4.2: P-value

The number of random HSPs with score $\geq S$ is described by a Poisson distribution. Therefore the chance of finding zero HSP with score $\geq S$ is e^{-E}, where E is the E-value of score S.

When E is small, then $e^{-E} \to 1$, and probability of finding no HSP with that score is high. Therefore, the p-value is $1 - e^{-E}$. When E-value is very small, P-value and E-value are almost identical. BLAST chose to use E-value. Below are comparisons of two values.



Green curve is p-value, which is equal to $1 - e^{-E}$. We can see from the diagram on the right: when E-value is between 0 and 0.1, the green curve matches the red curve nicely. All we care are the cases where both are small. When both are big, it's hard to see its homology.

The E-value and P-value have their physical meanings. For example, if S and T are random, we expect to see on average 0.01 HSP with such a high score. The alignment score, however, largely depends on the scoring matrix one chose. Saying that "the alignment score is 100" is like saying "the length is 100". There are *no* units for this measurement.

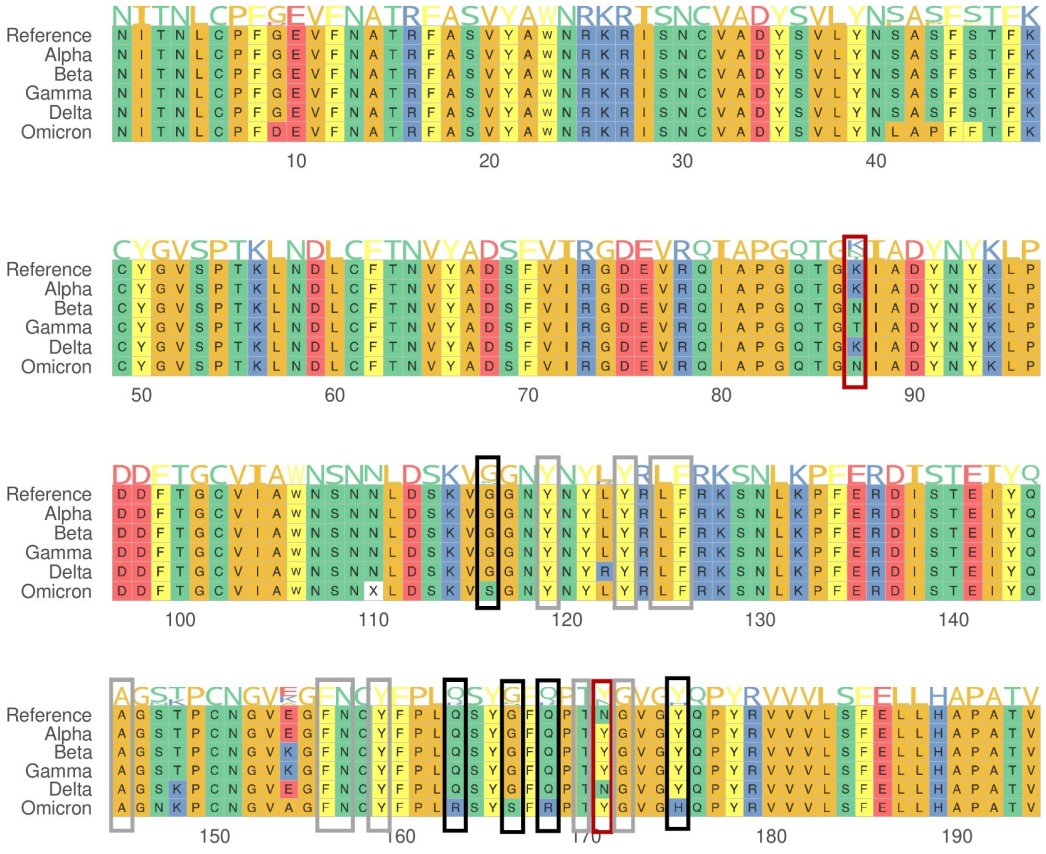
The statistics of gapped alignments The statistics developed above have a solid theoretical foundation only for local alignments that are not permitted to have gaps. However, many computational experiments and some analytic results strongly suggest that the same theory applies as well to gapped alignments. But we need to know how to compute K and λ. We don't know how to compute/estimate them mathematically. Thus, we need to do some simulation like repeating random alignments.

5

Multiple Sequence Alignment

Consider an example of multiple sequence alignment.

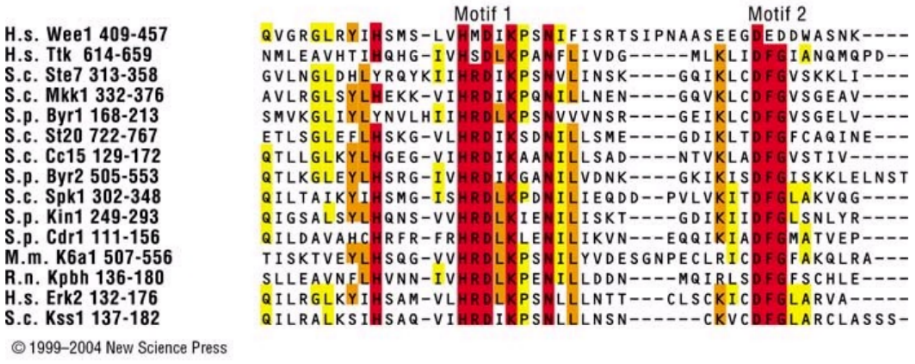
Fig.3



From this paper:

> *Contacting residues between SARS-CoV-2 and ACE2.* Boxes denote the contacting residues. Black boxes denote mutations unique to omicron, red boxes denote mutations occurring in multiple variants, and grey boxes denote no mutations in any variant.

