SequenceAlignment

September 6, 2018

1 Lecture 8: Sequence Alignment

CBIO (CSCI) 4835/6835: Introduction to Computational Biology

1.1 Overview and Objectives

In our last lecture, we covered the basics of molecular biology and the role of sequence analysis. In this lecture, we'll dive deeper into how sequence analysis is performed and the role of algorithms in addressing sequence analysis. By the end of this lecture, you should be able to:

- Define the notion of algorithmic complexity and how it relates to sequence alignment and analysis
- Describe and define the abstract problems of shortest common superstring (SCS) and longest common substring (LCS), and how they specifically relate to sequence analysis
- Recall different methods of scoring sequence alignments and their advantages and drawbacks
- Describe the different distance metrics and methods of scoring sequence alignments
- Explain why local or global sequence alignments are preferred in certain situations

1.2 Part 0: 'range"

This mysterious `range` function has showed up a few times so far. What does it do?

```
In [1]: r = range(10)
        print(r)
```

```
range(0, 10)
```

Not terribly useful output information, to be fair.

```
In [2]: r = range(10)
        l = list(r)  # cast it as a list
        print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range(i)` **generates a list of numbers from 0 (inclusive) to `i` (exclusive).** This is very useful for looping!

```
In [3]: for i in range(10):
            print(i)

0
1
2
3
4
5
6
7
8
9
```

You can also provide a *second* argument to `range`, which specifies a *starting* point for the counting (other than 0). That starting point is still *inclusive*, and the ending point still *exclusive*.

```
In [4]: for i in range(5, 10):
            print(i)

5
6
7
8
9
```

Finally, you can also provide a *third* argument, which specifies the *interval* between numbers in the output. So far, that interval has been 1: start at 0, to go `i`, by ones. You can change that "by ones" to whatever you want.

```
In [5]: for i in range(5, 10, 2):  # read: from 5, to 10, by 2
            print(i)

5
7
9
```

You can get really crazy with this third one, if you want: you can go *backwards* by putting in a negative interval.

```
In [6]: for i in range(10, 0, -2): # from 10, to 0, by -2
            print(i)
```

```
10
8
6
4
2
```

Same rules apply, though: the starting point is *inclusive* (hence why we see a 10), and the ending point is *exclusive* (hence why we *don't* see a 0).

`range` is particularly useful as a way of looping through a list of items *by index*.

```
In [7]: list_of_interesting_things = [93, 17, 5583, 47, 2359875, 4, 381]
        for item in list_of_interesting_things:
            print(item, end = " ")

93 17 5583 47 2359875 4 381
```

This is how we've seen loops so far: the loop variable (here it's `item`) is a literal item in the list. **But what if, in addition to the item, I needed to know *where* in the list that item was** (i.e., the item's list index)?

```
In [8]: list_length = len(list_of_interesting_things)
        for index in range(list_length):  # use range of the list length!
            item = list_of_interesting_things[index]  # pull out the item AT that index
            print("Item " + str(item) + " at index " + str(index))

Item 93 at index 0
Item 17 at index 1
Item 5583 at index 2
Item 47 at index 3
Item 2359875 at index 4
Item 4 at index 5
Item 381 at index 6
```

1.3 Part 1: Complexity

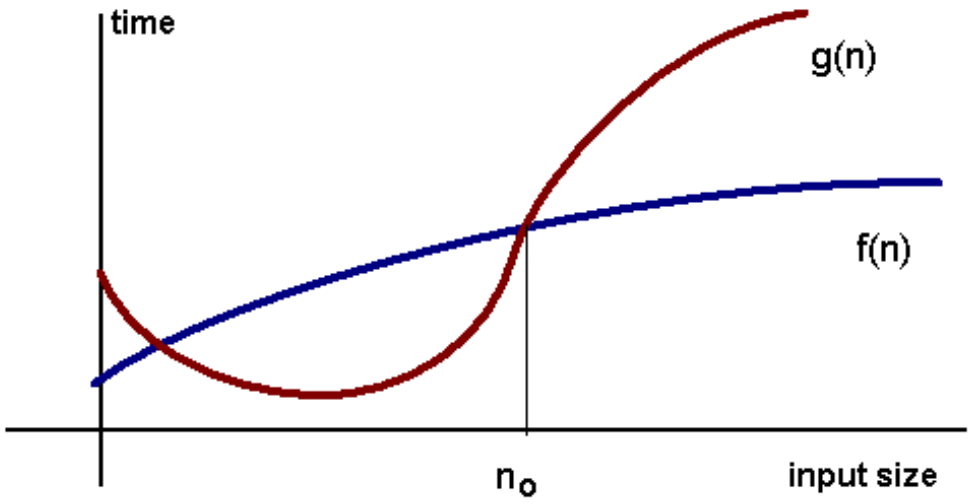
1.3.1 Big "Oh" Notation

From computer science comes this notion: how the runtime of an algorithm changes with respect to its input size.

$\mathcal{O}(n)$ - the "$\mathcal{O}$" is short for "order of the function", and the value inside the parentheses is always with respect to n, interpreted to be the variable representing the size of the input data.

1.3.2 Limits

Big-oh notation is a representation of limits, and most often we are interested in "worst-case" runtime. Let's start with the example from the last lecture.



bigoh