If we consider indels, alignment becomes harder. A multiple sequence alignment of k sequences is an insertion of gaps in the positions of the sequences, just like a pairwise alignment.



Red ones and yellow ones are almost the same. Arthur M. Lesk once said:

> "Two homologous sequences whisper, a multiple alignment shouts loudly."

5.1 Heuristic Algorithm for Multiple Alignment

There is a simple algorithm that can merge pairwise alignments to a multiple alignment. The algorithm does not guarantee the optimality of the result. But runs relatively fast.

Suppose we have the following two pairwise alignments:

- t: `A-GAGC`
- s_1: `ATGAGC`
- and
- t: `A-GAGC`
- s_2: `AGTTGC`

Our main idea is to use the shared sequence t to construct a multiple alignment of s_1, t, and s_2.

```
 t: A-GAGC              t: A-GA-GC
s1: ATGAGC      ⟹     s1: ATGA-GC      ⟹      t: A-GA-GC
    and                    and                s1: ATGA-GC
 t: AGA-GC              t: A-GA-GC             s2: A-GTTGC
s2: AGTTGC             s2: A-GTTGC
```

We observe that pairwise alignment is "induced" by the multiple alignment.

Algorithm Insert gaps to the two alignments, so that the superstrings for t become the same in the two alignments. Then put the two alignments together. We keep two pointers i and j for t in both pairs.

Property Maintains the pairwise alignment unchanged if ignoring the all-gap columns of the pairwise alignment.

The same idea can be applied to merging two multiple alignments as well. For two multiple alignments, as long as they share a common string, we can apply the same idea.

```
 t: A-GA-GC              t: A-GA-GC
s1: ATGA-GC             s1: ATGA-GC              t: A-GA-GC
s2: A-GTTGC      ⟹      s2: A-GTTGC      ⟹     s1: ATGA-GC
                                                 s2: A-GTTGC
 t: AGAGC                t: A-GA-GC              s3: A-TA--C
s3: ATA-C               s3: A-TA--C
```

The process can be continued to merge two multiple alignments together as a bigger one. If we always use t is the common string, the alignment between other sequences and t have not changed over the iterations. Note that the obtained multiple alignment may not be optimal. In the example above, if we only consider the pair alignment between s_1 and s_3, we should only open the gap between the second `T` and third `A`.

Algorithm 5: Heuristic Algorithm for Multiple Alignment

Input: $s_1, \ldots, s_n$

1 $A \leftarrow$ pairwise alignment of s_1 and s_2.
2 **for** $i \leftarrow 3..n$ **do**
3 Construct pairwise alignment P between s_1 and s_i.
4 $A \leftarrow \text{merge}(A, P, s_1)$, i.e., merging A and P using s_1 as the template.
5 **return** A

The order of the merging is important to get good (but not optimal) multiple alignment. We want to merge similar sequences first, as we don't want to open gaps in the beginning, which will be carried over for future merging operations. One way is to construct a minimum spanning tree, and then merge using the shared vertices.

The exact algorithm for multiple alignment is super-polynomial. Before we study it, let's examine a heuristic algorithm. A heuristic algorithm is an algorithm that gives up quality for speed. It usually does not offer any performance guarantee in terms of quality. Well... We already sacrificed the quality because of a simple scoring function anyway. If we cannot afford exponential time, the best we can ask for is a suboptimal solution. But practically, it might work sometimes. We do not want to spend super-polynomial (e.g. exponential) time.

5.2 Exact Algorithm for Multiple Alignment

When the optimal alignment is needed. There is an exact algorithm as well. First we need to define **score function** we want to optimize.

For each column, assuming the letters are $a_1, \ldots, a_n$ for the n sequences. Define a column score $S(a_1, \ldots, a_n)$. Then the alignment score is the total of the column score. There are more than one ways to define the column score. Let us examine the simplest one first.

SP-score

SP (Sum of Pair) score is the most widely used because its simplicity. We have a similarity matrix $s(a, b)$ for any two given letters (or dashes). For the k-th column with n letters $x_1, \ldots, x_n$. Define the SP-score for the k-th column by

$$S_k = \sum_{i,j} s\left(x_i, x_j\right)$$

The SP-score of the alignment is

$$\sum_{1 \le k \le m} S_k$$

We normally require $s(\text{-},\text{-}) = 0$.

Note that SP-score can be viewed from a different perspective as follows: For each pair of sequences S_i and S_j (including the gaps) in the multiple sequence alignment, define

$$S_{i,j} = \sum_k (S_i[k], S_j[k])$$

The SP score is then $\sum_{i,j} S_{i,j}$. The two perspectives are equivalent if $s(\text{-},\text{-}) = 0$.

Recall the pairwise alignment algorithm, we then borrow this idea to align three sequences. Use `X` to indicate a letter. The last column has 7 cases.

X	-	X	-	X	-	X
X	X	-	-	X	X	-
X	X	X	X	-	-	-

Let f be the column-wise score function. $DP[i_1, i_2, i_3]$ is the optimal score for $s_1[1..i_1], s_2[i..i_2], s_3[1..i_3]$. $DP[i_1, i_2, i_3]$ is maximum of the 7 cases.

- case `XXX`: $DP\left[i_1, i_2, i_3\right] = DP\left[i_1 - 1, i_2 - 1, i_3 - 1\right] + S\left(s_1\left[i_1\right], s_2\left[i_2\right], s_3\left[i_3\right]\right)$
- case `XX-`: $DP\left[i_1, i_2, i_3\right] = DP\left[i_1 - 1, i_2 - 1, i_3\right] + S\left(s_1\left[i_1\right], s_2\left[i_2\right], \text{-}\right)$
- ...

It is necessary to introduce some notation for n sequences...

Here `X` indicates a letter and a dash indicates an indel. If you regard `X` as 1 and - as 0. Then these cases are all the 3-bit binary numbers but `000`. In general, for n sequences, there are $2^n - 1$ cases. Let us use δ_j to indicate whether at sequence j the last column is a letter. $\delta_j = 1$ if it is a letter; $\delta_j = 0$ if it is a dash.

Algorithm 6: Algorithm for Optimal Multiple Alignment

1 Let $S(x_1, \ldots, x_n)$ be a function to compute the column score.
2 Let $D[i_1, \ldots, i_n]$ be the optimal multiple alignment score of $s_1[1..i_1], \ldots, s_n[1..i_n]$.
3 Consider the last column of the optimal multiple alignment. There are $2^n - 1$ cases.
4 For each case, the "sub-alignment" after removing the last column is also optimal.
5 Then

$$D\left[i_1, i_2, \ldots, i_n\right] = \max_{\delta_j = 0,1} \left(D\left[i_1 - \delta_1, \ldots, i_n - \delta_n\right] + S\left(b_1, \ldots, b_n\right)\right)$$

where

$$b_j = \begin{cases} \text{-}, & \text{if } \delta_j = 0 \\ s_j\left[i_j\right], & \text{if } \delta_j = 1 \end{cases}$$

We see that line 5 has a nested loop m^n iterations. For each recurrence relation, we need to maximize for every possible δ_j, 2^n possible cases. Thus the time complexity is $O(m^n 2^n T)$, where T is the time needed for computing a column score. Space complexity is $O(m^n)$.

This problem is NP-hard. So, no polynomial time algorithm exists (unless $P = NP$). Computing the optimal alignment with large m and n is hopeless. But at least this works for small m and n.

SP score (either maximization or minimization) is one of the simplest scoring function in use. There are better proposals to the scoring function. Let's examine a more sophisticated score called relative entropy. We want to maximize.

Relative Entropy

There are n letters $x_1, \ldots, x_n$ in a column. For letter a in alphabet, let n_a be the number of a in the n letters. Our two different models assume two different distributions of letters in that column.

- Model 1: (homology model) a occurs with probability $p_a = n_a/n$.
- Model 2: (random model) a occurs with "background" probability q_a, which is learned from a database.

We want to compute the likelihood ratio. Then we do log likelihood ratio as usual. First find the conditional probabilities,

$$\Pr(x_1, \ldots, x_n \mid \text{model } 1) = \prod_{1 \le i \le n} p_{x_i}$$
$$\Pr(x_1, \ldots, x_n \mid \text{model } 2) = \prod_{1 \le i \le n} q_{x_i}$$

Then log likelihood ratio is

$$\log \prod_{1 \le i \le n} \frac{p_{x_i}}{q_{x_i}} = \sum_{1 \le i \le n} \log \frac{p_{x_i}}{q_{x_i}} = \sum_{a \in \Sigma} n_a \log \frac{p_a}{q_a}$$

This is called the relative entropy at a column. Let E_j be the relative entropy of column j. Then the alignment score is equal to $\sum_{1 \le j \le m} E_j$, which we want to maximize. Note that $E_j \ge 0$, and $E_j = 0$ only if $p_a = q_a$ (This is because of a theorem in information theory).

A few assumptions of relative entropy score:

- Sequences are independent to each other.
- Columns are independent to each other.

These seem to be too strong but at least when these assumptions are true the score has a theory foundation. Other scores are more empirical. Note: If all q_a are equal to each other, then this is equivalent to the "minimum entropy" introduced in the reference book (Durbin page 138).

5.3 Approximation Algorithm

In the heuristic algorithm, it would be good if we can get some guarantee on the quality of the suboptimal result. Suppose we want to maximize a score, and the optimal value is OPT, how about computing a solution with score at least OPT/c for a small constant $c > 1$? If $c = 1.001$, this is not so bad. If this can be done, this is called a **ratio-c approximation algorithm** for the problem. For minimization problem, ratio c means the algorithm gives score at most $c \cdot OPT$.