```
In [9]: a = [1, 2, 3, 4, 5]
        for element in a:
            print(element)
```

```
1
2
3
4
5
```

How many steps, or iterations, does this loop require to run?

Alright, back to complexity:

```
In [10]: a = range(100)
         for element in a:
             print(element, end = " ")
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 3
```

How many iterations does this loop require?

For iterating once over any list using a single `for` loop, how many iterations does this require?

Algorithms which take n iterations to run, where n is the number of elements in our data set, are referred to as running in $\mathcal{O}(n)$ time.

This is roughly interpreted to mean that, for n data points, n processing steps are required.

Important to note: we never actually specify *how much time* a single processing step is. It could be a femtosecond, or an hour. Ultimately, it doesn't matter. What does matter when something is $\mathcal{O}(n)$ is that, if we add one more data point $(n+1)$, then however long a single processing step is, the algorithm should take only that much longer to run.

How about this code? What is its big-oh?

```
In [11]: a = range(100)
         b = range(1, 101)
         for i in a:
             print(a[i] * b[i], end = " ")
```

```
0 2 6 12 20 30 42 56 72 90 110 132 156 182 210 240 272 306 342 380 420 462 506 552 600 650 702 7
```

Still $\mathcal{O}(n)$. The important part is not (directly) the number of lists, but rather how we operate on them: again, we're using **only 1** `for` **loop**, so our runtime is directly proportional to how long the lists are.

How about this code?

```
In [12]: a = range(100)
         x = []
         for i in a:
             x.append(i ** 2)

         for j in a:
             x.append(j ** 2)
```

Trick question! One loop, as we've seen, is $\mathcal{O}(n)$. Now we've written a second loop that is also $\mathcal{O}(n)$, so literally speaking the runtime is $2 * \mathcal{O}(n)$, but what happens to the 2 in the limit as $n \rightarrow \infty$?

The 2 is insignificant, so the overall big-oh for this code is still $\mathcal{O}(n)$.

How about this code?

```
In [13]: a = range(100)
         for element_i in a:
             for element_j in a:
                 print(element_i * element_j, end = " ")
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Nested `for` loops are brutal–the inner loop runs in its entirety for every single iteration of the outer loop. In the limit, for a list of length n, there are $\mathcal{O}(n^2)$ iterations.

One more tricky one:

```
In [14]: xeno = 100
         while xeno > 1:
             xeno /= 2
             print(xeno, end = " ")
```

```
50.0 25.0 12.5 6.25 3.125 1.5625 0.78125
```

Maybe another example from the same complexity class:

```
In [15]: xeno = 100000
         while xeno > 1:
             xeno /= 10
             print(xeno, end = " ")
```

```
10000.0 1000.0 100.0 10.0 1.0
```