The known approximation algorithm for multiple alignment only works for the minimization version of the problem. Let $M[a, b]$ be a distance metric between two letters (possibly -) a and b satisfying **triangular inequality**: $M[a, a] = 0, M[a, b] \ge 0, M[a, c] \le M[a, b] + M[b, c]$.

The distance between two sequences $d'(S_i, S_j)$ is the total of their column scores using M as the score scheme. The SP score is $\sum_{i,j} d'(S_i, S_j)$. We want to minimize. Note that this is almost the same as before, except that now we want to minimize the difference.

For sequences S, T, R taken from the same multiple alignment, we have

$$d'(S, R) \le d'(S, T) + d'(T, R)$$

Note that S, T, R may have the '-' symbols in the sequence. Here are some notations:

- When context is clear (the multiple alignment is given), we also use $d'(s,t)$ to denote the distance between the original sequences s and t (without '-'). It is equal to $d'(S,T)$.
- When there are more than one multiple alignments $A_1, A_2, \ldots, A_n$, we use $d'_i(s,t)$ to indicate the distance induced by A_i.
- We use $d(s,t)$ to denote the optimal pairwise alignment between s and t. Thus $d(s,t) \leq d'(s,t)$.

Algorithm 7: Approximation algorithm for multiple alignment

Input: $s_1, \ldots, s_n$

1 **foreach** $i = 1..n$ **do**
2 Build optimal pairwise alignment between s_i and s_j for each $j = 1, 2, \ldots, n$.
3 Use these pairwise alignment to build a multiple alignment A_i.
4 Output the A_i with the minimum SP score.

We want to prove that this algorithm has ratio 2.

Fact 1

$d'_i(s_i, s_j) = d(s_i, s_j)$ for every j.

Recall that in A_i, s_i is in the center. Thus the induced pairwise alignment is optimal. And $d'_i(s_k, s_\ell)$ might not be optimal if $i \neq k$.

We want an upper bound on the score produced by the algorithm:

$$d'_i(s_k, s_\ell) \leq d'_i(s_i, s_k) + d'_i(s_i, s_\ell)$$

Let A^* be the optimal multiple alignment and d^* be pairwise alignment score induced by optimal multiple alignment A^*. Then we have

$$\begin{aligned}
\sum_{i=1}^{n} SP(A_i) &= \sum_i \sum_{k,\ell} d'_i(s_k, s_\ell) \\
&\leq \sum_i \sum_{k,\ell} \left(d'_i(s_i, s_k) + d'_i(s_i, s_\ell) \right) \\
&= \sum_i \Big(\sum_{k,\ell} d'_i(s_i, s_k) + \sum_{k,\ell} d'_i(s_i, s_\ell) \Big) \\
&= \sum_i \Big(n \cdot \sum_k d'_i(s_i, s_k) + n \cdot \sum_\ell d'_i(s_i, s_\ell) \Big) \\
&= \sum_i 2n \cdot \sum_k d'_i(s_i, s_k) \\
&= \sum_i 2n \cdot \sum_k d(s_i, s_k) \\
&\leq \sum_i 2n \cdot \sum_k d^*(s_i, s_k) \\
&= 2n \cdot \Big(\sum_i \sum_k d^*(s_i, s_k) \Big) \\
&= 2n \cdot SP(A^*) \\
&= 2n \cdot OPT
\end{aligned}$$

So, at least one of the A_i has score no more than $2 \cdot OPT$.

Theorem 5.1

The above algorithm is a ratio-2 approximation to multiple alignment under SP score.

For general score scheme (general score matrix) this ratio-2 may not hold. For example, when the score scheme does not satisfy triangular inequality. Or when the score scheme has both positive and negative scores.

To get a practical solution, the algorithm might combine different ways. Clustal ω, is the most frequently used multiple sequence alignments. Moreover, multiple sequence alignment has many applications:

- **Motif finding**: conserved regions of a protein family may be important motifs.
- **Structure prediction**: after multiple alignment of a protein family, their structures can be predicted together. DeepMind, AlphaFold 2.
- **Phylogeny**: the pairwise alignment induced by the multiple alignment can be more accurate than the optimal pairwise alignment in reality. Build multiple alignment first then do phylogeny by examining the evolution on each column.

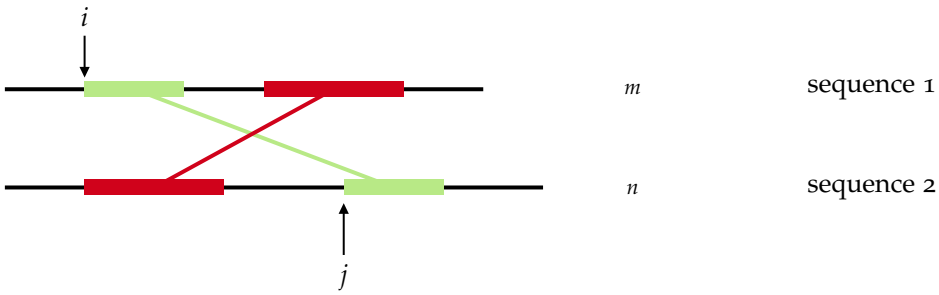
6

Seeding Methods in Homology Search

Recall that BLAST is very powerful. We will study some of their algorithms.

Consider another example. We want to compare two genomes. Smith-Waterman is the most accurate method: do dynamic programming, find high scoring entry and find good local alignment. Time complexity is $O(mn)$. So for human v.s. mouse: $3 \times 10^9 \times 3 \times 10^9 = 9 \times 10^{18}$ which will take many years.

Observe that in most cases, similarities or local alignments are not long, very short relative to the genomes.



For every pairs of (i, j), build a local alignment around it. This takes $O(mnT)$ where T is time for sub-area. This is not better than Smith-Waterman. But this leads to an important idea.

Most pairs of (i, j) are useless. We don't need to compare every i and j. Instead, We only want to try local alignments on the "promising" pairs of (i, j). In the context of sequence similarity search in bioinformatics, these "promising" pairs are called "**seeds**" or "**hits**". We then need a proper definition for hits, and • some efficient way to enumerate the hits faster than trying every pair of (i, j).

6.1 Short Consecutive Match

BLAST uses this idea as hits. The idea: find a consecutive (exact) match, and use it as hit. BLAST uses length 11 exact matches. For random i, j, the probability of exactly length k match between lengths m and n (random) sequences is $(4^{-k} \cdot mn)T$, where T is time spent on each of them. The time complexity becomes $O(4^{-k}mnT)$. By default, BLAST used $k = 11$, and it speeds up about four million.

The Idea behind Seeding

A true similarity has a high chance of being hit. Hitting at any position in the similarity will do. A random pair (i, j) has low chance of being hit. Thus, if we use hit to filter(i, j), we will detect most true similarities, and not wasting time on random pairs of (i, j).

6.2 Data Structure for Finding Hit

We will discuss a simple data structure using index table, which is different from what BLAST is using.

From wiki,

> Usually, the term k-mer refers to all of a sequence's subsequences of length , such that the sequence AGAT would have four monomers (A, G, A, and T), three 2-mers (AG, GA, AT), two 3-mers (AGA and GAT) and one 4-mer (AGAT). More generally, a sequence of length will have k-mers and total possible k-mers, where.

For each k-mer, we build an index table to remember all its occurrences in S. Then for each k-mer of T, find its hits in the index table. The index table can be a trie or a hash table. For example, given $S =$ `AATCTTAA`, we fill the index table as follows:

```
AA  ⟶  0, 6
AC
AG
AT  ⟶  1
CA
CC
CG
CT  ⟶  3
GA
GC
GG
GT
TA  ⟶  5
TC  ⟶  2
TG
TT  ⟶  4
```

We see `AA` 2-mer in index 0 and 6 of S. Then we go through 2-mer of $T =$ `GAACTTA`. Thus we found that (index) $j = 2$ (in T) will hit $i = 0, 6$ (in S).

There are 4^k entries. For each entry, it is a list of occurrences of a k-mer in S. All these lists cost $|S|$ space. The space complexity is not bad.

Building the index using S takes $O(n)$. Find matches between the index and sequence T: $O(m)$ time to scan T, plus we need to examine all of the N hits found. Let t be the examination time. Thus the overall runtime is $O(n + m + Nt)$. The term Nt is the most expensive part. Indexing overhead is small. In practice, most of the hits encountered are random hits.

We can do filtration for multiple rounds. In the previous time complexity, t in Nt will be large if the examined sequence is long. We can somehow spend extra time $t' \ll t$, to filter out random hits. Then Nt becomes $N \cdot t' + N' \cdot t$. After finding a hit, instead of trying to build a local alignment directly, BLAST uses another round of filtration to determine if a hit is a "good" or "bad" hit. Quick search in both directions; if most symbols match, it's a good hit. Otherwise it's bad. More precisely, use

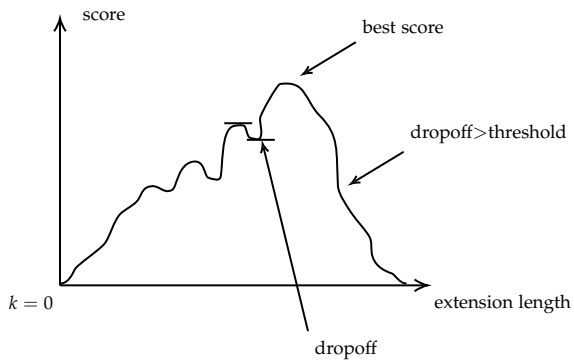
ungapped extension (linear time) to find HSPs. If an HSP is above a certain score threshold, build a local alignment around it.

6.3 HSP extension

Algorithm 8: HSP extension

```
for k ← 0.. do
    score += sc(S[i + k], T[j + k]) // go right
for k ← 1.. do
    score += sc(S[i − k], T[j − k]) // go left
```

But when to stop? Score will increase and decrease during the extension.