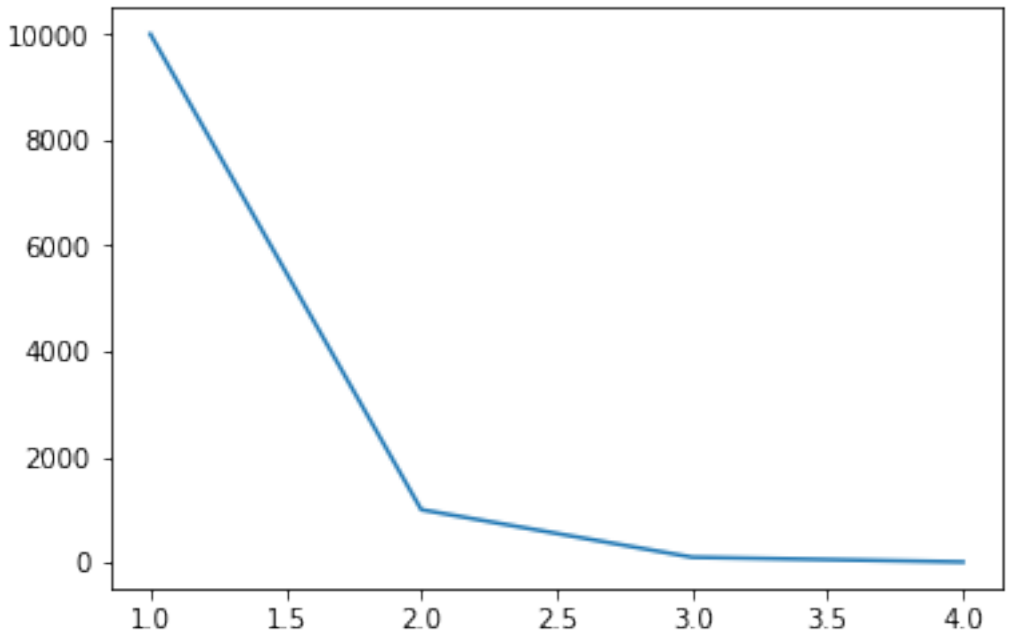
What does this "look" like?

```
In [16]: # I'm just plotting the iteration number against the value of "xeno".

         %matplotlib inline
         import matplotlib.pyplot as plt

         x = []
         y = []
         xeno = 10000
         i = 1
         while xeno > 1:
             x.append(i)
             y.append(xeno)
             xeno /= 10
             i += 1
         plt.plot(x, y)

Out[16]: [<matplotlib.lines.Line2D at 0x11e143c18>]
```



In the first one, on each iteration, we're dividing the remaining space by 2, halving again and again and again.

In the second one, on each iteration, we're dividing the space by 10.

$\mathcal{O}(\log n)$. We use the default (base 10) because, in the limit, constants don't matter.

1.4 Part 2: SCS and LCS

Recall from the last lecture what SCS (shortest common superstring) was:

- The shortest common superstring, given sequences X and Y, is the shortest possible sequence that contains all the sequences X and Y.

For example, let's say we have X = `ABACBDCAB` and Y = `BDCABA`. What would be the shortest common superstring?

Here is one alignment: `BDCABA` (second string) and `ABACBDCAB` (first string). The `ABA` is where the two strings overlap. The full alignment, `BDCABACBDCAB`, has a length of 12.

Can we do better?

`ABACBDCAB` and `BDCABA`, which gives a full alignment of `ABACBDCABA`, which has a length of only 10. So this alignment would be the SCS.

(When do we need to use SCS?)

1.4.1 Longest Common Substring (LCS)

In a related, but different, problem: longest common substring asks:

- Given sequences X and Y, the longest common substring is the constituent of the sequences X and Y that is as long as possible.

Let's go back to our sequences from before: X = `ABACBDCAB` and Y = `BDCABA`. What would be the longest common substring?

The easiest substrings are the single characters `A`, `B`, `C`, and `D`, which both X and Y have. But these are short: only length 1 for all. Can we do better?

`ABACBDCAB` and `BDCABA`, so the longest common substring is `BDCAB`.

(When do we need LCS?)

1.4.2 Rudimentary Sequence Alignment

Given two DNA sequences v and w:

v: `ATATATAT`

w: `TATATATA`

How would you suggest aligning these sequences to determine their similarity?

Before we try to align them, we need some objective measure of what a "good" alignment is!

1.5 Part 3: Distance Metrics

Hopefully, everyone has heard of *Euclidean distance*: this is the usual "distance" formula you use when trying to find out how far apart two points are in 2D space.

How is it computed?