Extension stops when drop off greater than threshold.

How long will the extension continue after reaching best score? We assume that

- After reaching best score, sequence becomes random.
  Thus $\Pr(match) = 1/4$ and $\Pr(mismatch) = 3/4$
- match $= 1$ and mismatch $= -1$

Thus expected score on each additional base is -0.5. If $dropoff = k$, then after $2k$ bases, the expected dropoff will reach k.

This idea does not guarantee we will find every single good local alignment. BLAST will fail on the case such that the longest exact match is of length < 11, then BLAST will not find this HSP. This brings us a dilemma:

- **Sensitivity** - needs shorter seeds. the success rate of finding a homology
- **Speed** - needs longer seeds. Mega-BLAST uses seeds of length 28.

6.4 Spaced Seeds

111*1**1*1**11*111 is called a spaced seed. We use * to denote "don't care" positions. For example,

```
GAGTACTCAACACCAACATTAGTGGCAATGGAAAAT...
||  |||||||||  |||||  ||  |||||  ||||||
GAATACTCAACAGCAACACTAATGGCAGCAGAAAAT...
       111*1**1*1**11*111
```

Note that * can be match and mismatch: first * is match and second * is mismatch. Hit is all the required matches are satisfied. BLAST's seed is 11111111111.

A homology/similarity region's actual sequences do not matter, the match/mismatch matters. Therefore, a region is often denoted by a binary `0-1` sequence:

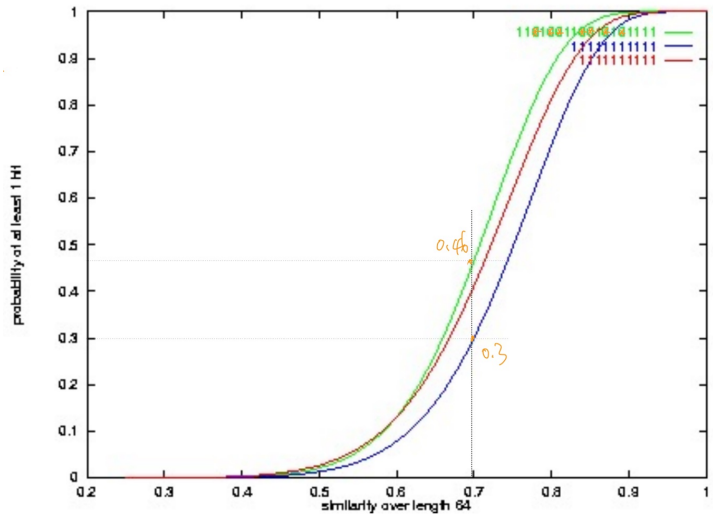
```
1101111100111011101111
```

A hit is then as follows

```
1101111100111011101111
   111*1**1*1**11*111
```

Now we change the definition of hit, we need to change data structure as well, efficiently. We build entries/indexes in the same way as consecutive seed. Except that now we have a length l, weight w seed. E.g. `11*1`. Each l-mer, take the w letters out and put in index table. Then for index `AAA`, it contains list of occurrences of `AA?A` in S. So now we take spaced k-mer, instead of consecutive k-mer. The index table can be a hash table.

The time complexity is the same: still waste the same amount of time on random hits. What about the sensitivity? Below is simulated sensitivity curves. Define HSP of length 64, and randomly assign 1 with probability p, and $p(0) = 1 - p$. x-axis is similarity level.



Why spaced seeds are better? Consider a HSP with similarity level p and a fixed position/index i, then it has match with p probability. If we use weight 11, then $\Pr(\text{seed hit pos } i) = p^{11}$. This doesn't change between spaced seed and consecutive seed. But what we care is $\Pr(\text{HSP contains at least 1 hit})$. But as spaced seeds are longer, the probability is lower. So this explanation doesn't work.

Consider

```
TTGACCTCACC?
||||||||||||?
TTGACCTCACC?
11111111111
 11111111111
```

It seems that we hit twice. However, if we hit pos 0, the chance of hitting pos 1 gets increased: we then only require last position to be matched, of probability p. So the hits between different positions within HSP are correlated. This idea works vice versa. BLAST's seed usually uses more than one hits to detect one homology, which is redundant.

```
CAA?A??A?C??TA?TGG?
|||?|??|?|??||?|||?
CAA?A??A?C??TA?TGG?
111*1**1*1**11*111
 111*1**1*1**11*111
```

For spaced seeds, hits are not strongly correlated, more independent. Spaced seeds uses fewer hits to detect one homology, which is more efficient. We see that PatternHunter's seed do not overlap heavily when shifts:

```
111*1**1*1**11*111
 111*1**1*1**11*111
  111*1**1*1**11*111
   111*1**1*1**11*111
    111*1**1*1**11*111
     111*1**1*1**11*111
      111*1**1*1**11*111
       111*1**1*1**11*111
        ......
```

The hits at different positions are independent. After first hit, the next three shift gives us 6 matches, and so on. Thus the probability of having the second hit is $3p^6 + \cdots$. For BLAST's seed, for the second position, we only need one additional match; for the third position, we need two additional matches. Thus probability is roughly $p + p^2 + p^3 + p^4 + \cdots$, which is bigger than spaced seed.

6.5 Lossless Filtration

Filtration is bad, because we give up sensitivity: some of HSPs will be lost. When seeds are short enough and HSP similarity is high enough, lossless filtration is also possible. For example, seed 111 can guarantee to match when a sufficiently long HSP has similarity 66.7%. To fail being hit by 111, the HSP must have a mismatch in every 3 adjacent positions. On the other hand, $110110110\cdots$, which has 66.6% similarity, will fail the seed 111.

Now consider spaced seed 11*1. We claim that $\forall \epsilon > 0$, seed 11*1 will hit every sufficiently long region with similarity $0.6 + \epsilon$.

Proof:
Suppose there is a sufficiently long region not hit by 11*1. For every HSP, we can divide it into different blocks of the form 1^a0^b. Notice that $a \leq 3$, otherwise it's already hit by 11*1. We then consider different cases for a:

- $a = 3$. We know next block starts with at least one 1. So in order to have mismatch, $b \geq 2$. This has identity level $\leq \frac{3}{5}$ as it has 3 matches out of 5: $\frac{a}{a+b}$.
- $a = 2$. Similarly, $b \geq 2$. Otherwise $11011^{a'}0^{b'}$ will produce a match. Identity level $\leq \frac{2}{4}$.
- $a = 1$. We need to have $b \geq 1$ to form a block. Identity level $\leq \frac{1}{2}$.

Thus in each block, similarity ≤ 0.6. So the long region's similarity is $< 0.6 + \epsilon$. □

6.6 Compute Seed's Sensitivity

We denote a probabilistic distribution of HSP by R, and $\Pr(R[i] = 1) = p$ is uniform random distribution. We want to compute $\Pr(\text{length-}n\ R \text{ is hit by a seed } x)$, where $|x| = k$.

Denote a length-k binary string by s, and Rs to be the concatenation of R and s. Let $D[i, s]$ be the

probability Rs is hit by x for a length-i R. Then by total probability law,

$$\begin{aligned}\Pr(\text{length-}n\ R \text{ is hit by a seed } x) &= \sum_{s' \in \{0,1\}^k} \Pr(s') \cdot \Pr(R_{n-k}s' \text{ is hit by } x) \\ &= \sum_{s' \in \{0,1\}^k} p^{\#1 \text{ in } s'}(1-p)^{\#0 \text{ in } s'} D[n-k, s']\end{aligned}$$

Then how to compute $D[i, s]$? Dynamic programming.

- Case I: s is hit by x. Then $D[i, s] = 1$.
- Case II: s is not hit by x. We have two subcases: R ends with 1 with probability p, and 0 with probability $1 - p$. Denote s' the length-$(k-1)$ prefix of s. Thus we hav the following recurrence relation:

$$D[i, s] = p \cdot D\left[i - 1, 1s'\right] + (1-p) \cdot D\left[i - 1, 0s'\right]$$

Algorithm 9: Seed x's sensitivity

```
Initialize D[0, s] // either 0 or 1, depending on whether s is hit by x
for i ← 1..n do
    for s ∈ {0,1}^k do
        if s is hit by x then
            D[i, s] = 1
        else
            D[i, s] = p · D[i − 1, 1s'] + (1 − p) · D[i − 1, 0s']
return ∑_s Pr(s) · D[n − k, s]
```

Here $\Pr(s) = p^{\#1 \text{ in } s'}(1-p)^{\#0 \text{ in } s'}$. Time complexity is $O(2^k n)$. There exists a more efficient algorithm $O(2^{\#0 \text{ in } x} n)$, which is beyond the scope of this course.

There's no good algorithm to directly optimize the spaced seed, but we can do exhaustive search to enumerate all spaced seeds with weight 11 and no longer than 18, calculate the sensitivity of each, and output the one with the highest sensitivity. This is the ONLY known algorithm that guarantees the finding of optimal seed. Many heuristics exist to find suboptimal seeds.