For two points in 2D space, a and b, their Euclidean distance $d_e(a, b)$ is defined as:

$d_e(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2)}$

So if $a = (1, 2)$ and $b = (5, 3)$, then:

$d_e(a, b) = \sqrt{(1-5)^2 + (2-3)^2} = \sqrt{(-4)^2 + (-1)^2} = \sqrt{16+1} = 4.1231$

How can we measure distance between two sequences?

There is a metric called **Hamming Distance**, which counts number of differing corresponding elements in two strings.

We'll represent the Hamming distance between two strings v and w as $d_H(v, w)$.

v: `ATATATAT`

w: `TATATATA`

$d_H(v, w) = 8$

That seems reasonable. But, given how similar the two sequences are (after all, the LCS of these two is 7 characters), what if we shifted one of the sequences over by one space?

v: `ATATATAT-`

w: `-TATATATA`

Now, what's $d_H(v, w)$?

$d_H(v, w) = 2$

The only elements of the two strings that don't overlap are the first and last; they match perfectly otherwise!

1.5.1 Edit distance

Hamming distance is useful, but it neglects the possibility of insertions and deletions in DNA (what is the only thing it counts?). So we need something more robust.

The *edit distance* between two strings is the *minimum number of elementary operations* (insertions, deletions, or substitutions / mutations) required to transform one string into the other.

Hamming distance: i^{th} letter of v with i^{th} letter of w (how hard is this to do?)

Edit distance: i^{th} letter of v with j^{th} letter of w (how hard is this to do?)

Hamming distance is easy, but gives us the wrong answers. Edit distance gives us much better answers, but it's hard to compute: **how do we know which i to pair with which j?**

What's the edit distance for v = `TGCATAT` and w = `ATCCGAT`?

One solution:

1. TGCATA**T** (delete last T)
2. TGCAT**A** (delete last A)
3. **A**TGCAT (insert A at front)
4. AT**C**CAT (mutate G to C)
5. ATCC**G**AT (insert G before last A)

`ATCCGAT` == `ATCCGAT`, done in 5 steps!

Can it be done in 4 steps?

(. . . mmmmaybe–but that's for next week!)

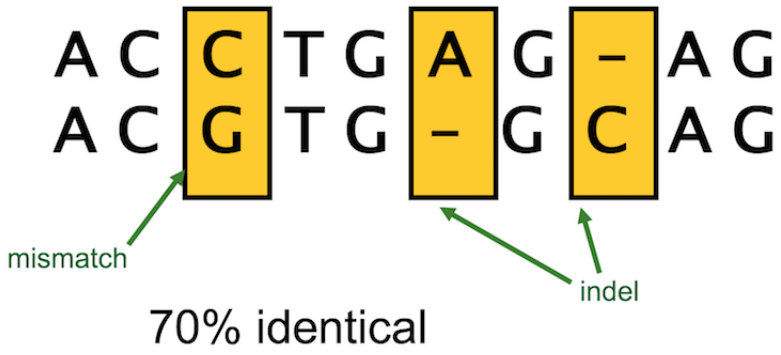
1.6 Part 4: Global vs Local Alignment

indel is a portmanteau of "insertion" and "deletion", so we don't need to worry about which strand we're actually referring to.

What is the edit distance here?

1.6.1 Highly conserved subsequences

Things get hairier when we consider that two genes in different species may be similar over **short, conserved regions** and dissimilar over remaining regions.



dnasequence

Homeobox regions have a short region called the *homeodomain* that is highly conserved among species–responsible for regulation of patterns of anatomical development in animals, fungi, and plants.

- A global alignment would not find the homeodomain because it would try to align the *entire* sequence.
- Therefore, we search for an alignment which has a low edit score *locally*, meaning we have to search aligned substrings of the two sequences.

Here's an example global alignment that minimizes edit distance over the entirety of these two sequences:

Here's an example local alignment that may have an *overall larger edit distance*, but it finds the highly conserved substring:

"BUT!", you protest.

"If the local alignment has a higher edit score, how do we find it at all?"

We've already seen that we need to consider three separate possibilities when aligning sequences:

1. Insertions / Deletions (characters added or removed)
2. Mutations / Substitutions (characters modified)
3. Matches (characters align)

With Hamming distance, two characters were either the same or they weren't (options 1 and 2 above were a single criterion).

With edit distance, we separated #1 and #2 above into their own categories, but they are still weighted the same (1 insertion = 1 mutation = 1 edit)

Are all insertions / deletions created equal? How about all substitutions?