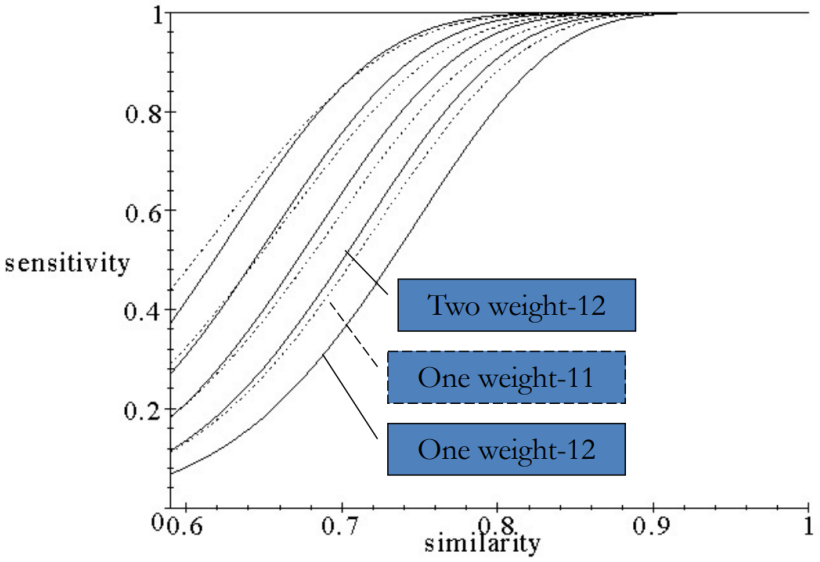
6.7 Multiple Spaced Seeds

Seeds with different shapes can detect different homologies. Some seeds may detect more homologies than others. This leads to the use of optimized spaced seed. Can use several seeds simultaneously to hit more homologies. Approaching 100% sensitive homology search. Also the overlap of seeds is waste of time.

Consider a set of seeds found by optimization algorithm: (homology identity = 0.7, homology length = 64)

```
111*11**1*11*1*111
1111***1***1**11*1*111
11**11*1**1*1***11*111
111*1***1111**1***11*1
```

To use multiple seeds, one only needs to search multiple times with different seeds, and combine results. Of course, you can search with them simultaneously. In either case, this slows down approximately k times if k seeds are used. Is it worth it? How does it compare with using one shorter seed? This is a trade-off. We can consider the tradeoff between weight of seeds and number of multiple seeds.



Solid curves: Multiple $(1, 2, 4, 8, 16)$ weight-12 spaced seeds. Dashed curves: Optimal spaced seeds with weight $= 11, 10, 9, 8$. Typically, "Doubling the seed number" gains better sensitivity than "decreasing the weight by 1".

6.8 Seeding for Proteins - BLASTP

With nucleotides, we're requiring k positions with exact matches. However, for proteins, that's not really reasonable: some amino acids mutate to another one very often, namely we don't require exact matches, but also positive score matches. So BLASTP looks for 3- or 4-letter protein sequences that are "very close" to each other, and then builds matches from them. Where very close, then total BLOSUM score in the short window is at least $+13$ for 4-mer (or $+11$ for 3-mer).

One way is to find all neighbors of 3-mer, which are all 3-mers give BLOSUM score ≥ 11. Thus we can use computer to pre-compute all neighbors of every 3-mer, and put in the index table. The details:

1. For every 3-mer, find all "neighboring" 3-mers that, score at least +11 (or whatever). Build these into a data structure NeighborList.
2. Build a hash table H for S of its 3-mers, just like for the nucleotide case
3. For every 3-mer x in T, retrieve all neighbors from NeighborList. For each neighbor, query H to find hits in S.

NeighborList is a small structure: there are only 8000 3-mers

The original way is with BLASTP:

- Build an automaton that reflects all string close to short strings in T (the short sequence)
- Scan S (the longer sequence), looking for matches.

Details: https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm

Which sequence to index? That's actually a tough question. Here's a typical scenario:

- S is the human genome (length n)
- P_1 is a short protein sequence (length m_1)

- P_2 is another short protein sequence (length m_2)

If we're smart, build an index for S, *once*, and then look up the short sequences in it. Added time for P_2 is more like $O(m_2)$, not $O(n + m_2)$.

Memory is a concern in old days, when indexing the human genome, but not anymore today. We probably should index the longer sequence. BLASTN (1990) indexes the query, not the database. BLAT (2000) indexes the database, not the query. BLASTP also indexes the query.

As an extension to this idea, consider two-hit BLAST: it requires two seeds (probably shorter) that are nearer than k from each other, and base the alignment on their enclosing box. In other words, two short hits are very close. Potentially even fewer false positives, but one has to use shorter seeds. There's quite a tradeoff here. This might lose sensitivity, but we can gain sensitivity back by reducing weight of seeds.

7

Proteomics and Mass Spectrometry

蛋白质组学和质谱法

7.1 Motivation

Primary structure of **protein** is a sequence of amino acid. 20 frequent amino acids. Fold into a complex 3D structure.

Fundamental questions Identify, sequence, and quantify all the proteins in a biological sample.

- **Identification**: determine which proteins in a database present in the sample.
- **Sequencing**: determine the amino acid sequence without needing a database.
- **Quantification**: determine the quantity change of each protein under two different biological conditions.

Example: Biomarker

HER2-positive breast cancer is a breast cancer that tests positive for a protein called human epidermal growth factor receptor 2 (HER2), which promotes the growth of cancer cells.

From Mayo Clinic:

> HER2-positive breast cancers tend to be more aggressive than other types of breast cancer. They're also less responsive to hormone treatment. However, treatments that specifically target HER2 are very effective.

Example: Immunopeptides (免疫活性肽)

Tumor (肿瘤) or infected cells "present" some abnormal peptides at the MHC on cell surface. CD8+ T cells (aka T killer cells) recognize the abnormal peptides and kill the cell. An actively pursued method in immunotherapy is to identify/predict the peptides presented and train the T cells to target them. Mass spec is the best tool to identify these peptides.