1.6.2 Scoring Matrices

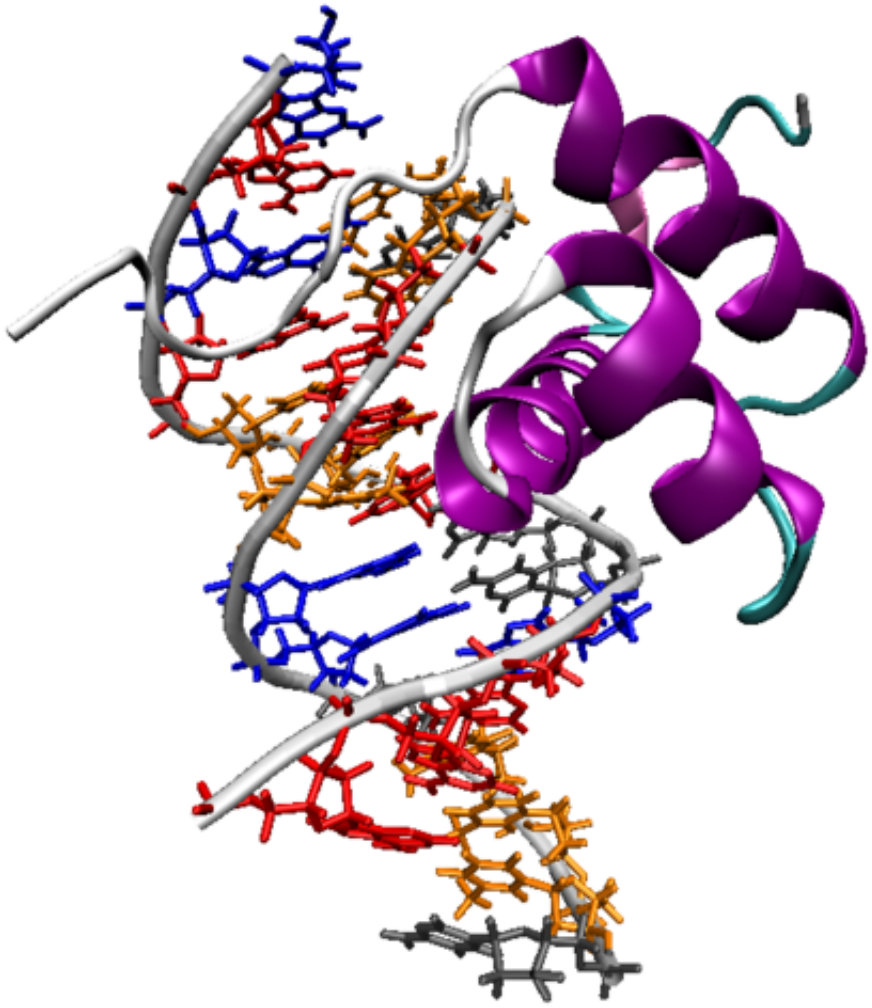
Say we want to align the sequences:

v = `AGTCA`

w = `CGTTGG`

But instead of using a standard edit distance as before, I give you the following *scoring matrix*:

This matrix gives the specific edit penalties for particular substitutions / insertions / deletions.



homeobox

```
--T--CC-C-AGT--TATGT-CAGGGGACACG—A-GCATGCAGA-GAC
  |   || |   ||   | | | |||     || | | |   | ||||    |
AATTGCCGCC-GTCGT-T-TTCAG----CA-GTTATG—T-CAGAT--C
```

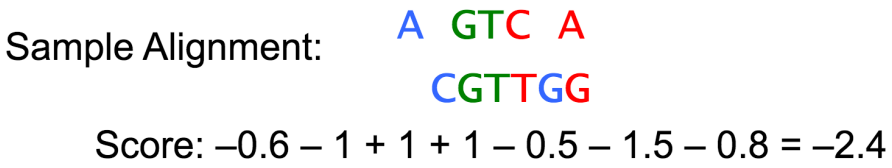
global

```
                    tccCAGTTATGTCAGgggacacgagcatgcagagac
                       ||||||||||||
aattgccgccgtcgttttcagCAGTTATGTCAGatc
```

local

	A	G	T	C	—
A	1	-0.8	-0.2	-2.3	-0.6
G	-0.8	1	-1.1	-0.7	-1.5
T	-0.2	-1.1	1	-0.5	-0.9
C	-2.3	-0.7	-0.5	1	-1
—	-0.6	-1.5	-0.9	-1	n/a

dnascoring



samplealignment

It also allows us to codify our understanding of biology and biochemistry into how we define a "good" alignment. For instance, this penalizes matching A with G more heavily than C matched with T.

Here is a sample alignment using this scoring matrix:

1.6.3 Making a scoring matrix

Scoring matrices are created based on biological evidence.

Some mutations, especially in amino acid sequences, may have little (if any!) effect on the protein's function. Using scoring matrices, we can directly quantify that understanding.

- Polar to polar mutations (aspartate -> glutamate)
- Nonpolar to nonpolar mutations (alanine -> valine)
- Similarly behaving residues (leucine -> isoleucine)

1.6.4 Standard scoring matrices

For nucleotide sequences, there aren't really "standard" scoring matrices, since DNA is less conserved overall and less effective to compare coding regions.

There are, however, some common amino acid scoring matrices. We'll discuss two:

1. PAM (**P**oint **A**ccepted **M**utation)
2. BLOSUM (**Blo**cks **Su**bstitution **M**atrix)

1.6.5 PAM

PAM is a more theoretical model of amino acid substitutions.

It is always associated with a number, e.g. 1 PAM, written as PAM_1. This means the given PAM_1 scoring matrix is built to reflect a **1% average change in all amino acid positions** of the polypeptide, according to evolution.

	Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	...
	A	R	N	D	C	Q	E	G	H	I	L	K	...
Ala A	13	6	9	9	5	8	9	12	6	8	6	7	...
Arg R	3	17	4	3	2	5	3	2	6	3	2	9	
Asn N	4	4	6	7	2	5	6	4	6	3	2	5	
Asp D	5	4	8	11	1	7	10	5	6	3	2	5	
Cys C	2	1	1	1	52	1	1	2	2	2	1	1	
Gln Q	3	5	5	6	1	10	7	3	7	2	3	5	
...													
Trp W	0	2	0	0	0	0	0	0	1	0	1	0	
Tyr Y	1	1	2	1	3	1	1	1	3	2	2	1	
Val V	7	4	4	4	4	4	4	4	5	4	15	10	

pam250

Some important notes: - This is an *average*. Even with PAM_{100}, not every residue will have changed (some are more conserved than others) - Some residues may have mutated several times! - Some residues may have mutated back to their original state! - Some residues may not have changed at all

PAM_{250} is a widely used scoring matrix.

Mutating A to A is clearly the most preferable (highest score in that row of 13 points), but after 250 evolutions, a mutation from A to G also seems very favorable (12 points).

1.6.6 BLOSUM

Unlike PAM, scores in BLOSUM are derived from direct empirical observations of the frequencies of substitutions in blocks of local alignments in related proteins. They both, however, often obtain identical alignment scores.

Like PAM, BLOSUM also has a number associated with it, this time to represent the observed substitution rate between two proteins sharing some amount of similarity.

$BLOSUM_{62}$ is a common scoring matrix, representing substitution rates in proteins sharing no more than 62% identity.

1.7 Next week

We'll look at how to use these matrices to determine the best alignments of sequences!

1.8 Administrivia

- **You should everything you need now for Assignment 2.** Any questions so far? Due a week from today!
- **We [briefly] jump back on the Python train next week**.
- **We also have a guest lecturer!** More details on Tuesday.

1.9 Additional Resources

1. Jones, Neil C. and Pevzner, Pavel A. *An Introduction to Bioinformatics Algorithms*, Chapter 6. 2004. ISBN-13: 978-0262101066
2. Based heavily on the modified slides of Dr. Phillip Compeau.

	C	S	T	P	…	F	Y	W
C	9	-1	-1	3	…	-2	-2	-2
S	-1	4	1	-1	…	-2	-2	-3
T	-1	1	4	1	…	-2	-2	-3
P	3	-1	1	7	…	-4	-3	-4
…	…	…	…	…	…	…	…	…
F	-2	-2	-2	-4	…	6	3	1
Y	-2	-2	-2	-3	…	3	7	2
W	-2	-3	-3	-4	…	1	2	11

blosum62